# A Framework for Integrating Artificial Neural Networks and Logic Programming

J.H.M. Lee
Department of Computer Science
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: jlee@cs.cuhk.hk

V.W.L. Tam
Department of Computer Science
University of Melbourne, 3052
Parkville, Australia
Email: vtam@cs.mu.oz.au

**Abstract**

Many real-life problems belong to the class of constraint satisfaction problems (CSP's), which are NP-complete, and some NP-hard, in general. When the problem size grows, it becomes difficult to program solutions and to execute the solution in a timely manner. In this paper, we present a general framework for integrating artificial neural networks and logic programming so as to provide an efficient and yet expressive programming environment for solving CSP's. To realize this framework, we propose PROCLANN, a novel constraint logic programming language. The PROCLANN language retains the simple and elegant declarative semantics of constraint logic programming. Operationally, PROCLANN uses the standard goal reduction strategy in the frontend to generate constraints, and an efficient backend constraint-solver based on artificial neural networks. Its operational semantics is probabilistic in nature. We show that PROCLANN is sound and weakly complete. A novelty of PROCLANN is that while it is a committed-choice language, PROCLANN supports non-determinism, allowing the generation of multiple answers to a query. An initial prototype implementation of PROCLANN is constructed and demonstrates empirically that PROCLANN out-performs the state of art in constraint logic programming implementation on certain hard instances of CSP's.

## 1 Introduction

The problem at hand is that of *constraint satisfaction problems* (CSP's) defined in the sense of Mackworth [25], which can be stated briefly as follows:

> We are given a set of variables, a domain of possible values for each variable, and a conjunction of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a *consistent* assignment of values to the variables so that all the constraints are satisfied simultaneously.

CSP's represent a mathematical abstraction for a class of basic problems which is important in artificial intelligence, operations research and computer science in general. These general problems occur in all walks of industrial applications such as scene labeling, resource allocation, planning and scheduling, just to name a few. Most of these problems have to be solved in a timely and efficient manner in real-life situations due to their high time costs. In the case of rescue, for example, the time cost can be measured in terms of the number of lives lost per unit time. CSP's are, in general, NP-complete and some are even NP-hard [6]. Thus, a general algorithm designed to solve any CSP will necessarily require exponential time in problem size in the worst case.

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 programming 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 [41] introduces consistency techniques [25, 27] into logic programming so that constraints are used actively to prune search space *a priori*. The framework is realized in the CHIP language [12], which has been successfully applied to solving such industrial applications as car sequencing [9], disjunctive scheduling, graph coloring, and firmware design [10]. A full account of its applications can be found in [10, 41, 11, 28]. CHIP's execution mechanism is still based on tree search and backtracking, which are the main barriers to the efficient execution of the language towards some hard or large-scale CSP's.

A key component of a constraint logic programming (CLP) system is the constraint-solver. For various constraint domains, many classical constraint-solving techniques exist but the techniques are all algorithmic in nature and cannot escape from the curse of NP-completeness. Rabin [32] proposes *randomized* or *probabilistic* algorithms for concrete algorithmic problems in computer science although the idea can be traced back to the Monte Carlo methods [3, 16]. Many artificial neural network (ANN) models fall into the camp of probabilistic algorithm.

There has been recent interest in applying ANN to CSP's and constraint optimization problems. For examples, Hopfield network, the Boltzmann Machine and the elastic network have been used in the traveling salesman problems [18, 1, 13], the Tangram puzzles [23] and the $N$-queens problem [34] with satisfactory results. Wang and Tsang [43, 39] propose GENET, a generic ANN model, for solving general CSP's with binary constraints. Wang and Tsang [44] also propose a cascadable VLSI design for GENET. A VLSI implementation of GENET would provide a potential speed gain in an order of $10^6$ to $10^8$ over existing CSP languages running on commercial workstations [44].

ANN, while efficient, is 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 logic programming and ANN. A CSP is specified as a 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 *expressive* and *efficient* programming language.

There have been other attempts [14, 33], 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. Theorem proving becomes an energy minimization process. Our proposal represents a radically different approach, which aims at tight coupling of logical deduction and neural computation.

