Towards the Integration of Artificial Neural Networks and Constraint Logic Programming

J.H.M. Lee and V.W.L. Tam

Department of Computer Science
The Chinese University of Hong Kong
Hong Kong
{jlee,wltam}@cs.cuhk.hk

Abstract

In this paper, we present a general framework for integrating artificial neural networks (ANN) into constraint logic programming for solving CSP's. This framework is realized in a novel programming language PROCLANN, which uses the standard goal reduction strategy as frontend to generate constraints for an efficient backend ANN-based constraint-solver. PROCLANN retains the simple and elegant declarative semantics of constraint logic programming. Its operational semantics is probabilistic in nature but it possesses soundness and completeness results. An initial prototype of PROCLANN is constructed and provides empirical evidence that PROCLANN compares favourably against the state of art in CLP implementations on certain hard instances of CSP.

1 Motivation and related work

The problem at hand is that of *constraint satisfaction problems* (CSP) defined in the sense of Mackworth [13]. CSP is a general problem which occur in many areas of applications, such as computer vision, planning, resource allocation in scheduling and temporal reasoning, etc. There are two concerns when handling CSP's: *programming* and *efficient execution*. The former is well addressed by logic programming in Prolog. The relational nature of constraints and the declarativeness of CSP's make Prolog an ideal language to specify CSP. The basic backtracking tree-search execution strategy of Prolog, however, usually produces unacceptable performance on a computer even on medium size problems, as the tree contains too many possibilities to be exhaustively searched. Van Hentenryck [18] introduces consistency techniques [13] into logic programming so that constraints are used actively to prune search space *a priori*. The framework is realized in the CHIP language [5], which has been successfully applied to solving numerous industrial applications [3, 4]. CHIP's execution mechanism is still based on tree search and backtracking, which are main barriers to efficient solving of some hard or large-scale CSP's.

There has been recent interest in applying ANN to optimization and CSP. For examples, Hopfield network, the Boltzmann Machine and the elastic network have been used in the traveling salesman problems [9, 1, 6], the Tangram puzzles [12] and the N-queens problem [15] with satisfactory results. Wang and Tsang [19, 17] propose GENET, a generic ANN model, for solving general CSP's with binary constraints. A massively parallel implementation of GENET may attain a theoretical speedup in the order of 10^8 over sequential heuristic search. Wang and Tsang [20] also propose a cascadable VLSI design for GENET.

ANN, while efficient, can be difficult to program. Translating a CSP into a neural network is often a tedious and error-prone task. This is where logic programming can help. We propose to integrate constraint logic programming and ANN. A CSP is specified as a constraint logic program. The logic programming part of the execution mechanism generates the corresponding neural network of the CSP, which is then submitted to the backend ANN-based constraint-solver for further scrutiny. This way we obtain the best of both worlds: *an easy-to-program language with high efficiency.*

There has been other attempts [7, 14], with different motivations, in the amalgamation of logic programming and ANN. These approaches are all translational, in which a set of logical formula is translated into a neural network and theorem proving becomes an energy minimization process. We propose a radically different approach, which aims at tight coupling of logical deduction and neural network computation.

The paper is organized as follows. Section 2 states the criteria for the ANN-based constraint solver. The syntax and semantics of the PROCLANN language is presented in section 3. Soundness and probabilistic completeness results are shown. In section 4, we review the GENET model and show how GENET satisfies the criteria as a constraint solver. In section 5, we describe the PROCLANN prototype and its comparison to CHIP using the graph-coloring problem. We summarize our contributions, limitations of PROCLANN, and shed light on future work in section 6.

2 An ANN-based constraint-solver

We propose using an ANN model as a CLP language solver. The language framework that we shall present later is *not tied* to any particular ANN model but the chosen model has to satisfy the following criteria. **Network uniqueness:** every CSP (or constraint network) can be translated into a *unique* network topology in the model. **Soundness:** every answer generated by the neural network model must be a solution of the corresponding CSP. **Probabilistic Completeness:** if a CSP has solution θ, then there exists *non-zero* probability that the corresponding neural network has θ as answer. **Incrementality:** the model must be amenable to *efficient* incremental execution.

