GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement*

Andrew Davenport, Edward Tsang, Chang J. Wang and Kangmin Zhu

Department of Computer Science, University of Essex,
Wivenhoe Park, Colchester,
Essex CO4 3SQ, United Kingdom.
{daveat,edward,cwang,kangmin}@essex.ac.uk

Abstract

New approaches to solving constraint satisfaction problems using iterative improvement techniques have been found to be successful on certain, very large problems such as the million queens. However, on highly constrained problems it is possible for these methods to get caught in local minima. In this paper we present GENET, a connectionist architecture for solving binary and general constraint satisfaction problems by iterative improvement. GENET incorporates a learning strategy to escape from local minima. Although GENET has been designed to be implemented on VLSI hardware, we present empirical evidence to show that even when simulated on a single processor GENET can outperform existing iterative improvement techniques on hard instances of certain constraint satisfaction problems.

Introduction

Recently, new approaches to solving constraint satisfaction problems (CSPs) have been developed based upon iterative improvement (Minton *et al.* 1992; Selman & Kautz 1993; Sosic & Gu 1991). This technique involves first generating an initial, possibly "flawed" assignment of values to variables, then hill-climbing in the space of possible modifications to these assignments to minimize the number of constraint violations. Iterative improvement techniques have been found to be very successful on certain kinds of problems, for instance the min-conflicts hill-climbing (Minton *et al.* 1992) search can solve the million queens problem in seconds, while GSAT can solve hard, propositional satisfiability problems much larger than those which can be solved by more conventional search methods.

These methods do have a number of drawbacks. Firstly, many of them are not complete. However, the size of problems we are able to solve using iterative improvement techniques can so large that to do a complete search would, in many cases, not be possible anyway. A more serious drawback to iterative improvement techniques is that they can easily get caught in local minima. This is most likely to occur when trying to solve highly constrained problems where the number of solutions is relatively small.

In this paper we present GENET, a neural-network architecture for solving finite constraint satisfaction problems. GENET solves CSPs by iterative improvement and incorporates a learning strategy to escape local minima. The design of GENET was inspired by the heuristic repair method (Minton *et al.* 1992), which was itself based on a connectionist architecture for solving CSPs—the Guarded Discrete Stochastic (GDS) network (Adorf & Johnston 1990). Since GENET is a connectionist architecture it is capable of being fully parallelized. Indeed, GENET has been designed specifically for a VLSI implementation.

After introducing some terminology we describe a GENET model which has been shown to be effective for solving binary CSPs (Wang & Tsang 1991). We introduce extensions to this GENET model to enable it to solve problems with general constraints. We present experimental results comparing GENET with existing iterative improvement techniques on hard graph coloring problems, on randomly generated general CSPs and on the *Car Sequencing Problem* (Dincbas, Simonis, & Van Hentenryck 1988). Finally, we briefly explain what we expect to gain by using VLSI technology.

Terminology

We define a constraint satisfaction problem as a triple (Z, D, C) (Tsang 1993), where:

- Z is a finite set of *variables*,
- D is a function which maps every variable in Z to a set of objects of arbitrary type. We denote by D_x the set of objects mapped by D from x, where $x \in Z$. We call the set D_x the domain of x and the members of D_x possible *values* of x.
- C is a set of *constraints*. Each constraint in C restricts the values that can be assigned to the variables in Z simultaneously. A constraint is a *nogood*

*Andrew Davenport is supported by a Science and Engineering Research Council Ph.D Studentship. This research has also been supported by a grant (GR/H75275) from the Science and Engineering Research Council.

if it forbids certain values being assigned to variables simultaneously.

An *n-ary* constraint applies to n variables. A *binary* CSP is one with unary and binary constraints only. A *general* CSP may have constraints on any number of variables.

We define a *label*, denoted by $\langle x, v\rangle$, as a variable-value pair which represents the assignment of value v to variable x. A *compound label* is the simultaneous assignment of values to variables. We use $(\langle x_1, v_1\rangle, \ldots, \langle x_n, v_n\rangle)$ to denote the compound label of assigning $v_1, \ldots, v_n$ to $x_1, \ldots, x_n$ respectively. A *k-compound label* assigns k values to k variables simultaneously. A *solution tuple* of a CSP is a compound label for all the variables in the CSP which satisfies all the constraints.

