Neural Networks and Optimization Problems
A Case Study: The Minimum Cost Spare Allocation Problem

Michail G. Lagoudakis
The Center for Advanced Computer Studies
University of Southwestern Louisiana
P.O. Box 44330, Lafayette, LA 70504
mgl4822@usl.edu

Abstract

An *optimization problem* can be defined as a pair of an objective function and a set of constraints on the variables of the function. The goal is to find out the values of the variables that lead to an optimal value of the function (either minimum or maximum), while satisfying all the constraints. During the last decades, an alternative model of computation has been explored, namely the *neural network model*. It turned out that several Hopfield-type networks can be employed successfully to provide approximate (near-optimal or even optimal) solutions to hard optimization problems. This is due to the property of reducing their "energy function" during evolution, leading to a local or global minimum. In this report, the general methodology of the approach is described as well as the different network models usually employed as optimizers. Then, a case study involving the Minimum Cost Spare Allocation Problem (or equivalently Vertex Cover in bipartite graphs) is presented. Finally, the experimental results (using a simulation implemented in C) demonstrate clearly the advantages and the limitations of the approach in terms of solution quality and computation time.

Keywords: Hopfield Neural Network, Boltzmann Machine, Cauchy Machine, Optimization Problems.

1 Overview

This project report contains a survey of the methodology used to apply neural networks on optimization problems in order to derive approximate (near-optimal) solutions and a case study with simulation implemented in C. It came out as a semester project for the CMPS 523 course, "The Computational Basis of Intelligence", offered by Dr. Anthony Maida in Spring 1997.

Section 2 introduces the concept of an optimization problem and section 3 presents the Hopfield neural network model along with its characteristics and variations. Section 4 explains how optimization problems can be mapped on neural networks using the penalty function method and section 5 presents the different models of networks most frequently used as optimizers, namely the analog Hopfield network, the Boltzmann machine, the Cauchy machine and a Hybrid scheme based on both the Boltzmann and Cauchy machine. Finally, in section 6 the case study is presented as well as the corresponding experimental results.

2 Optimization Problems

The optimization problems constitute a large class of difficult, in general, problems which has attracted the interest of the research community and gave rise to a research area known as Combinatorial Optimization. In such problems, we try to optimize (maximize or minimize) some quantity, while satisfying some constraints. The term "combinatorial" comes from the fact that typically there is a large number of possible (feasible) solutions, like $O(2^N)$ or $N!$, where N is the size of the problem, of which we want only the best.

More formally an *optimization problem* can be defined as a pair of an objective function $f(x_1, x_2, x_3, \ldots, x_n)$ and a set of constraints C on the variables of the function. The goal is to find out the values of the variables x_i that lead to an optimal value of the function f (either minimum or maximum), while satisfying all the constraints in C. Usually, the variables x_i are constrained to discrete values. Depending on the desired optimal value we can distinguish between minimization and maximization problems. Also, depending on the linearity or non-linearity of the objective function we can distinguish between linear and non-linear optimization.

Optimization problems are of particular interest in the area of Operations Research and they appear in many real situations. However, their complexity (most of them are NP-hard problems) makes it difficult to be solved in practice using the conventional approaches and computing devices.

3 Neural Networks

3.1 History

During the last decades, an alternative model of computation has been explored, namely the *Neural Networks*. Their origins go back to several attempts to model the human brain function and many impressive results have been derived so far. In 1982, John Hopfield triggered the research interest in the area with a short paper [Hopf82] demonstrating the computational capabilities of networks with symmetric connections. Such networks were previously considered as not worth studying because they are not brain-like. The main contribution of the paper was the introduction of the energy function of the network which turns to be their most useful property in the context of optimization. This neural network model was named after him and comes with many variations.

In 1985 and 1987, Hopfield and Tank published some results ([HoTa85], [HoTa87]) on how to go about using neural networks to solve optimization problems. The first problem formulated in terms of neural network was the Travelling Salesman Problem and the work mentioned above demonstrated how circuits of simple unit can solve hard problems. However, the problem of local minima is the main limitation of the approach and several techniques have been proposed to overcome it, like stochastic networks, simulated annealing, etc. On the other hand, the parallel nature of some neural network models seems to be very promising in reducing computation time. Two recent published volumes ([ChUn93], [WaTa96]) contain most of the work on the area of neural optimization for the demanding readers.

3.2 Hopfield Networks

The *Hopfield Neural Network Model* consists of a fully connected network of units (or neurons). The connections between the units are weighted; for any two units i & j, w_{ij} is the weight of the connection

between them. The model assumes symmetrical weights (w_{ij} = w_{ji}) and in most of the cases zero self-coupling terms (w_{ii} = 0). A connection with positive weight is an *excitatory* connection, as opposed to an *inhibitory* connection which has negative weight. A unit *i* is characterized by its *output* (or *state*) v_i, the *activation* (or *network input*) u_i that receives from the other units and a *threshold* θ_i. The *network state* is given by the output (or state) vector $\vec{v}$ = (v_1,v_2, ... ,v_n).