This paper develops the idea of embedding ANN-based constraint-solver in logic programming from a theoretical framework to the design and prototype implementation of a new logic programming language, and to the development of applications using the language. There are two main kinds of contribution of this paper depending on the perspective we choose. As far as the logic programming community is concerned, we propose a new logical inference system which is based on ANN while the usual logic programming semantics is retained. It is the first logical inference system which uses ANN as the constraint-solver. Our proposal may encourage more experiments on the integration of other stochastic search techniques and logic programming to improve the performance of logic programming systems. Outside the logic programming community, the contribution of this research is to present a declarative language for solving CSP's using ANN. Imperative programming languages can be used in programming solutions for CSP's but the burden of checking the correctness of the ANN model and handling the compilation errors, logical errors and exceptional cases in the program is laid upon programmers. Similar to other logic programming languages, programmers using our proposed language only have to concentrate their effort on specifying the CSP and formulating the problem statement into a query. The logic programming component of our language automatically generates constraints from the query and the program, and pass the constraints to the ANN-based constraint-solver.

The paper is organized as follows. Section 2 defines objective criteria for ANN-based constraint-solvers that can be used in our framework. Since the GENET model is adopted in our prototype implementation, we review the GENET model and show how GENET satisfies the criteria. The syntax and semantics of the PROCLANN language are presented in section 3. Soundness and weak completeness results of PROCLANN are given and the notion of probabilistic non-determinism is introduced. Section 4 describes the PROCLANN prototype implementation and benchmarking results. We summarize our contributions and limitations of PROCLANN, and shed light on future work in section 5.

## 2 An ANN-Based Constraint-Solver

ANN is chosen as the backend constraint-solver in our proposed framework for its efficiency on some large-scale or hard instances of CSP's [22, 2, 8]. In the following, we first define notations used in the paper. Next we present objective criteria for ANN models that can be used as constraint-solver in our framework. GENET is a general ANN model for solving CSP's. We review the GENET model, study its dynamics, and show that it satisfies our criteria. GENET is adopted in our prototype implementation.

### 2.1 Notations

We denote $X_1, X_2, \cdots, X_n$ the domain variables, $D_1, D_2, \cdots, D_n$ the finite domains on which the variables take their values, and $v_1 \in D_1, v_2 \in D_2, \cdots, v_n \in D_n$ the value assignments given to variables. A *unary* constraint $C_{X_i}$ is defined over $D_i$, where $1 \leq i \leq n$. A *binary* constraint $C_{X_iX_j}$ is defined over a subset of $D_i \times D_j$ for all $1 \leq i, j \leq n$ and $i \neq j$. In all cases, we assume $C_{X_iX_j} = C_{X_jX_i}$. A *general* constraint $C_{X_{i_1}X_{i_2}\cdots X_{i_k}}$ is defined over a subset of $D_{i_1} \times D_{i_2} \times \cdots \times D_{i_k}$, where $1 \leq i_1, i_2, \cdots, i_k \leq n$. A *binary CSP* is a CSP with unary and binary constraints only. A *general CSP* is one which involves general constraints.

A *label* [38], denoted by $< X, v >$, is 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 $(< X_1, v_1 >, < X_2, v_2 >, \cdots, < X_n, v_n >)$ to denote the compound label of assigning $v_1, v_2, \cdots, v_n$ to $X_1, X_2, \cdots, X_n$ respectively. A *$k$-compound label* assigns $k$ values to $k$ variables simultaneously. A *solution tuple* of a CSP with $n$ variables is a $n$-compound label for all the variables in the CSP so that all constraints are satisfied.

## 2.2 Criteria for ANN-based Constraint-solver

In this section, we present the objective criteria for the ANN-based constraint-solver: *network uniqueness*, *soundness*, *probabilistic completeness* and *incrementality*. Our proposed language framework is *not tied* to any particular ANN model but the chosen model has to satisfy the following criteria:

1. **(Network Uniqueness)** Every CSP can be translated into a network in the ANN model and the network topology is *unique*. This is to ensure that every CSP has a unique representation in the model.

2. **(Soundness)** Every answer generated by the ANN model must be a solution of the corresponding CSP. In other words, answers generated by the model must be *correct* with respect to the corresponding CSP. Soundness of the model is essential in establishing the soundness of our proposed framework, which is an important property for logic programming systems.

3. **(Probabilistic Completeness)** If a CSP has solution $\theta$, then there exists *non-zero* probability that the corresponding neural network converges to a solution state with $\theta$ as answer. We say that the neural network model is *probabilistically complete*. The usefulness of an ANN model is depleted if it is capable of only locating some but not all solutions of a CSP. The completeness criteria is probabilistic since ANN models are probabilistic in nature.

4. **(Incrementality)** Adding new constraints to an existing solvable set of constraints is a frequent and primitive step in a CLP system. Thus the model must be amenable to *efficient* incremental execution.

## 2.3 GENET: a Generic Neural Network Model

GENET [43, 39] is a generic neural network simulator designed to solve binary CSP's with finite domains. It has also been applied to solve general CSP's such as the car-sequencing problem [8]. We describe the network structure and the convergence procedure of GENET. Then we explain the behavior of GENET's network convergence from an energy perspective.