Binary GENET

Network Architecture

The GENET neural network architecture is similiar to that of the GDS network. In the GENET network each variable i in Z is represented by a *cluster* of *label nodes*, one for each value j in its domain. Each label node may be in one of two states "on" or "off". The state $S_{\langle i,j\rangle}$ of a label node representing the label $\langle i, j\rangle$ indicates whether the assignment of the value j to variable i is true in the current network state. The output of a label node $V_{\langle i,j\rangle}$ is 1 if $S_{\langle i,j\rangle}$ is "on" and 0 otherwise.

All binary constraints in GENET must be represented by *nogood* ground terms. Binary constraints are implemented as inhibitory (negatively weighted) connections between label nodes which may be modified as a result of learning. Initially all weights are set to 1.

The input to each label node $I_{\langle i,j\rangle}$ is the weighted sum of the output of all the connected label nodes:

$$I_{\langle i,j\rangle} = \sum_{k \in Z, l \in D_k} W_{\langle i,j\rangle\langle k,l\rangle} V_{\langle k,l\rangle} \qquad (1)$$

where $W_{\langle i,j\rangle\langle k,l\rangle}$ is the connection weight between the label nodes representing the labels $\langle i, j\rangle$ and $\langle k, l\rangle$[1].

Since there are only connections between incompatible label nodes the input to a label node gives an indication of how much constraint violation would occur should the label node be in an on state. If no violation would occur the input would be a maximum of zero. A CSP is solved when the input to all the on label nodes is zero—such a state is called a *global minima*.

Each cluster of label nodes is governed by a *modulator* which effectively implements a variation of the *min-conflicts* heuristic (Minton *et al.* 1992). The purpose of the modulator is to determine which label node in the cluster is to be on. Only one label node in a cluster may be on at any one time. The modulator selects the label node with the highest input to be on, with ties being broken randomly. When the modulator changes the label node which is on in a cluster we say it has made a *repair*.

GENET Convergence Procedure

A *state* of a GENET network represents a complete assignment of values to variables i.e. exactly one label node in each cluster is on. The initial state of the GENET network is determined randomly—one label node per cluster is selected arbitrarily to be on. GENET iterates over *convergence cycles* until it finds a global minima. We define a convergence cycle as:

1. *foreach cluster in parallel do*[2] *update states of all label nodes in cluster,*
2. *if none of the label nodes have changed state in step 1 then*
   (a) *if the input to all on nodes is zero then solution found—terminate,*
   (b) *else activate learning,*
3. *goto step 1.*

Learning

Like most hill-climbing searches, GENET can reach local optimal points in the search space where no more improvements can be made to the current state—in this case we say the network is in a *minima*. A *local minima* is a minima in which constraints are violated. GENET can sometimes escape such minima by making sideways moves to other states of the same "cost". However in some minima this is not possible, in which case we say the network is in a *single-state* minima. To escape local minima we adjust the weights on the connections between label nodes which violate a constraint according to the following rule:[3]

$$W^{t+1}_{\langle i,j\rangle\langle k,l\rangle} = W^{t}_{\langle i,j\rangle\langle k,l\rangle} \quad V_{\langle i,j\rangle} V_{\langle k,l\rangle} \qquad (2)$$

where $W^t_{\langle i,j\rangle\langle k,l\rangle}$ is the connection weight between label nodes representing $\langle i, j\rangle$ and $\langle k, l\rangle$ at time t.

By using weights we associate with each constraint a cost of violating that constraint. We can also associate with each GENET network state a cost which is the sum of the magnitudes of the weights of all the constraints violated in that state.

Learning has the effect of "filling in" local minima by increasing the cost of violating the constraints which are violated in the minima. After learning, constraints which were violated in the minima are less likely to be violated again. This can be particularly useful in

[1] If there is no constraint between two label nodes representing $\langle i, j\rangle$ and $\langle k, l\rangle$ then $W_{\langle i,j\rangle\langle k,l\rangle} = 0$.

[2] We do not want clusters to update their states at exactly the same time since this may cause the network to oscillate between a small number of states indefinitely. In a VLSI implementation we would expect the clusters to update at slightly different times.

[3] Morris (Morris 1993) has recently reported a similiar mechanism for escaping minima.

structured CSPs where some constraints are more critical than others (Selman & Kautz 1993).