Each unit receives input from all the other units and forwards its output to all the other units. The way the output of a unit is updated is defined by the *dynamics* of the network. In general, the output behavior is described as a function of the activation, where the activation depends on the weighted summation of the inputs and the threshold. The McCullogh and Pitts [McPi43] dynamics rule, usually employed in Hopfield networks, has the following form[1]:

$$v_i(t+1) = \Phi(u_i(x)) = \Phi\left(\sum_{j=1}^{n} w_{ij} v_j(t) + \theta_i\right) \tag{1}$$

The function $\Phi(x)$ can take several forms. It can be the unit *step* function, (or *Heaviside*) function

$$\Theta(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{2}$$

in which case we have a discrete Hopfield network with binary states {0, 1}. If the *sign* function

$$Sign(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases} \tag{3}$$

is used, the network is discrete but with values {1, –1}. Finally, a network with continuous-valued units (analog Hopfield network), where the values fall in the range [0, 1] or [-1, 1], can be obtained by employing the *sigmoid* (or *logistic*) function

$$g_\beta(x) = \frac{1}{1 + e^{-2\beta x}} \tag{4}$$

or the *hyperbolic tangent* function

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5}$$

respectively. Note that

$$Sign(x) = 2\Theta(x) - 1 \qquad and \qquad \tanh(\beta x) = 2g_\beta(x) - 1 \tag{6}$$

so the {0, 1} (or [0, 1]) model can be easily transformed to {-1, 1} (or [-1, 1]) and vice-versa. In the limit β–>∞ the values become discrete.

All the networks above are *deterministic* in the sense that the next state is an explicit function of the previous state and the characteristics of the network. In the case of the *stochastic* Hopfield networks the probability of a unit to be in a particular state is drawn from a probability distribution, e.g. Boltzmann

[1] Note that throughout this report a threshold value θ_i is viewed as a negative number added to the weighted input, as opposed to a positive value subtracted from the weighted input.

or Cauchy. Alternatively, the stochastic network can be viewed as a deterministic network, where the threshold of a unit is variable and is drawn from a probability density.

Another distinction comes out from the updating policy. It can be *synchronous*, in which case all the units are updated simultaneously at each time step, or *asynchronous*, where either the units are updated in sequence one at each time step (*sequential* asynchronous update) or at each time step one randomly chosen unit is updated (*random* asynchronous update).

Concluding this section, the properties that characterize different Hopfield networks are summarized in the following list:

1. Discrete vs Continuous-Valued Units
2. Deterministic vs Stochastic Networks
3. Synchronous vs Asynchronous Update (Sequential or Random)

3.3 Energy Function

The *energy function* of a Hopfield network is a function defined over the state space of the network and has the following form:

$$E(\vec{v}) = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} v_i v_j w_{ij} - \sum_{i=1}^{n} v_i \theta_i \tag{7}$$

The main property of the function above is that it always decreases (not necessarily monotonically) during the evolution of the network. This is due to the dynamics of the systems. Alternatively, the energy function can be viewed as defining an energy landscape and the dynamics can be thought of as the motion of a small sphere on the energy surface under the influence of gravity and friction. Consequently, the network performs as a minimizer of the energy function and will be trapped soon, during its evolution, into a local or global minimum of the function. This property is crucial for the optimization purposes.

It is easy to show that, whenever a unit i changes state, the *energy difference* ΔE_i is given by the following formula:

$$\Delta E_i = -u_i \Delta v_i = -\left(\sum_{j=1}^{n} w_{ij} v_j + \theta_i\right)\Delta v_i \tag{8}$$

Consider the {0, 1} network with dynamics given by (1) and (2). If the activation is positive, then either Δu_i=0 (1–>1) or Δu_i=1 (0–>1) and thus $\Delta E_i \leq 0$. Similarly, if the activation is negative, then either Δu_i=0 (0–>0) or Δu_i=–1 (1–>0) and again $\Delta E_i \leq 0$. The argument is similar for the other models.

It should be noted that, the main property of the energy function described above is held only if the weights of the connections are symmetric and the self-coupling terms are zero or positive. The Hopfield network, as defined previously, fits these requirements and thus its energy function is a, so called, *Lyapunov* function.

4 Mapping Problems on Networks

4.1 The Idea

In the previous section, it was shown that a Hopfield network performs as a minimizer of its energy function. The application of such neural networks on optimization comes out from the following idea:

If the problem can be coded as an energy function, with the property that the better the solution, the lower the energy level, then a network that corresponds to this energy function can be used to minimize (locally or globally) the function and thus provide an optimal or near-optimal solution. So, the procedure begins with the construction of the energy function and then the parameters of the network (number of units, weights of connection, thresholds, update policy or even the dynamics) are adjusted to reflect the problem. Then the network is initialized to some initial state and is let to run until it comes to equilibrium from where a solution can be drawn.