3 The PROCLANN language

We describe the PROCLANN (**PRO**gramming in **C**onstraint **L**ogic with **A**rtificial **N**eural **N**etwork), a committed-choice language. This design decision is justifiable by the fact that goal reduction is used only to inject constraints into a neural network but not to perform searching. The operational semantics of PROCLANN is similar to that of the CLP scheme [10] but the use of an ANN-based constraint-solver is made explicit in the *PROCLANN Computation Model*, which is both sound and *probabilistically complete*.

3.1 Syntax and declarative semantics

Since we are interested in CSP, we need the notion of domains and domain variables [18], abbreviated as *d-variables*. A *domain* is a finite set of constants. Each d-variable has the form X^d, where d is the associated domain of X^d. To differentiate d-variables from Herbrand variables, we call the latter *h-variables*.

The constraint domain $\mathcal{D}$ in consideration defines a set $\mathcal{C}$ of *primitive constraints*. PROCLANN only supports binary constraints. As specified in the CLP scheme [10], the equality constraint `#=`[1] is in $\mathcal{C}$. An *atom* is any standard atomic formula whose predicate symbol is different from those of the primitive constraints. A *clause* in PROCLANN has the form

$$H \text{ :- } \vec{G} \mid \vec{C}, B_1, \ldots, B_n. \quad (n \geq 0)$$

where

1. H (the *head*) is an atom, $\vec{G}$ (the *guard*) is a conjunction, possibly empty, of system predicates, $\vec{C}$ (the *constraints*) is a conjunction, possibly empty, of atomic constraints, and $B_1, \ldots, B_n$ (the *body*) is a conjunction of atoms;
2. $\vec{G}$ contains no d-variables;
3. $\vec{C}$ contains only d-variables and ground terms;

The "`:-`" symbol is read as *if*, and the commas and the "`|`" symbol are read as conjunction. If a clause has no guard, the "`|`" can be omitted. Thus a clause has the same declarative reading as a Horn clause with domain variables [18]. A *program* is a set of clauses. A *query* in PROCLANN is simply a clause without the head and the guard. The concepts of truth, model, and logical consequences of PROCLANN are equivalent to those of CHIP [18].

We conclude this subsection by a PROCLANN program shown in figure 1 for the graph-coloring problem of figure 2. The built-in predicate `in/2` declares that each element of `L` is a list of d-variables with domain $\{1, \ldots, n\}$. The "`#\=`" predicate stands for the disequality constraint. The `colorVertices/3` predicate specifies the constraints for domain variable `X` in `L`. And the `checkNbs/3` predicate states the disequation "`X #\= Nb`" if two vertices are adjacent to each other in the graph. The program is almost identical to the CHIP version, except that there is no need to perform explicit labeling operation using the CHIP `indomain/1` predicate.

3.2 Operational semantics

The introduction of d-variables prompts for modification of the unification algorithm. Similar to

[1] All PROCLANN constraints are preceded by the `#` symbol.

```
colorGraph(N,L,Map) :- L in 1..N,
                       colorVertices(Map).

colorVertices([]).
colorVertices([[X,Nbs]|Vts]) :-
    checkNbs(X,Nbs),
    colorVertices(Vts).

checkNbs(_,[]).
checkNbs(X,[Nb|Nbs]) :- X #\= Nb,
                        checkNbs(X,Nbs).

testGraph(N,L) :- map(L,Map),
                  colorGraph(N,L,Map).

map([X1,X2,X3,X4,X5,X6,X7,X8],[[X1,[X2,X6,
  X8]],[X2,[X1,X3,X4,X6]],[X3,[X2,X4]],
  [X4,[X2,X3,X5]],[X5,[X4,X6,X7]],[X6,
  [X1,X2,X5,X7,X8]],[X7,[X5,X6,x8]],[X8,
  [X1,X6,X7]]]).
```

Figure 1: A graph-coloring program in PROCLANN.

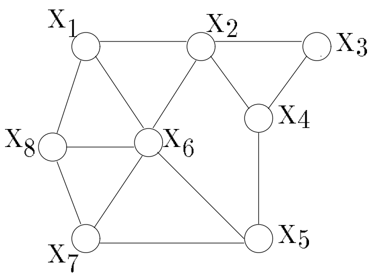


Figure 2: An example graph.

CHIP [18], we have three cases to consider in the new definition.

1. If a h-variable X is unified with a d-variable Y^e, unification succeeds and the binding $\{X/Y^e\}$ is generated.
2. If two d-variables X^d and Y^e are unified, unification succeeds and the constraint X^d #= Y^e is generated. No binding is generated.
3. If a d-variable X^d is unified with a constant c, unification succeeds and the constraint X^d #= $Y^{\{c\}}$ is generated. Again no binding is generated.