Learning is activated when the GENET network state remains unchanged after a convergence cycle. Thus learning may occur when GENET, given the choice of a number of possible sideways moves to states of the same cost, makes a sideways move back to its current state. We consider this a useful feature of GENET since it allows the network to escape more complicated *multi-state* minima composed of a "plateau" of states of the same cost.

A consequence of learning is that we can show GENET is not complete. This is because learning affects many other possible network states as well as those that compose the local minima. As a result of learning new local minima may be created. A discussion of the problems this may cause can be found in (Morris 1993).

General Constraints

Many real-life CSPs have general constraints e.g. scheduling, car sequencing (Dincbas, Simonis, & Van Hentenryck 1988). In this section we describe how can we represent two types of general constraint, the *illegal* constraint and the *atmost* constraint, in a GENET network. One of our motivations for devising these particular constraints has been the Car Sequencing Problem, a real-life general CSP once considered intractable (Parrello & Kabat 1986) and which has been successfully tackled using CSP solving techniques (Dincbas, Simonis, & Van Hentenryck 1988).

Since we cannot represent general constraints by binary connections alone, we introduce a new class of nodes called *constraint nodes*. A constraint node is connected to one or more label nodes.

Let c be a constraint node and L be the set of label nodes which are connected to c. Then the input I_c to the constraint node c is the *unweighted* sum of the outputs of these connected label nodes:

$$I_c = \sum_{\langle i,j\rangle \in L} V_{\langle i,j\rangle} \tag{3}$$

We can consider the connection weights between constraint nodes and label nodes to be assymetric. The weight on all connections from label nodes to constraint nodes is 1 and is not changed by learning. Connection weights from constraint nodes to label nodes are, like for binary constraints, initialised to 1 and can change as a result of learning. The input to label nodes in networks with general constraints C is now given by:

$$I_{\langle i,j\rangle} = \sum_{k\in Z, l\in D_k} W_{\langle i,j\rangle\langle k,l\rangle} V_{\langle k,l\rangle} + \sum_{c\in C} W_{c,\langle i,j\rangle} V_{c,\langle i,j\rangle} \tag{4}$$

where $V_{c,\langle i,j\rangle}$ is the output of the constraint node c to the label node $\langle i,j\rangle$.

The learning mechanism for connection weights $W^t_{c,\langle i,j\rangle}$ between constraint nodes c and label nodes $\langle i,j\rangle$ is given by:

$$W^{t+1}_{c,\langle i,j\rangle} = \begin{cases} W^t_{c,\langle i,j\rangle} & 1 \quad \text{if } S_c > 0 \\ W^t_{c,\langle i,j\rangle} & \text{otherwise} \end{cases} \tag{5}$$

where S_c is the state of the constraint node.

The Illegal Constraint

The $illegal(\langle x_1,v_1\rangle,\ldots,\langle x_k,v_k\rangle)$ constraint specifies that the k-compound label $L = (\langle x_1,v_1\rangle,\ldots,\langle x_k,v_k\rangle)$ is a nogood. An illegal constraint is represented in a GENET network by an illegal constraint node, which is connected to the k label nodes which represent the k labels in L.

$$S_{ill} = I_{ill} \quad (k \quad 1) \tag{6}$$

The state S_{ill} of the illegal constraint node is negative if less than k 1 of connected label nodes are on. In this case there is no possibility that the constraint will become violated should another node become on. A constraint node in this state outputs 0 to all the connected label nodes.

If k 1 of the connected label nodes are on then we want to discourage the remaining off label node from becoming on, since this will cause the constraint to be violated. However, we do not wish to penalize the label nodes which are already on, since the constraint remain satisfied even if they do change state. In this case we want to output 1 to the label node which is off and 0 to the remaining label nodes.

Finally, if all the connected label nodes are on then the constraint is violated. We want to penalize all these nodes for violating the constraint, so we give them all an output of 1 to encourage them to change state.