### 2.3.1 Network Structure

Given any binary CSP, GENET generates a connected network as follows:

- Each domain variable in the CSP is represented by a *cluster* (i.e. a collection) of nodes.

- Each node $i$ is used to represent a value in the domain and has two attributes: state $S_i$ and input $I_i$.

- The *state* of a node $S_i$ is either 1 for *on* or 0 for *off*.
- Connections among the nodes are constructed according to the constraints with only inhibitory connections between nodes with incompatible values. The weight of the connection between nodes $i$ and $j$ is denoted by $W_{ij}$ which is always a negative integer and initially given the value -1.
- The *input* $I_i$ to each node $i$ is a weighted sum of all its connected nodes' states, i.e. $I_i = \sum_{j=1,j\neq i}^{k} W_{ij} \times S_j$, where $k$ is the number of nodes to which node $i$ is connected.

To illustrate how a network is constructed for a binary CSP, let us take a simple but tight[1] binary CSP as example. Assume that there are five finite domain variables $X_1$ to $X_5$, all with domain $d = \{1, 2, 3\}$. There are two kinds of constraints: (1) $X_i + X_{i+1}$ is even and (2) either $X_1 = 2$ or $X_5 = 2$.

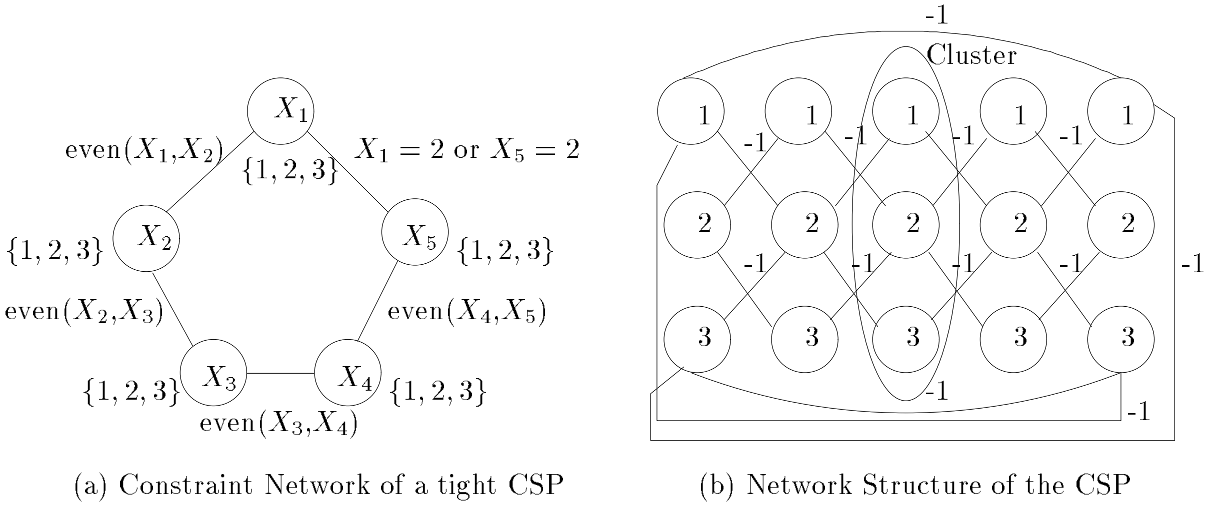



Figure 1: The constraint network and the GENET network of a tight CSP.

Figure 1 (a) is the constraint network of the example as defined in the sense of Mackworth [25]. Figure 1 (b) is the corresponding network topology in GENET. Each column of 3 nodes represents a cluster for each domain variable. And each connection is initially set to $-1$. The cross inhibitory connections are constructed according to constraints of type (1). For example, when $X_1 = 2$ and $X_2 = 1$, their sum is 3 (odd) which violates constraint of type 1. So, the second node in the first column and the first node in the second column are connected. For constraint of type (2), consider the case in which $X_1 = 1$. Thus $X_5$ should be 2. In order to fulfill this requirement, node 1 of

[1] 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.

the first cluster should send inhibitory effect to the first and third node of $X_5$. The other two connections are constructed similarly.

GENET is originally designed for binary constraints. Davenport *et al.* [8] extend GENET to a multi-layered network structure to encompass special cases of general constraints, such as the `atmost` constraints [42]. We refer the readers to [8] for details of the extended model. Our analysis of GENET in subsequent sections applies also to the extended GENET.