Therefore, the output of the unification is a set of equality constraints $\vec{C_u}$ and a set θ of bindings of the form X/t, in which X is a h-variable.

We define the *PROCLANN Computation Model* in the following. Let P be a PROCLANN program and Q_i a query. Q_{i+1} is *derived* from $Q_i = \leftarrow \vec{C}, B_1, \ldots, B_j, \ldots, B_q$, where $\vec{C}$ is a conjunction of primitive constraints, if the following conditions are satisfied:

1. B_j is the *selected* atom in Q_i;
2. there exists a clause "H :- $\vec{G}$ | $\vec{C'}, A_1, \ldots, A_n$." ($n \geq 0$) in P such that
   (a) H is unifiable with B_j generating substitution θ_i and constraint set $\vec{C_u}$, and
   (b) the execution of the guard $\vec{G}$ succeeds;
3. the corresponding neural network of $\vec{C''} = \vec{C} \cup \vec{C_u} \cup \vec{C'}$ finds a solution in finite amount of time;
4. $Q_{i+1} = \leftarrow \vec{C''}, B_1, \ldots, B_{j-1}, A_1, \ldots, A_n, B_{j+1}, \ldots, B_q$.

A *derivation* is a, possibly infinite, sequence of queries $Q = Q_0, Q_1, Q_2, \ldots$ such that Q_{i+1} is derived from Q_i generating substitution θ_{i+1}. A derivation is *successful* if it is finite with m steps and the last query contains no atoms. In this case, the constraint accumulated in the last goal is called the *answer constraint* and the associated neural network is called the *answer network*. A *computed answer* of a successful derivation is the union of a solution state θ_N of the answer network N and the composition of substitutions $\theta = \theta_1 \cdots \theta_m$ restricted to the h-variables in Q. There is no notion of failed derivation in PROCLANN.

3.3 Soundness and completeness

The PROCLANN computation model is very similar to the operational semantics of the CLP scheme, except that the use of neural network model is made explicit. It follows that the PROCLANN computation model is *sound*.

Theorem 3.1: (Soundness of the PROCLANN Computation Model) Let P be a PROCLANN program. If a query $\leftarrow Q$ has a successful derivation with computed answer θ, then

$$P \models \forall(Q\theta),$$

where $\forall(F)$ is the universal closure of the formula F.

Proof: From soundness of constraint solver (section 2). ∎

The completeness result requires the notion of *probabilistic completeness*.

Theorem 3.2: (Probabilistic Completeness of the PROCLANN Computation Model) Let P be a PROCLANN program, $\leftarrow Q$ a query, $\theta = \theta_d \cup \theta_h$ a substitution, where θ_d contains only bindings for d-variables and θ_h contains only those for h-variables. If $P \models \forall(Q\theta)$, then there exist a successful derivation for $\leftarrow Q$ with answer network N such that

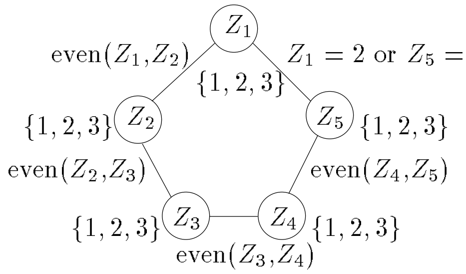
- there is a *non-zero* probability that N will find a solution, represented as substitution θ_d, and
- if θ'_h is the computed answer of the successful derivation restricted to h-variables, then there exist substitution σ such that $\theta_h = \theta'_h\sigma$.

Proof: From probabilistic completeness of constraint solver (section 2). ∎

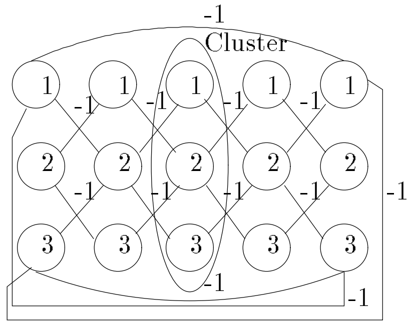
Theorem 3.2 reviews the probabilistic nature of PROCLANN to the fullest extent. It means that if the CSP specified by the program is solvable, then there always exists a successful derivation and the answer network obtained has non-zero probability to converge to any solution of the CSP.