We summarize the output $V_{ill,\langle i,j\rangle}$ from an illegal constraint node *ill* to a label node representing the label $\langle i,j\rangle$ by:

$$V_{ill,\langle i,j\rangle} = \begin{cases} 0 & \text{if } S_{ill} < 0 \\ 1 + S_{ill} \quad V_{\langle i,j\rangle} & \text{otherwise} \end{cases} \tag{7}$$

The Atmost Constraint

We can easily extend the illegal constraint node architecture to represent more complex constraints. For instance, given a set of variables *Var* and values *Val* the *atmost(N, Var, Val)* constraint specifies that no more than N variables from *Var* may take values from *Val*. The atmost constraint node is connected to all nodes of the set L which represent the labels $\{\langle i,j\rangle | i \in Var,\ j \in Val,\ j \in D_i\}$. This constraint is a modification of the atmost constraint found in the CHIP constraint logic programming language.

The state S_{atm} of an atmost constraint node is determined as follows:

$$S_{atm} = I_{atm} \quad N \tag{8}$$

The output from an atmost constraint node is similar to that for the illegal constraint node, although we have the added complication that a single variable may have more than one value in the constraint. We do not want label nodes in the same cluster to receive different inputs from a particular constraint node since, in situations where the network would normally be in a single state local minima, we would find the network oscillating about the states of these label nodes. Instead, we give the output of an atmost constraint node *atm* to a label node representing the label $\langle i, j \rangle$ as follows:

$$V_{atm,\langle i,j \rangle} = \begin{cases} 0 & \text{if } S_{atm} < 0 \\ 1 \quad \text{Max}\{V_{\langle i,k \rangle} | k \in Val\} & \text{if } S_{atm} = 0 \\ 1 & \text{otherwise} \end{cases} \quad (9)$$

Experimental Results

Graph Coloring

In (Selman & Kautz 1993) it is reported that the performance of GSAT on graph coloring problems is comparable with the performance of some of the best specialised graph-coloring algorithms. This surprised us since a graph coloring problem with N vertices to be colored with k colors would require, in a conjunctive normal form (CNF) representation, $N \times k$ variables. Since each of these variables has a domain size of 2 the size of the search space is 2^{Nk}. To represent such a problem as a CSP would require N variables of domain size k, giving a search space of size k^N. For example, the 250 variable 29 coloring problem in Table 1 has a search space size in GENET of 4×10^{365} possible states. This is far smaller than the corresponding size of 3×10^{2183} states possible in GSAT.

Another difference between GSAT and GENET is the way in which they make repairs. GSAT picks the best "global" repair which reduces the number of conflicts amongst all the variables, whereas GENET makes "local" repairs which minimizes the number of conflicts for each variable. Thus we would expect repairs made by GSAT to be of "higher quality" than those of GENET, although they are made at the extra expense of considering more possibilities for each repair.

We compared GSAT[4] and GENET[5] on a set of hard graph coloring problems described in (Johnson *et al.* 1991), running each method ten times on each problem. We present the results of our experiments in Tables 1 and 2. Both GSAT and GENET managed to solve all the problems, although GSAT makes many more repairs to solve each problem. These results seem to confirm our conjecture that for CSPs such as graph coloring GENET is more effective than GSAT because of the way it represents such problems.

[4]We ran GSAT with MAX-FLIPS set to $10\times$ the number of variables, and with averaging in reset after every 25 tries.

[5]All experiments were carried out using a GENET simulator written in C++ on a Sun Microsystems Sparc Classic.

graph nodes	graph colors	median time	median number of repairs
125	17	8.0 hours	65,197,415
125	18	30 secs	65,969
250	15	5.0 secs	2,839
250	29	1.8 hours	7,429,308

Table 1: GSAT on hard graph coloring problems.

graph nodes	graph colors	median time	median number of repairs
125	17	2.6 hours	1,626,861
125	18	23 secs	7,011
250	15	4.2 secs	580
250	29	1.1 hours	571,748

Table 2: GENET on hard graph coloring problems.

Random General Constraint Satisfaction Problems

There are two important differences between a sequential implementation of GENET and min-conflicts hill-climbing (MCHC). The first is our learning strategy for escaping local minima. The second difference is in choosing which variables to update. MCHC selects randomly a variable to update from the set of variables which are currently in conflict with other variables. In GENET we randomly select variables to update from the set of all variables, regardless of whether they conflict with any other variables.

Our aim in this experiment was to try to determine empirically what effect these individual modifications to MCHC was making to the effectiveness of its search.

We compared GENET with a basic min-conflicts hill-climbing search, a modified MCHC (MCHC2) and a modified version of GENET (GENET2). MCHC2 randomly selects variables to update from the set of all variables, not just those which are in conflict. MCHC2 can also be regarded as a sequential version of GENET without learning. In GENET2 variables are only updated if they are in conflict with other variables.