### 2.3.2 Network Convergence

In a GENET network, only one node in a cluster is on at any moment. We say that the variable as represented by the cluster is *assigned* the value as represented by the label of the on-node. A *network state* is defined as 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 the solution state(s). The GENET network convergence procedure is defined as pseudo-code in algorithm 1.

---

Initially one node in each cluster is randomly selected to be on.
**while** (the input to all on-nodes is non-zero) **do**

- Every node $i$ calculates its input $I_i$. In each cluster, the node with the maximum input will be turned on. To avoid chaotic or cyclic wandering of network states, if there are several nodes with the maximum input, *the node that is already on will remain on*. Otherwise, GENET selects randomly one out of them. This step is called the *state update rule*. Apply the state update rule until all on-nodes remain unchanged.

- When the network settles in a stable state (i.e. all on-nodes unchanged after the application of the state update rule), 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 *halts* with *success*. Otherwise, the state represents a *local maxima*.

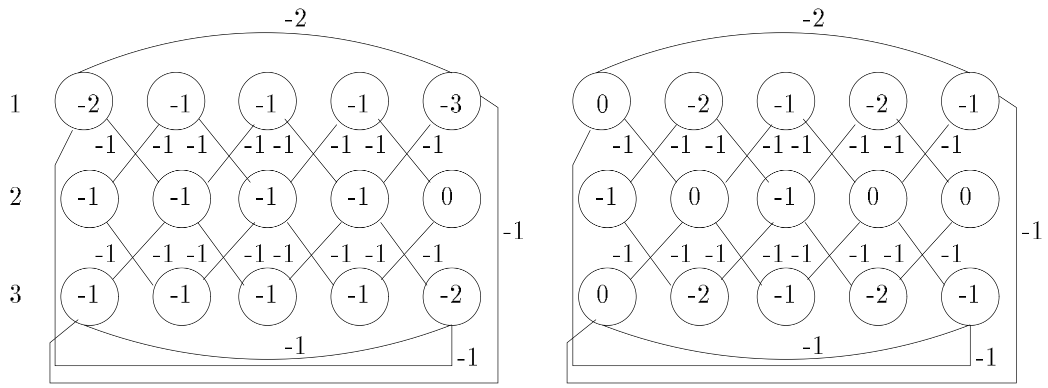
- If the network settles in a local maxima of cluster input(s), a *heuristic learning rule* is applied to update all connection weights using $W_{ij} := W_{ij} - S_i \times S_j$. Note that if either of node $i$ or $j$ is not on, then either $S_i$ or $S_j$ is zero. Thus heuristic learning affects the weights of only arcs connecting two on-nodes.

**endwhile**

Algorithm 1: The GENET network convergence procedure.

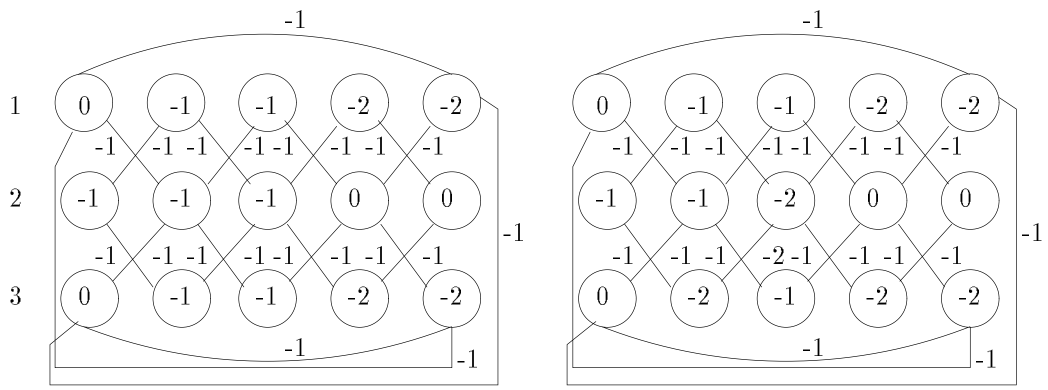
---

Figure 2 illustrates the state update rule and heuristic learning rule of GENET using the tight binary CSP shown in figure 1. Figure 2(a) shows a state transition from state 1 to state 2 using the state update rule. In the first cluster, nodes 2 and 3 have the same input. One of them is selected on at random in state 2. In this example, node 2 is chosen. In clusters 2, 3, and 4, all nodes share the same input. The ones that are already on remains on in state 2. In the fifth cluster, node 2 has the largest input and it becomes on in state 2. The input of each node in state 2 is re-computed accordingly. State 1 of figure 2(b) is trapped in a local maxima. The only on-nodes being connected are node 3 of cluster 2 and node 2 of cluster 3. The heuristic learning rule penalizes this connection by decrementing its weight by 1. The input of each node is re-computed after the weight update. The stability of the network state is thus destroyed, enabling further state updates.