4.2 The Penalty Function

The task of constructing the appropriate energy function is not, in general, an easy task. The basic constraint is that it must be quadratic in order to meet the form (7). The most common approach is the penalty (or cost) function approach [LLHZ93]. The energy function is initialized to the objective function of the problem and for each constraint a penalty term is added. Thus, the problem from a constrained optimization form is reduced to an unconstrained minimization problem. If $f(\vec{x})$ is the objective function of the problem and *C* the set of constraints, then the energy function will have the form:

$$E(\vec{x}) = mf(\vec{x}) + \sum_{c \in C} a_c P_c(\vec{x}) \tag{9}$$

where *m*=+1 for minimization problems or *m*=-1 for maximization problems. The penalty term $P_c(\vec{x})$ for each constraint $c \in C$ has the property that it is zero if and only if the corresponding constraint *c* on $\vec{x}$ is satisfied, otherwise it has a positive value (where possible it is desirable for this value to increase with respect to the degree of violation). Finally, the constants α_c define the relative weight concerning the satisfaction of some constraints against some other and/or the relative weight of satisfying all the constraints or minimizing $sf(\vec{x})$. The task of tuning these parameters is generally a hard task, but at least it provides a means of customizing the problem according to the current needs. In the extreme case, it can be $\alpha_c = \alpha_c(t)$, so the parameters can be adjusted dynamically during the evolution of the system.

The constraints in *C* are in general equality or inequality constraints. In the sequel, the way these two cases are attacked is presented.

4.3 Equality Constraints

An equality constraint has the general form

$$c_k : \quad \sum_j x_j = \gamma_k \tag{10}$$

where γ_k is a constant. Since the energy function should be quadratic (at most), an appropriate penalty term for such constraints is the following:

$$P_{c_k}(\vec{x}) = \left(\sum_j x_j - \gamma_k \right)^2 \tag{11}$$

The penalty term is zero if and only if the constraint is satisfied, otherwise the penalty is proportional to the degree of violation.

4.4 Inequality Constraints

An inequality constraint has the general form

$$c_k : \quad \sum_j x_j \leq \gamma_k \tag{12}$$

where γ_k is a constant. The penalty term for such constraints has the following general form:

$$P_{c_k}(\vec{x}) = \Omega\left(\sum_j x_j - \gamma_k\right) \tag{13}$$

The function $\Omega(y)$ should penalize only configurations where *y*>0. One possible choice for $\Omega(y)$ is a sigmoid function (eq. 4), with the disadvantage of having the same penalty (=1) independently of the degree of violation. A better alternative, proposed in [OhPS93] is

$$\Omega(y) = y\Theta(y) \tag{14}$$

using the step function (eq. 2). The penalty term is zero if and only if the constraint is satisfied, otherwise the penalty is proportional (linear) to the degree of violation. The slope of $\Omega(y)$ is implicitly given by the strength of the constraint α_c (see eq. 9).

4.5 Deriving the Network

As soon as the energy function has been constructed, the elements of the network (number of units, weights, thresholds) can be derived. Generally, the number of units depends on the number of *x*'s in the array $\vec{x}$. If all *x*'s are binary, then one unit per *x* is needed. As far as concerning the weights and the thresholds, equation (8) is used. For each unit *i* the *energy difference* ΔE_i is calculated (derivatives can be used for this purpose) and it is transformed into a form similar to (8). By analogy, the coefficient of v_j will give the weight w_{ij}, whereas the constant terms will give the threshold θ_i. In most of the cases, these values can be easily verified by common reasoning with respect to the problem under investigation.

One difficulty, at this point, rises from the fact that as the problem size grows, the network size (units and connections can be intractable large and inapplicable to practical domains. The limitations are due to storage and computing power requirements. For the second, several techniques for parallel implementations have been proposed.

4.6 Initializing the Network

An important issue is how to initialize the network state before you let it run. If the network is initialized to a state that corresponds to a possible solution, it will stuck there since all the constraints are satisfied, although this solution may not be optimal. Such states corresponds to local minima and are not desirable as initial states. Moreover, different initialization may lead to different solutions, with different quality and/or computation time. In most cases, a random state is used for initialization.

5 The Optimizers

In this section several different models of networks used for optimization are presented. The fist two are simple cases with many disadvantages, whereas the rest four models are more sophisticated and attempt to overcome the limitations of the first two. For the rest, it is assumed that we attempt to solve discrete optimization problems.

5.1 Discrete Synchronous Hopfield Network

The operation of this model is rather simple. The dynamics of the system are given by (1) and either (2) or (3) and all the units are updated in parallel. However, the evolution of the network is not continuous over the edges of the state hypercube (for discrete units) leading to oscillation phenomena, where the network is trapped oscillating between two states. Moreover, the network is not guaranteed to decrease at each step and since the system may fail to come into equilibrium a solution is not always derived. On the other hand, it has the advantage of being fully parallelizable.