We produced a set of general CSPs with varying numbers of the *atmost(N,Var,Val)* constraint, where $N = 3$, $|Var| = 5$ and $|Val| = 5$. The problems were not guaranteed to be solvable. Each problem had fifty variables and a domain of ten values. The set of variables and values in each constraint were generated randomly. At each data-point we generated ten problems. We ran each problem ten times with GENET, GENET2, MCHC and MCHC2. We set a limit of five hundred thousand repairs for each run, after which failure was reported if no solution had been found.

Figure 1 shows that MCHC2 solves more problems than MCHC. This is to be expected since, because MCHC2 can modify the values of variables which are

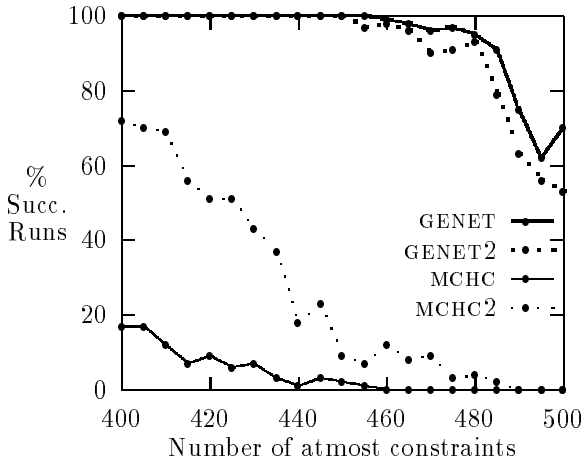

Figure 1: Comparison of percentage of successful runs for GENET and min-conflicts hill-climbing searches on randomly generated general constraint satisfaction problems.

not in conflict, it is less likely to become trapped in local minima. The performance of GENET2 shows that learning is an even more effective way of escaping local minima. However Figure 1 shows that combining these two approaches in GENET is the most effective way of escaping minima for this particular problem set.

The Car Sequencing Problem

We have been using the car-sequencing problem as a benchmark problem during the development of a GENET model which would solve general CSPs. The car-sequencing problem is a real-life general CSP which is considered particularly difficult due to the presence of global atmost constraints. For a full description of the car sequencing problem see (Dincbas, Simonis, & Van Hentenryck 1988).

We compared GENET with MCHC, MCHC2 and CHIP. CHIP is a constraint logic programming language which uses a complete search based on forward-checking and the fail-first principle to solve CSPs. In our experiments we used randomly generated problems of size 200 cars and utilisation percentages in the range 60% to 80%. At each utilisation percentage we generated ten problems. The problems all had 200 variables with domains varying from 17 to 28 values and approximately 1000 atmost constraints of varying arity. All the problems were guaranteed to be solvable. We ran the GENET, MCHC and MCHC2 ten times on each problem, with a limit of one million repairs for each run, after which failure was reported. This limit corresponded to a running time of approximately 220 seconds at 60% utilisation up to 270 seconds at 80% utilisation. We used the method described in (Dincbas, Simonis, & Van Hentenryck 1988) to program the problems in CHIP, which included adding redundant constraints to speed up the search. With a time limit of one hour to solve each problem CHIP managed to solve two problems at 60% utilisation, one problem at 65% utilisation, two problems at 70% utilisation and one problem at 75% utilisation. The results for min-conflicts hill-climbing and GENET on 200 car sequencing problems are given in Tables 3 and 4.

utilisation %	MCHC % succ. runs	MCHC median repairs	MCHC2 % succ. runs	MCHC2 median repairs
60	78	737	85	549
65	81	586	82	524
70	82	670	85	508
75	76	1282	82	811
80	29	10235	51	4449

Table 3: A comparison of MCHC and MCHC2 on 200 car sequencing problems.

utilisation %	GENET % succ. runs	GENET median repairs	GENET3 % succ. runs	GENET3 median repairs
60	84	463	100	452
65	87	426	100	439
70	83	456	100	426
75	85	730	100	686
80	50	4529	100	1886

Table 4: A comparison of GENET and GENET3 on 200 car sequencing problems.