State 1 : (1,3,2,2,1) State 2 : (2,3,2,2,2)

(a) State Update Rule




State 1 : (1,3,2,2,2) State 2 : (1,3,2,2,2)

(b) Heuristic Learning Rule

Figure 2: Diagrams to show the GENET network convergence.

### 2.3.3 Energy Perspective

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 $E(S)$ of a state $S$ is the sum of all on-nodes' inputs, which is a non-positive quantity since (1) all initial weights are negative and (2) the heuristic learning rule only decreases the weights. In a state 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 state. At worst, the energy remains the same. The maximum possible energy is zero, which happens only when all on-nodes' inputs are zero. That means that no two on-nodes are connected to each other for binary CSP's. Since only nodes with incompatible values are connected for binary constraints, a zero-energy state represents a solution. 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 neighboring state, thereby destroying the current state's local maximality.

## 2.4 Properties of GENET

The construction of the network structure in GENET satisfies the network uniqueness criterion by definition. The success criterion of GENET guarantees soundness.

**Theorem 2.1**: (Soundness of GENET) If the GENET network of a CSP $P$ settles in a solution state $S$, then the value assignment $\theta$ corresponding to $S$ is a solution of $P$.

**Proof:** A GENET network settles in a solution state when all on-nodes have zero input. The network is in zero-energy state. A zero-energy state means that *no two* on-nodes are connected and no on-node is inhibited by any constraint node if it exists. Thus, the value assignment to each cluster (variable) cause no violation of constraints in the network, and represents a solution to the CSP in consideration. ∎

GENET satisfies the probabilistic completeness criterion trivially.

**Theorem 2.2**: (Probabilistic Completeness of GENET) Assume that the randomizer is fair. If $\theta$ is a solution of a CSP $P$, then there exists non-zero probability that GENET will settle in a solution state that corresponds to $\theta$. ∎

**Proof:** The network convergence procedure of GENET is probabilistic in nature. The first step in the network convergence procedure is to *randomly select* an on-node in each cluster. As we assume the randomizer is fair, there is non-zero probability that the GENET network will settle in any possible network state as its initial state. Thus, there is non-zero probability that GENET will settle in any solution state of a CSP $P$ by the initial random selection. ∎

The probabilistic completeness property of GENET guarantees only a non-zero probability to find every solution of a CSP. GENET, a probabilistic simulator, may sometimes take a long time to converge and may even get into oscillation. Thus, we impose resource limit, such as memory or CPU time, on GENET's convergence procedure. When there is exhaustion of resource, we simply re-start the network convergence procedure for another iteration.

### 2.5 Incremental GENET

The original GENET operates in the "batch" mode: the entire network must be constructed before neural computation starts. In CLP systems, however, adding new constraint to an existing solvable set of constraints is a primitive and frequent step during computation. Therefore, an efficient incremental version of GENET is necessary. We present a naive incremental adaptation of GENET, I-GENET, in pseudo-code in algorithm 2. Since the GENET network convergence procedure is

---

Initialize the network to an empty network
**while** there are more domain variables and increments of constraints to add **do**

- Create a cluster of nodes for each domain variable added.
- **foreach** (each pair of nodes and each constraint in the increment) **do**
  **if** the values denoted by the pair of nodes violates the constraint **then**
  Create an arc between the nodes and set its weight to -1.
  **endforeach**
- Execute the GENET convergence procedure until success or resource exhaustion.

**endwhile**

Algorithm 2: The I-GENET network convergence procedure.

---

part of the I-GENET procedure, the network state is randomized in each iteration of I-GENET. Incrementality originates from the re-use of the weights of the arcs, which are computed using the heuristic learning rule and accumulated in each iteration. In other words, the network is *trained* while it is being built incrementally.

Before we construct incremental GENET, we have in fact modified the way node inputs are calculated. The inputs of all nodes are calculated in each convergence cycle of the original GENET model. In the modified version, we calculate the inputs of only the nodes whose neighboring nodes change state. This modification is implemented in the modified GENET and I-GENET. We compare the original GENET, the modified GENET and I-GENET on the $N$-Queens problem in figure 3.