5.2 Discrete Asynchronous Hopfield Network

This model is similar to the previous one with the difference that the units are updated sequentially. As a result, the oscillation phenomena are eliminated and the energy function is guaranteed to decrease at each step. The network will eventually come into equilibrium, from where a solution can be drawn. However, the system can be easily trapped into a local minimum and is highly dependent on the initial state. Additionally, it cannot be parallelized since it is strictly sequential.

5.3 Analog Hopfield Network

The dynamics of the model are given by (1) and either (4) or (5). In the context of optimization, synchronous, asynchronous or continuous updating can be applied. Although the output of such networks is continuous they can be used to solve discrete optimization problems. In the equilibrium state the outputs closer to 1 are taken as 1 and the output closer to 0 (-1) are taken as 0 (-1). The potential disadvantage of this network is that it may be trapped in local minima inside the hypercube without reaching any of the corners.

The parameter β of the sigmoid function is usually taken as $\beta = \frac{1}{T}$ where *T* is a virtual temperature. The temperature *T* adjusts the sharpness of the sigmoid (or hyperbolic tangent) function and at the limit *T*->0 (absolute temperature) the output becomes discrete. If we start the network at the temperature where we want to measure the outputs, it may take a long time to come to equilibrium Usually, the *simulated annealing* technique is applied. We start the network at a relatively high temperature and gradually we cool it down. This technique helps the system to converge faster avoiding most of the local minima. Also, as the temperature is lowered the outputs tend to be more and more discrete. Another advantage of this model is that it can be used to solve iteratively mean field problems. This approach is known as *mean field annealing*.

5.4 Boltzmann Machine

The operation of the Boltzmann machine integrates the dynamics of the discrete asynchronous Hopfield model with the simulated annealing technique. At each step *t*, a unit *i* of the network is selected randomly and the energy difference ΔE_i that will be caused by a change of its state is calculated using (8). If ΔE_i is negative (i.e. the energy is decreased) the change is definitely accepted, otherwise it is accepted with probability $P_i^B(t)$ that depends on the quantity $e^{(\Delta E_i/T_B)}$. T_B is a temperature that decreases according to some annealing schedule.

Usually, the *Metropolis criterion* is used as the acceptance criterion, which is given by

$$P_i^B(t) = \begin{cases} 1 & \text{if } \Delta E_i(t) < 0 \\ \frac{1}{1+e^{\left(\frac{\Delta E_i(t)}{T_B(t)}\right)}} & \text{if } \Delta E_i(t) \geq 0 \end{cases} \tag{15}$$

The *logarithmic schedule* can be used to decrease the temperature:

$$T_B(t) = \frac{T_B(t-1)}{1 + t \log(1+r)} \tag{16}$$

where *r* is a parameter that adjusts the speed of the schedule.

The Boltzmann machine can be very effective when used with the appropriate annealling schedule and is able to perform a wide exploration of the problem state space. However, it is strictly sequential and cannot be parallelized. Several attempts to parallelize its operation (e.g. group updates) have to cope with some kind of trade-off between the solution quality and the actual speedup.

5.5 Cauchy Machine

The Cauchy machine extends the discrete synchronous Hopfield model. The behavior of a unit *i* at each time step *t* is given by

$$v_i(t) = \Theta(u_i(t)) \tag{17}$$

During operation, the activation is updated using the following *motion equation*:

$$\frac{du_i(t)}{dt} = -\frac{\partial E(\vec{v})}{\partial v_i} \tag{18}$$

For simulation purposes, the first-order approximation of the dynamics above is used:

$$\frac{\Delta u_i}{\Delta t} = -\frac{\Delta E_i}{\Delta v_i} \qquad \text{and} \qquad u_i(t+\Delta t) = u_i(t) + \Delta u_i \tag{19}$$

Note that the activation plays the role of an accumulator, a kind of a memory that stores the cumulative activation of the unit during the network's operation time. In that way, the probability of two units to change state simultaneously is reduced significantly, thus oscillation phenomena are avoided and the system will eventually come to equilibrium. This is strengthened also by the fact that each unit follows gradient descent dynamics (see eq. (18) & (19) above). When the energy function is given by (7), then using (8) and (19), we can derive the motion equation for each unit *i*:

$$\frac{\Delta u_i}{\Delta t} = \sum_{j=1}^{n} w_{ij} v_j + \theta_i \qquad \text{and} \qquad u_i(t+\Delta t) = u_i(t) + \left(\sum_{j=1}^{n} w_{ij} v_j + \theta_i \right) \Delta t \tag{20}$$

A state vector $\vec{v}$ constitutes an equilibrium state for the network if for all *i* the following condition is satisfied

$$(v_i = 1 \ \text{ and } \Delta u_i \geq 0) \text{ or } (v_i = 0 \ \text{ and } \Delta u_i \leq 0) \tag{21}$$