From Table 3 it can be seen that MCHC2 is more effective than MCHC at solving the car-sequencing problem. However the results for MCHC2 and GENET are very similiar, indicating that learning is having very little or no effect in GENET. This can be attributed to the presence of very large plateaus of states of the same cost in the search space. Learning is activated only when GENET stays in the same state for more than one cycle, thus learning is less likely to occur when these plateaus are large. To remedy this problem we made a modification to GENET to force learning to occur more often. We define the parameter p_{sw} as the probability that, in a given convergence cycle, GENET may make sideways moves. Thus, for each convergence cycle, GENET may make sideways moves with probability p_{sw}, and may only make moves which decrease the cost with probability $1 \quad p_{sw}$. Thus, if GENET is in a state where only sideways moves may be made then learning will occur with a probability of at least $1 \quad p_{sw}$. The results for GENET3 in Table 4, where p_{sw} is set to 0.75, shows that this modification significantly improves the performance of GENET.

VLSI Implementation

Although the results presented so far have been obtained using a GENET simulator on a single processor machine, it is the aim of our project to implement GENET on VLSI chips. A full discussion of a VLSI implementation for GENET would be beyond the scope of this paper[6] so in this section we describe what we expect to gain by using VLSI technology.

A disadvantage of the min-conflicts heuristic, as noted by Minton (Minton *et al.* 1992), is that the time taken to accomplish a repair grows with the size of the problem. For a single-processor implementation of GENET the cost of determining for a single variable the best value to take is proportional to the number of values in the domain of the variable and the number constraints involving that value. To determine for each variable the best value to take can potentially be performed at constant time in a VLSI implementation of GENET no matter how large the domain or how highly constrained the problem. This would mean that the time taken for GENET to perform a single convergence cycle would be constant, no matter what the problem characteristics[7]. Since we estimate the time taken to perform one convergence cycle using current VLSI technology to be of the order of tens of nanoseconds, this would allow all the CSPs mentioned in this paper to be solved in seconds rather than minutes or hours.

Conclusion

We have presented GENET, a connectionist architecture for solving constraint satisfaction problems by iterative improvement. GENET has been designed to be implemented on VLSI hardware. However we have presented empirical evidence to show that even when simulated on a single processor GENET can outperform existing iterative improvement techiques on hard binary and general CSPs,

We have developed strategies for escaping local minima which we believe significantly extend the scope of hill-climbing searches based on the min-conflicts heuristic. We have presented empirical evidence to show that GENET can effectively escape local minima when solving a range of highly constrained, real-life and randomly generated problems.

Acknowledgements

We would also like to thank Alvin Kwan for his useful comments on earlier drafts of this paper. We are grateful to Bart Selman and Henry Kautz for making their implementation of GSAT available to us.

[6] A VLSI design for GENET is described in (Wang & Tsang 1992)

[7] The size of problem would be limited by current VLSI technology

References

Adorf, H., and Johnston, M. 1990. A discrete stochastic neural network algorithm for constraint satisfaction problems. In *Proceedings of the International Joint Conference on Neural Networks.*

Dincbas, M.; Simonis, H.; and Van Hentenryck, P. 1988. Solving the car-sequencing problem in logic programming. In *Proceedings of ECAI-88.*

Johnson, D.; Aragon, C.; McGeoch, L.; and Schevon, C. 1991. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning. *Operations Research* 39(3):378–406.

Minton, S.; Johnston, M.; Philips, A.; and Laird, P. 1992. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58:161–205.

Morris, P. 1993. The breakout method for escaping from local minima. In *Proceedings of the Twelth National Conference on Artificial Intelligence.* AAAI Press/The MIT Press.

Parrello, B., and Kabat, W. C. 1986. Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. *JOURNAL of Automated Reasoning* 2:1–42.

Selman, B., and Kautz, H. 1993. Domain independent extensions to GSAT: Solving large structured satisfiability problems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence.*

Sosic, R., and Gu, J. 1991. 3,000,000 queens in less than one minute. *SIGART Bulletin* 2(2):22–24.

Tsang, E. 1993. *Foundations of Constraint Satisfaction.* Academic Press.

Wang, C., and Tsang, E. 1991. Solving constraint satisfaction problems using neural-networks. In *Proceedings IEE Second International Conference on Artificial Neural Networks.*

Wang, C., and Tsang, E. 1992. A cascadable VLSI design for GENET. In *International Workshop on VLSI for Neural Networks and Artificial Intelligence.*