4 GENET

We have chosen the GENET model [19, 17] to demonstrate the feasibility of our proposal. GENET is a generic neural network simulator that can be used to solve general CSP's with finite domains. To illustrate how a GENET network is constructed for a CSP, let us take a simple but tight[2] binary CSP as example. Assume there are five finite domain variables Z_1 to Z_5, all with domain $d = \{1, 2, 3\}$. Let V_i denotes the value assigned to Z_i. There are two constraints: (1) $V_i + V_{i+1}$ is even and (2) either $V_1 = 2$ or $V_5 = 2$. Figure 3 (a) is the constraint network of the example defined in the sense of Mackworth [13]. Figure 3 (b) is the corresponding network topology in GENET. Each column of 3 nodes represents a cluster for each domain variable. Each connection weight W_{ij} between nodes i and j is initially set to -1. The cross inhibitory connections are constructed according to constraint 1. For example, when $V_1 = 2$ and $V_2 = 1$, their sum is 3 (odd) which violates constraint 1. So, the second node in first column and first node in second column



(a) Constraint network of a tight CSP



(b) The GENET topology of the CSP

Figure 3: A tight CSP.

is connected. For constraint 2, consider when $V_1 = 1$, V_5 should then be 2. In order to fulfill this requirement, node 1 of first cluster should send inhibitory effect to the first and third node of Z_5. The other two connections are constructed similarly.

4.1 Dynamics of GENET

The state S_i of node i is either 1 for *on* or 0 for *off*. The input I_i to each node i is a weighted sum of all its connected nodes' states, i.e. $I_i = \sum_{j=1, j\neq i}^{k} W_{ij} \times S_j$. At any moment, only one node in a cluster is *on*. We say that the variable as represented by the cluster is *assigned* the value as represented by the label of the on-node. For the purpose of our discussion, we define a *network state* to be an assignment of values to each of the clusters (variables). A *solution state* is a *consistent* assignment of values to the variables so that none of the constraints are violated. Dynamics of GENET concerns how the network changes states and connection weights before it reaches a solution state. It can be realized in its network convergence procedure as follows. (1) Initially one node in each cluster is randomly selected to be on. (2) Every node then calculates its input. (3) In each cluster, node with the maximum input will be turned on. To avoid chaotic or cyclic wandering of network states, if there are several

[2] Tightness is defined here as the number of solutions over the size of search space. A tight CSP is one which has few solutions over a large search space.

nodes with maximum inputs, node that is already on will remain on. Otherwise, GENET will just randomly select one out of them. This step is called the *state update rule*. (4) When the network settles in a stable state (i.e. with all on-nodes unchanged), GENET checks to see if all the on-nodes have zero input. If so, the state represent a *solution* to the CSP and the simulator *stops* with *success*. Otherwise, the state represents a *local maximum*. (5) If the network settles in a local maximum of cluster input(s), a *heuristic learning rule* will update the all connection weights by $W_{ij} := W_{ij} - S_i \times S_j$. (6) Goto step 2.

4.2 GENET as a constraint solver

We can easily verify the network uniqueness property of GENET. In the following, we briefly review the *soundness*, *probabilistic completeness* and *incrementality* properties of GENET.

The success criterion of GENET guarantees soundness, as demonstrated in the following theorem.

Theorem 4.1: If the GENET network of a CSP P settles in a solution state S, then the value assignment θ corresponding to S is a solution of P.

Proof: A GENET network is in a solution state when all on-nodes have zero input, i.e. *no two* on-nodes are connected. Thus the value assignment to each cluster (variable) causes no violation of constraints in the network, and represent a solution of the CSP. ∎

We can understand the network convergence procedure from an energy maximization perspective. Let $N(S)$ be the set of on-nodes of state S. We define the *energy* of state S by

$$E(S) = \sum_{i \in N(S)} I_i = \sum_{i \in N(S)} \sum_{j \in C_i} W_{ij} \times S_j,$$

where C_i is the set of all nodes connected to node i. The energy is a non-positive quantity since (1) the initial weights are negative and (2) the heuristic learning rule only decreases the weights. In a network convergence step, the state update rule chooses a node with maximum input from within a cluster. In general the energy of the new state is higher than that of the old one. At worst, the energy remains the same. The maximum possible energy is zero, which happens only when all on-nodes' inputs are zero: a solution state. Therefore, GENET works by maximizing the energy of the network. A network may be trapped in a local maxima, whose energy is less than zero. In this case, the heuristic learning rule will be applied to alter the weight of the arcs. The net effect of the alteration is that the energy of the current state is lowered with respect to its neighbouring state, thereby destroying the current state's local maximality.