In order to provide the system with stochastic hill-climbing capabilities, the *Distributed Cauchy Machine* has been developed, where Cauchy color noise is added in the updating procedure. The output of unit

i at each time step *t* is stochastically updated with probability $P_i^C(t)$ which depends on the Cauchy distribution:

$$P_i^C(t) = \begin{cases} s_i(t) & \text{if } v_i(t) = 0 \\ 1 - s_i(t) & \text{if } v_i(t) = 1 \end{cases} \tag{22}$$

$$s_i(t) = P\{v_i(t) = 1\} = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{u_i(t)}{T_C(t)}\right) \tag{23}$$

The virtual temperature $T_C(t)$ is usually given by a *fast annealing* schedule:

$$T_C(t) = \frac{T_C^0}{1 + \beta t} \tag{24}$$

where T_C^0 is the initial temperature and β is real parameter in the range [0, 1] that controls the speed of the schedule. In the extreme case, where T_C = 0, the network becomes deterministic.

The Cauchy machine in general provides solutions of lower quality than the Boltzmann machine, however it has the advantage of being fully parallelizable. Also, since the activation is cumulative, changes at the state of a unit can be done only after many time steps, for the activation must change sign. The lack of this flexibility becomes more obvious when the system has operated for a long time and the activations have large absolute values.

5.6 Hybrid Scheme

The hybrid update scheme presented here is suggested in [PaLS97] and attempts to combine both the advantages of the Boltzmann and the Cauchy machine. It extends the synchronous discrete Hopfield model and the stochastic update rule is based on a convex combination of the Boltzmann and Cauchy machine update rules. At each time step *t*, the probability of accepting a state change concerning unit *i* is given by

$$P_i^H(t) = \alpha P_i^C(t) + (1 - \alpha) P_i^B(t) \tag{25}$$

where α is a control parameter in the range [0, 1]. A large value of α will result to a typical Cauchy machine, whereas a small value will result to a synchronous Boltzmann machine destroying the convergence property. The two temperatures is not necessary to be equal. A good choice would be to update $T_C(t)$ according to some annealing schedule and then take $T_B(t) = \lambda T_C(t)$, where λ is an adjustable parameter.

The above stochastic update rule accepts changes suggested by the energy difference (through $P_i^B(t)$) but such changes should be reflected on the activation $u_i(t)$, otherwise at the next time step the unit will return to the previous value, since it is still suggested by the activation. The following rule[2] ensures that this will not happen:

$$\text{if } (u_i(t + \Delta t) <> u_i(t)) \text{ and } \left(P_i^C(t) < 0.25\right) \text{ and } \left(P_i^B(t) > 0.75\right) \text{ then } (u_i(t + \Delta t) = -u_i(t)) \tag{26}$$

Experimental results has shown that the solutions produced by the hybrid scheme are similar to the solutions produced by the Boltzmann machine but the convergence time is larger, similar to that of the Cauchy machine. However, the hybrid scheme is fully parallelizable, thus the parallel implementation provides solutions similar to that of the Boltzmann machine but in substantially less time.

[2] The term $(P_i^B(t) > 0.75)$ in the rule is not included in the original paper. However, our experimental results suggest that it is necessary for the network to converge.

6 A Case Study: The Minimum Cost Spare Allocation Problem

6.1 The Problem

The Minimum Cost Spare Allocation Problem deals with the optimal repair (i.e. minimum cost repair) of faulty cells within a 2–dimensional array of cells. These cells can be processors, memory cells or VLSI components in practical applications. A number of spare rows and a number of spare columns is given with different costs associated to rows and columns. One spare row (column) can replace any row (column) in the array repairing all the faulty cells across this row (column). A feasible solution is obtained if we assign spare rows and columns on the array so that all the faulty cells are repaired. An optimal solution is a feasible solution with minimum cost.

The problem can be described more formally as follows: Given is a 2–dimensional array *M* with dimensions ($R \times C$), a set of positions in the array $FC=\{ (fr_k, fc_k) : 1 \le fr_k \le R, 1 \le fc_k \le C, k=1, 2, ..., NFC \}$ which represents faulty cells, a number of *SR* spare rows that can replace any row of the array with a cost *SRC* associated with each one of them and a number of *SC* spare columns that can replace any column of the array with a cost *SCC* associated with each one of them. The objective is to repair all the faulty cells, by assigning spare rows and columns, with the minimum cost. Let r_i, $i=1, 2, ..., R$, be a binary variable with a value of 1, if a spare row is assigned to row i and 0 otherwise. Similarly, let c_j, $j=1, 2, ..., C$, be a binary variable with a value of 1, if a spare column is assigned to column j and 0 otherwise. Then the problem can be described as an optimization problem. Minimize

$$f(\vec{r}, \vec{c}) = SRC \sum_{i=1}^{R} r_i + SCC \sum_{j=1}^{C} c_j \tag{27}$$