The graphs are generated by executing the three versions of GENET on the $N$-Queens problems from 10 to 100 queens on a SUN SPARCstation 10 model 30. For each test case, the measurement recorded is an average of ten runs. The modified GENET shows a speedup over GENET on the

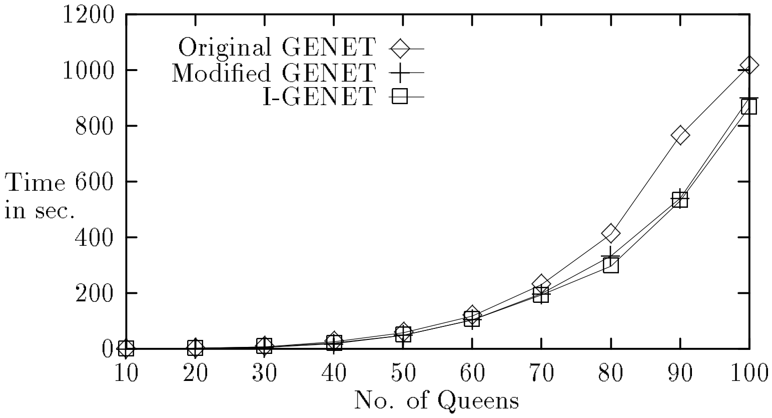



Figure 3: Comparison of GENET, Modified GENET and I-GENET.

$N$-Queens problem. The result also shows that I-GENET is at least as efficient as the modified GENET on the tested problems. Thus we show empirically that GENET satisfies the incrementality criterion.

# 3 A Theoretical Framework of PROCLANN

In this section, we describe **PRO**gramming in **C**onstraint **L**ogic with **A**rtificial **N**eural **N**etwork (PROCLANN), which is a CLP language for handling constraints over finite domains. PROCLANN, a committed-choice language, features a tight coupling of logical deduction and neural computation. The operational semantics of PROCLANN is similar to that of the CLP scheme [19, 20] but the use of an ANN-based constraint-solver is made explicit in the *PROCLANN Computation Model*. PROCLANN is both *sound* and *weakly complete*. Although it is a committed-choice language, PROCLANN supports the notion of *probabilistic non-determinism*, allowing the generation of multiple answers to a query.

## 3.1 Syntax and Declarative Semantics

Since we are interested in CSP, we need the notion of domains and domain variables [40], abbreviated as *d-variables*. A *domain* is a non-empty 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 [24], we call the latter *h-variables*. We refer to $h$-variables and $d$-variables as *variables*.

A PROCLANN term is the same as a CHIP term [40]. The constraint domain $\mathcal{D}$ in consideration defines a set $\mathcal{C}$ of *primitive constraints*. As specified in the CLP scheme [19, 20], the equality constraint **#=** is included in $\mathcal{C}$. An *atom* is any standard well-formed formula whose predicate symbol is different from those of the primitive constraints. A *clause* in PROCLANN has the form

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

where

1. $H$ (the *head*) is an atom, $\vec{G}$ (the *guard*) is a conjunction, possibly empty, of built-in test predicates (such as $X > Y, X \leq Y$), $\vec{C}$ (the *constraints*) is a conjunction, possibly empty, of primitive constraints, and $B_1, \ldots, B_n$ (the *body*) is a conjunction, possibly empty, 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 [40]. A *program* is a finite 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 [40].

We conclude our description by two PROCLANN programs. Figure 4 shows a PROCLANN

```
nqueen(N,L) :- length(L,N), L in 1..N, constraint(L).

length([],0).
length([H|T],N) :- N > 0 | N1 is N-1, length(T,N1).

constraint([]).
constraint([X|Y]) :- safe(X,Y,1), constraint(Y).

safe(X,[],Nb).
safe(X,[H|T],Nb) :- noattack(X,H,Nb), Newnb is Nb + 1, safe(X,T,Newnb).

noattack(X,Y,Nb) :- X #\= Y, X #\= Y+Nb, Y #\= X+Nb.
```

Figure 4: A PROCLANN program for the $N$-Queens problem.

program for the $N$-Queens problem, the task of which is to place $N$ queens onto a $N \times N$ chessboard so that no two queens attack each other. Two queens attack each other if they are on the same row, column or diagonal. We use one $d$-variable to denote the row number of a queen on each column. The built-in predicate `in/2` declares that each element of a list is a $d$-variable with domain specified as a list or range. The goal "`L in 1..N`" declares a list `L` of $d$-variables, each of which has domain $\{1, \ldots, \texttt{N}\}$. The "`#\=`" operator stands for the disequality constraint. Thus the `noattack/3` predicate specifies that no two queens can be on the same row or the same diagonals.

```
testGraph(N,L) :- graph(L,Graph), colorGraph(N,L,Graph).

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

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

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

Figure 5: A PROCLANN program for the graph-coloring problem.

Figure 5 contains a PROCLANN program for the graph-coloring problem, the task of which is to color all nodes in a graph, possibly non-planar, so that no two connected nodes share the same color. An example graph is shown in figure 6. The graph is encoded in the `graph/2` predicate as

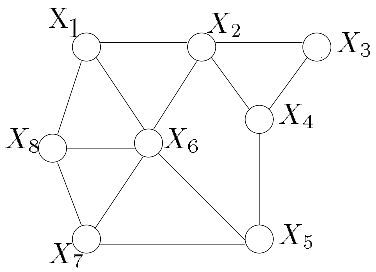



Figure 6: An example graph.

follows:

```
graph([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]]]).
```

The first argument of the `graph/2` predicate is a list `[X1,X2,X3,X4,X5,X6,X7,X8]`, each element of which is declared as a $d$-variable denoting a node in the graph. The second argument of `graph/2` is a list `Graph` of adjacency lists. Each adjacency list specifies a node $N$ followed by a list $L$ of nodes adjacent to $N$. The `colorVertices/1` predicate takes an adjacency list "`[N,L]`" and specifies the corresponding disequality constraints between `N` and the elements of `L`. The $N$-Queens program and the graph-coloring program are almost identical to their corresponding CHIP counterparts, except that there is no need to perform explicit labeling operation using the CHIP `indomain/1` predicate [41].

### 3.2 Unification in PROCLANN

Since PROCLANN and CHIP both handle constraints over finite-domain variables, the PROCLANN unification scheme is an adaptation of the $d$-unification scheme in CHIP [40]. A substitution for the $h$-variable and $d$-variable in the $d$-unification scheme is called *d-substitution*.

**Definition 3.1**: A *d-substitution* $\theta$ is a finite set of the form $\{X_1/t_1, \cdots, X_n/t_n\}$ where

1. each $X_i$ is a variable;
2. each $t_i$ is a term distinct from $X_i$;
3. all the $X_i$ are distinct;
4. if $X_i$ is a $d$-variable $X^e$, $t_i$ is a constant included in $e$ or is a $d$-variable $X^l$ with $l \subset e$.

Each element $X_i/t_i$ is called a binding for $X_i$. $\theta$ is called a *ground substitution* if the $t_i$'s are all ground terms. An *expression* is either a term, a literal, or a conjunction or disjunction of literals. A *simple expression* is either a term or an atom. We denote the identity substitution by $\xi$ with which $E\xi = E$ for all expression $E$. The definitions of composition, domain and co-domain of $d$-substitutions and the most general $d$-unifier are given in the following.

**Definition 3.2**: Let $\theta = \{u_1/s_1, \cdots, u_m/s_m\}$ and $\sigma = \{v_1/t_1, \cdots, v_n/t_n\}$ be $d$-substitutions. Then the composition $\theta\sigma$ of $\theta$ and $\sigma$ is the $d$-substitution obtained from the set

$$\{u_1/s_1\sigma, \cdots, u_m/s_m\sigma, v_1/t_1, \cdots, v_n/t_n\}$$

by deleting any binding $u_i/s_i\sigma$ for which $u_i = s_i\sigma$ and deleting any binding $v_j/t_j$ for which $v_j \in \{u_1, \cdots, u_m\}$.

**Definition 3.3**: Let $\theta$ and $\sigma$ be $d$-substitutions, $W$ and $V$ be expressions and $W \subset V$. $\sigma$ is an instance of $\theta$ and $\theta$ is more general than $\sigma$ on $W$, denoted by $\sigma \geq \theta\,[W]$ if and only if there exists a $d$-substitution $\delta$ satisfying $\sigma = \theta\delta\,[W]$.

**Definition 3.4**: Let $\theta$ be a $d$-substitution and $V$ be the set of variables in the expressions. The *domain* of a substitution, denoted $\text{dom}(\theta)$ is defined as $\{x \in V | x\theta \neq x\}$. The *codomain* of a substitution, denoted $\text{codom}(\theta)$ is defined as $\{x\theta | x\theta \neq x\}$. Also, the set of variables in the expression $T$ is denoted $\text{var}(T)$.

**Definition 3.5**: Let $S = \{s_1, \cdots, s_n\}$ be a finite set of simple expressions. A $d$-substitution $\theta$ is called a $d$-unifier for $S$ if and only if $s_1\theta = \cdots = s_n\theta$. A $d$-unifier $\theta$ for $S$ is called a *most general unifier* (mgu) for $S$ if and only if for each $d$-unifier $\sigma$ of $S$, $\sigma \geq \theta\,[\text{var}(S)]$. We say that $S$ is *unifiable*.

The unification algorithm in PROCLANN is an adaptation of $d$-unification [41]. As in $d$-unification, we have three cases to consider in addition to the conventional unification algorithm [26]:

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 if $d \cap e \neq \emptyset$ and the binding $\{X^d/Y^e\}$ is generated.