We state without proof theorem 4.2, which demonstrates that GENET is probabilistically complete.

Theorem 4.2: Assume that the randomizer is fair. If θ is a solution of a CSP P, then there exists non-zero probability that GENET will settle in a solution state that corresponds to θ. ∎

The original GENET operates in the "batch" mode: the entire network must be constructed before computation starts. In CLP systems, however, adding new constraint to an existing solvable collection is a primitive and frequent step in the operation. Therefore, an efficient incremental version of GENET is necessary. A naive but efficient incremental GENET is as follows:

```
initialize the network to the empty network
while there are more constraint increments
    add the next increment to the network
    execute GENET until it finds a solution
endwhile
```

Each execution of GENET in the incremental scheme carries forward the state and weight information from the preceding execution. Incrementality originates from the re-use of the weights of the arcs, which is computed using the heuristic learning rule in previous cycle. In other words, the network is *trained* while it is being built incrementally. Therefore, GENET satisfies incrementality.

5 Prototype and benchmarking

To demonstrate the feasibility of our proposal, we have built a prototype of PROCLANN using Quintus Prolog Release 3.1 and C. The Quintus Prolog system takes the role of the frontend CLP system, which generates constraints by goal reductions. The backend ANN-based constraint solver, incremental GENET, is implemented as a Quintus Prolog foreign C object with two interface functions: `addConstraints(L)` and `addDomain(X,D)`, where L is a list of constraints in textual form, and X is a d-variable with domain D represented as list. The `addDomain/2` predicate instructs incremental GENET to add the d-variable X^D into the network by creating a cluster of $|D|$ nodes. The goal reduction part of PROCLANN is performed naturally by Prolog. Constraints are passed to incremental GENET in a derivation step via the

graph		CHIP	PROCLANN
nodes	colors	CPU time	CPU time
125	17	> 48 hrs	2.3 hrs
125	18	> 48 hrs	2.5 mins
250	15	?	1.1 hrs
250	29	?	4.6 hrs

Table 1: Results on hard graph coloring problems.

`addConstraints/1` predicate. The following is the revised syntax of the `checkAdj/3` predicate in our prototype language:

```
checkAdj(X,[H|T],[H|Nbs]) :-
      addConstraints([ X #\= H ]),
      checkAdj(X,T,Nbs).
```

There is no Prolog coding involved. The foreign C object consists of around 1600 lines of C code.

We compare our implementation with Cosytec CHIP version 4.0.1 using the graph-coloring problem. All benchmarking was performed on a SUN SPARCstation 10 model 30. The graph-coloring problem is to color a graph, possibly non-planar, so that no two adjacent vertices in the graph have the same color. It is a practical CSP which has many useful applications, such as production scheduling. We compare PROCLANN and CHIP on a set of hard graph-coloring problems described in [11]. The PROCLANN graph-coloring program is shown in figure 1. The results are presented in table 1. The CPU time reported in the table is the median time[3] over ten runs. Note that the first-fail principle [8] is used in labeling in the CHIP program.

CHIP cannot manage to solve either one of the cases for 125 vertices within 48 hours. For the cases of 250 vertices, CHIP runs out of memory on our machine with 32M memory. PROCLANN never fails in all cases over the ten runs. These results support our conjecture that the integration of artificial neural networks into constraint logic programming provides an efficient and yet easy-to-program environment for solving CSP's.

6 Concluding remarks

The contribution of this paper is four-fold. First, we identify that artificial neural network models can be used as backend constraint-solver of a CLP system. Any ANN model that satisfies the network uniqueness, the soundness, the probabilistic completeness, and the incrementality criteria can be used for the purpose. We study the GENET model in particular and show that it satisfies the criteria. Second, we define the PROCLANN language and its computation model. Since PROCLANN is *not tied* to any specific ANN model, it can be regarded as a general theoretical framework for the integration of constraint logic programming and artificial neural network. Third, PROCLANN retains important semantic properties of logic programming, namely soundness and completeness. The latter is probabilistic. Fourth, a prototype implementation of PROCLANN is built and its efficiency compares favourably against a commercial CHIP implementation. The prototype was constructed in only two man-months. There are still many optimization possibilities. PROCLANN is also amenable to massively parallel implementation, which will yield a higher degree of speedup.