subject to the following constraints

$$\sum_{i=1}^{R} r_i \le SR \tag{28}$$

$$\sum_{j=1}^{C} c_j \le SC \tag{29}$$

$$\sum_{k=1}^{NFC} (1 - r_{fr_k})(1 - c_{fc_k}) = 0 \tag{30}$$

The problem already described is equivalent to the Vertex Cover problem in bipartite graphs, where each row is a node at the one side and each column is a node at the other side of the graph. The faulty cells are represented as links (not directed) between row-nodes and column-nodes. A cost is assigned to all the row-nodes and similarly to all the column-nodes. The problem now is to find a vertex cover with minimum cost, where the number of the row-nodes and the number of column-nodes in the cover are constrained by some maximum values.

The problem, in either of the two forms, is proved to be NP-HARD. Even if we drop the constraints (28) and (29) and assume infinite number of spare rows and columns, the problem is still intractable. Thus, it is difficult (or practically impossible) to find an optimal solution for large instances of the problem. Sometimes, a near-optimal solution is approvable. Here is where the neural network approach comes in. In the sequel, a neural network that solves the problem (actually the relaxed version) is presented.

6.2 The Energy Function

Recalling section 4 it is straightforward to construct the appropriate energy function (actually the problem formulated in such a way in order to make the mapping procedure easier). The energy function will be:

$$E(\vec{r},\vec{c}) = SRC\sum_{i=1}^{R} r_i + SCC\sum_{j=1}^{C} c_j + \alpha_1\Omega\left(\sum_{i=1}^{R} r_i - SR\right) + \alpha_2\Omega\left(\sum_{j=1}^{C} c_j - SC\right) + \alpha_3\left(\sum_{k=1}^{NFC}(1-r_{fr_k})(1-c_{fc_k})\right)^2 \quad (31)$$

At this point, we will drop the constraints on the number of spares (since it is difficult to handle them and it goes beyond our knowledge at present) and we will continue with the relaxed version of the problem. Also, note that the square at the last term of the function is not necessary since the quantity under it is always positive. Thus, the new energy function will be

$$E(\vec{r},\vec{c}) = SRC\sum_{i=1}^{R} r_i + SCC\sum_{j=1}^{C} c_j + \alpha_1\sum_{k=1}^{NFC}(1-r_{fr_k})(1-c_{fc_k}) \quad (32)$$

It is clear that the minimum of this function corresponds to the optimal solution. The last terms will be zero and the first two will give the optimal cost. The parameter α_1 gives the relative weight between the objectives of repairing all the cells or achieving the minimum cost. An appropriate value that satisfies both constraints can be found by experimentation.

6.3 The Network

The network will contain (*R*+*C*) units in total, one for each row (row-units) and one for each column (column-units). If a row-unit is firing then a spare row should be assigned to this row of the array. Similarly, firing column-units correspond to spare column assignments.

In order to derive weights and thresholds we must calculate the energy difference caused by a state change of a unit. Since there are two types of units, we handle the two cases separately. We have

$$\Delta E_{r_i} = \left(SRC + \alpha_1\sum_{k=1}^{NFC}(-1)\delta(i,fr_k)(1-r_{fc_k})\right)\Delta r_i \quad (33)$$

$$\text{or } \Delta E_{r_i} = -\left(-SRC + \alpha_1\sum_{k=1}^{NFC}\delta(i,fr_k) - \alpha_1\sum_{k=1}^{NFC}\delta(i,fr_k)r_{fc_k}\right)\Delta r_i \quad (34)$$

$$\text{or } \Delta E_{r_i} = -\left(\sum_{k=1}^{NFC}(-\alpha_1)\delta(i,fc_k)r_{fc_k} - SRC + \alpha_1\sum_{k=1}^{NFC}\delta(i,fc_k)\right)\Delta r_i \quad (35)$$

where $\delta(x,y) = \begin{cases} 1 & x = y \\ 0 & \text{o/w} \end{cases}$

Comparing the equation above with (8) it is easy to derive the following for each row-unit *i*, *i*=1,2, ..., *R* :

- Threshold $\theta_i = -SRC + \alpha_1(\#$ of faulty cells in row i)
- Weights $w_{ij} = \begin{cases} -\alpha_1 & \text{if } (r_i, r_j) \text{ is a faulty cell} \\ 0 & \text{otherwise} \end{cases}$

Similarly for the column-units we can derive

$$\Delta E_{c_j} = -\left(\sum_{k=1}^{NFC} (-\alpha_1)\delta(fr_k, j) r_{fr_k} - SCC + \alpha_1 \sum_{k=1}^{NFC} \delta(fr_k, j) \right) \Delta rc_j \tag{36}$$

and for each column-unit *j*, j=1, 2, ..., *C* we have

- Threshold $\theta_j = -SCC + \alpha_1(\#$ of faulty cells in column j$)$
- Weights $w_{ji} = \begin{cases} -\alpha_1 & \text{if } (r_i, r_j) \text{ is a faulty cell} \\ 0 & \text{otherwise} \end{cases}$

Note that there are no connections within the row-units or within the column-units but only from row-units to column-units and vice-versa.

It is rather easy to interpret to values above. The negative weight has the meaning that if a unit is firing then it covers all the faulty cells in its row or column, so it will try to prevent the units that could cover the same cells from firing. The positive term in the threshold will give firing priority to units with many faulty cells in their row or column, whereas the negative term will prevent units with high cost.

6.4 Implementation

A simulation of the network already described has been implemented using the C Programming Language. The code consists of three major modules:

1. *Problem Representation*: This module[3] is responsible for the input and output and the representation of the problem, or better, of the particular instance.
2. *Problem Mapping*: This module is responsible for converting the problem and mapping it on a neural network according to 6.3.
3. *Neural Network*: This module implements a general Hopfield neural network model, which was used to implement all the variations described in section 5.

A limitation of the current implementation comes from the fact that the code is not optimized, so that duplicate computations are eliminated. This is due to the generality of the basic implemented network in order to support the different variations. Perhaps, individual implementations for each variation will result to more optimal code.

Another improvement may be yield by adapting the network to the problem. In general, the number of faulty cells is much more smaller than the total number of cells. That leads into a weight matrix with most of the elements equal to zero. Maybe a linked list representation of the weight matrix will reduce the memory requirements of the network ($O\left((R+C)^2\right)$). In this case, the data structure used to represent the problem can be extended to store also the network.

Parallel versions for the parallelizable models have not been implemented yet, but it is hoped to be part of future work.

---

[3] This module of code was actually borrowed from my project for CMPS 530, where a parallel branch and bound algorithm for the same problem was developed.

6.5 Experimentation

Ten sample instances were used to demonstrate the approach. The number of parameters involved in the problem is large (5), leading in a plethora of different instances. The samples presented here are of relatively small "size", because large "size" instances require time and patience (both not available at this time). The results were compared with an optimal A* search algorithm for the same problem. However, the superiority of the approach (if any) against the optimal algorithm, would reveal only in large instances using parallel updates. All the experiments were conducted on SUN SPARCstation 4.

As far as concerning the parameters of the network, the following decisions were taken.

Analog Hopfield with Simulated Annealing The initial temperature was taken equal to 10 and a simple annealing schedule with decreasing rate equal to 0.996 was used, $T(s+1) = rate \times T(s)$.

Boltzmann Machine The initial temperature was taken equal to 5 and the annealing followed the logarithmic schedule (16) with rate equal to 0.000001.

Cauchy Machine The initial temperature was taken equal to 2, following a fast annealing schedule (24) with rate equal to 1. The time step *dt* was taken relatively small for better approximation (0.005).

Hybrid Scheme The value of α which determines the relative contribution gave best results, when it was taken equal to 0.75[4], i.e. when the contribution was 75% Cauchy Machine and 25% Boltzmann machine. The temperatures were the same as above and the parameter λ was taken equal to 2.5.

For all the four models, the termination condition was either 3 consecutive iterations with the same stable state, or a maximum number of 2,000 iterations. Each iteration corresponds to a full course of updates (all the units have been updated).

A crucial decision is the value of α_1 (see eq. 32–36), that determines the relative strength of the constraints. A small value will lead to an "optimal" cost, but probably with some cells uncovered (!), whereas a large value will cover all the cells but not necessarily with the optimal cost. After many experiments, a good value for α_1 was found to be the following:

$$\begin{aligned} \alpha_1 = 0.2 \times MinCost + 0.8 \times MaxCost + \delta(MinCost, MaxCost) \times MinCost \\ \text{where } MinCost = \min(SpareRowCost, SpareColumnCost) \\ \text{and } MaxCost = \max(SpareRowCost, SpareColumnCost) \end{aligned} \tag{37}$$

The following table summarizes the results.

[4] It seems to be a contradiction here, since in the original paper [SaLS97] its best value was determined as 0.25. However, it may be problem dependent (they study the Minimum Independent Set problem).