GENET handles only binary constraints over finite domains. This apparent jeopardy to the expressiveness of PROCLANN is resolved by the fact that, with finite domains, all non-binary constraints can readily be transformed into binary constraints [16]. We have not yet investigated its effect on efficiency. The GENET algorithm is semi-decidable, just as the behaviour of most other random-based algorithms. So is PROCLANN. Therefore, there is no notion of finite failure of PROCLANN and it cannot be used to detect inconsistent CSP's. To avoid infinite looping, we usually impose a limit on the number of convergence cycles or the consumption of other system resources. When PROCLANN exits from resource exhaustion, the user is not able to tell whether he/she is unlucky in that particular run or the CSP is inconsistent. Lastly, PROCLANN is a committed-choice language. It cannot handle CSP's which require tree search on constraints, an example of which is disjunctive scheduling.

The following is interesting future work. First, the inherent data parallelism in GENET suggests implementing PROCLANN on massively parallel SIMD computers. The idea is to execute goal reduction at the frontend of the SIMD machine. Constraints generated will be passed to the incremental GENET running in the massively parallel backend. Second, the incremental GENET model supports only binary constraints. A GENET architecture for solving general constraints such as illegal and atmost constraints in the car-sequencing problem is presented in [2]. It is interesting to check if the model can be adapted for

[3] Median time is used since only results of 10 runs are collected.

all general constraints. Third, we use the GENET model only as a case study. Other ANN models should be investigated. Fourth, the PROCLANN computation model is independent of the constraint domain although the PROCLANN language is tailored to finite domain constraint solving. We propose to study the suitability of other constraint domains for PROCLANN.

Acknowledgement

We are indebted to Lai Wan Chan, Ho Fung Leung, and Edward Tsang for fruitful discussions. We thank the anonymous referees for their insightful comments. Access to CLP($\mathcal{R}$), BNR Prolog, CLP(BNR), and Echidna is gratefully acknowledged. This research was supported in part by a RGC Earmarked Grant (A/C 221500260) of the Hong Kong University and Polytechnic Grants Committee.

References

[1] E.H.L. Aarts and J.H.M. Korst. Boltzmann machines for traveling salesman problems. *European Journal of Operational Research*, 39:79–95, 1989.

[2] A. Davenport, E.P.K. Tsang, C.J. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *(to appear) Proceedings of AAAI'94*, 1994.

[3] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of the European Conference on Artificial Intelligence*, pages 290–295, 1988.

[4] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:75–93, 1990.

[5] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.

[6] R. Durbin and D. Willshaw. An analogue approach to the traveling salesman problem using an elastic net method. *Nature*, 326:689–691, 1987.

[7] P. Eklund and F. Klawonn. Neural fuzzy logic programming. *IEEE Transactions on Neural Networks*, 3(5):815–818, September 1992.

[8] R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[9] J.J. Hopfield and D. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[10] J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM POPL Conference*, pages 111–119, Munich, January 1987.

[11] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by simulated annealing: an experimental evaluation; part 2 graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

[12] Oflazer K. Solving tangram puzzles : A connectionist approach. *International Journal of Neural Systems*, 8(5):603–616, 1993.

[13] A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.

[14] T.J. Reynolds, H.H. Teh, and B.T. Low. Neural logic programming. In *Proceedings of the 2nd International IEEE Conference on Tools for A.I.*, pages 485–491, 1990.

[15] O. Shagrir. A neural net with self-inhibiting units for the n-queens problem. *International Journal of Neural Systems*, 8(3):249–252, 1993.

[16] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[17] E.P.K. Tsang and C.J. Wang. A generic neural network approach for constraint satisfaction problems. In G Taylor, editor, *Neural Network Applications*, pages 12–22. Springer-Verlag, 1992.

[18] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, 1989.

[19] C.J. Wang and E.P.K. Tsang. Solving constraint satisfaction problems using neural networks. In *Proceedings of the IEE 2nd Conference on Artificial Neural Networks*, pages 295–299, 1991.

[20] C.J. Wang and E.P.K. Tsang. A cascadable VLSI design for GENET. In *Proceedings of the Oxford VLSI Workshop*, 1992.