Instance	Faulty Cells	Rows	Columns	Row Cost	Column Cost	α_1	Analog Hopfield	Boltzmann Machine	Cauchy Machine	Hybrid Scheme	Exact Algorithm
1	5	5	5	1	9	6.0	9 1.15 sec 17,700	9 0.32 sec 2,190	9 0.11 sec 3,010	9 1.83 sec 4,120	9 0.35 sec
2	12	10	10	8	15	13.6	32 2.67 sec 30,500	32 0.95 sec 1,760	32 1.42 sec 4,200	32 0.66 sec 3,880	32 0.28 sec
3	50	20	20	3	3	6.0	48 4.01 sec 70,800	48 1.76 sec 3,040	48 2.01 sec 20,840	48 2.79 sec 22,960	48 6.00 sec
4	100	20	20	3	3	6.0	60 5.22 sec 70,800	60 0.27 sec 3,120	72 2.86 sec 25,240	60 3.04 sec 24,800	60 5.48 sec
5	100	10	30	7	2	6.0	70 6.46 sec 74,840	70 1.31 sec 3,440	70 1.69 sec 24,640	70 2.11 sec 36,440	58 0.61 sec
6	30	100	100	3	15	12.6	78 92.81 sec 354,000	78 1.23 sec 3,800	78 32.26 sec 159,000	78 44.73 sec 208,200	78 0.07 sec
7	60	100	100	3	3	6.0	117 97.28 sec 354,000	120 1.70 sec 4,000	117 44.13 sec 218,400	117 62.79 sec 304,200	117 1200.00 sec
8	60	100	100	1	9	7.4	45 94.59 sec 400,000	45 1.27 sec 4,600	45 50.23 sec 248,800	45 84.83 sec 400,000	45 0.29 sec
9	1000	250	250	10	10	20.0	2520 428.48 sec 735,000	2830 2.27 sec 4,000	2720 219.51 sec 431,500	2490 171.16 sec 322,500	out of memory
10	1000	500	500	10	10	20.0	3960 1951.01 sec 1,470,000	4410 1.64 sec 7,000	3970 1484.94 sec 1,327,000	3920 1617.16 sec 1,029,000	out of memory

Table 1 Experimental Results. For each model, the first number is the cost found, the second is the execution time and the third the number of cycles (individual updates). For the exact algorithm only the optimal cost and the execution time are provided.

Observing the table above, one can figure out easily when the neural network approach is most appropriate. The exact algorithm guarantees the optimal solution, but it is out of question for large instances. Its time and space requirements grow like $O\left(2^{TFC}\right)$, where *TFC* is the total number of faulty cells and is highly dependent on the values of the two costs. The size of the array has little impact on it.

On the other hand, the neural network cannot guarantee the optimal solution, but only a "good" solution, without any indication of the error. However, he time and space requirements are completely different and depend solely on the size of the array (see 6.4). The number of faulty cells, as well as the two costs, seem to have no impact on its performance. For these reasons, large instances of the problem can be handled, where the exact algorithm fails.

7 Conclusion

In this report we studied the application of neural networks on optimization problems. Neural Optimization seems to be a very promising area and, if nothing else, it provides some alternative solutions for problems which are considered to be difficult.

Although the neural network model was not inspired by optimization, the results presented here and elsewhere prove its success in other areas, that the ones it was intended for. This is another example of how advances in research and science can be realized by interdisciplinary research and study. May

be there are many other things that we ignore, because of the degree of specificity present in our days, instead of a general consideration under a global framework.

8 References

[ChUn93] Cichocki, A. & Unbehauen, R. *Neural Networks for Optimization and Signal Processing*, John Wiley & Sons, 1993.

[HeKP91] Hertz, J.A, Palmer, R.G. & Krogh, A.S. *Introduction to the Theory of Neural Computation*, Addison-Wesley Pub, 1991, ch. 1–4.

[Hopf82] Hopfield, J. "Neural Networks and Physical Systems with Emergent Collective Computational Capabilities", in *Proceedings of the National Academy of Science, USA*, **79**, 1982, pp. 2254–2558.

[HoTa85] Hopfield, J.J & Tank, D.W. "Neural Computation of Decisions in Optimization Problems", in *Biological Cybernetics*, **52**, 1985, pp. 141–152.

[HoTa87] Hopfield, J.J & Tank, D.W. "Computing with Neural Circuits: A Model", in *Science*, **233**, 1987, pp. 625–633.

[LLHZ93] Lillo, W., Loh, M., Hui, S., Zak, S. "On Solving Constrained Optimization Problems with Neural Networks: A Penalty Method Approach", in *IEEE Transactions on Neural Networks*, **4**, 6, 1993, pp. 931–940.

[McPi43] McCulloch, W.S. & Pitts, W. "A Logical Calculus of Ideas Immanent in Neurous Activity", in *Bulletin of Mathematical Biophysics* **5**, 1943, pp. 115–133.

[OhPS93] Ohlsson, M., Peterson, C. & Soderberg, B. "Neural Networks for optimization problems with inequality constraints: the knapsack problem", in *Neural Computation*, **5**, 2, 1993, pp. 331–339.

[PaLS97] Papageorgiou, G. Likas, A. & Stafylopatis, A. "A Hybrid Neural Optimization Scheme Based on Parallel Updates", submitted for publication, 1997.

[RaRa95] Rao, V.B & Rao, H.V. *C++ Neural Networks and Fuzzy Logic*, 2nd Edition, MIS Press, 1995, ch. 15.

[WaTa96] Wang, J. & Takefuji, Y. *Neural Computing for Optimization and Combinatorics*, World Scientific Pub Co, 1996.