3. If a $d$-variable $X^d$ is unified with a constant $c$, unification succeeds if $c \in d$ and the binding $\{X^d/Y^{\{c\}}\}$ is generated.

A detailed unification algorithm is given in [37].

A PROCLANN computation can be viewed as an interleaving of logical deduction and neural computation. The former produces bindings for the $h$-variables while the latter produces bindings for the $d$-variables. Accordingly, we partition the $d$-substitution into two parts: the *$\alpha$-substitution* which is the $d$-substitution restricted to the $h$-variables and the *$\beta$-substitution* which is the $d$-substitution restricted to the $d$-variables. The $\alpha$- and $\beta$-substitutions are special cases of the $d$-substitution. Hence, the notion of composition, domain and codomain of $\alpha$- and $\beta$-substitutions follow those of $d$-substitution. The definition of most general unifier is extended to $\alpha$- and $\beta$-substitutions.

**Definition 3.6**: Let $S = \{s_1, \cdots, s_n\}$ be a finite set of simple expressions. A $\alpha$-substitution $\vartheta$ is called a $\alpha$-unifier if and only if there exists a $d$-unifier $\theta$ for $S$ and $\vartheta$ contains the set of bindings for all the $h$-variables in $\theta$. A $\alpha$-unifier $\vartheta$ is called a *most general $\alpha$-unifier* if and only if there exists a most general $d$-unifier $\sigma$ for $S$ and $\vartheta$ contains the set of bindings for all the $h$-variables in $\sigma$.

**Definition 3.7**: Let $S = \{s_1, \cdots, s_n\}$ be a finite set of simple expressions. A $\beta$-substitution $\varepsilon$ is called a $\beta$-unifier if and only if there exists a $d$-unifier $\theta$ for $S$ and $\varepsilon$ contains the set of bindings for all the $d$-variables in $\theta$. A $\beta$-unifier $\varepsilon$ is called a *most general $\beta$-unifier* if and only if there exists a most general $d$-unifier $\sigma$ for $S$ and $\varepsilon$ is the set of bindings for all the $d$-variables in $\sigma$.

The bindings generated from case (2) and (3) of the unification algorithm are composed to form the $\beta$-substitution for the $d$-variables. Since the bindings in the $\beta$-substitution are meant to be consumed by the ANN-based constraint-solver of PROCLANN, they have to be transformed to a set of equality constraints before they are passed to the constraint-solver. The following definition shows the transformation of a $\beta$-substitution to a set of *equality constraints* defined in PROCLANN.

**Definition 3.8**: A *$d$-replacement* $d(\varepsilon)$ of *$\beta$-substitution* $\varepsilon = \{X_1/Y_1, \cdots, X_n/Y_n\}$ is a finite set of the form $\{X_1 \texttt{ #= } Y_1, \cdots, X_n \texttt{ #= } Y_n\}$. The set $\{X_1 \texttt{ #= } Y_1, \cdots, X_n \texttt{ #= } Y_n\}$ obtained from the $d$-replacement is then injected into the ANN-based constraint-solver for neural computation. Hence, the unification scheme in PROCLANN facilitates logical deduction and neural computation using $\alpha$- and $\beta$-substitutions.

### 3.3 The PROCLANN Computation Model

The **PROCLANN I**nference **R**ule (**PROCLANN-IR**) forms the core part of the PROCLANN computation model.

**Definition 3.9**: 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 the most general $\alpha$-unifier $\vartheta_i$ and most general $\beta$-unifier $\varepsilon_i$, and
   (b) the execution of the guard $\vec{G}$ succeeds;
3. the corresponding neural network of $\vec{C''} = \vec{C} \cup d(\varepsilon_i) \cup \vec{C'}$ converges to a solution state in finite amount of time;
4. $Q_{i+1} = \leftarrow \vec{C''}, B_1, \ldots, B_{j-1}, A_1, \ldots, A_n,\ B_{j+1}, \ldots, B_q$.

**Definition 3.10**: Let $Q$ be the initial query. A *derivation* is a, possibly infinite, sequence of queries $Q_0(=Q), Q_1, Q_2, \ldots$ such that $Q_{i+1}$ is derived from $Q_i$ generating substitution $\theta_{i+1}(=\alpha_{i+1} \cup \beta_{i+1})$. A PROCLANN derivation may be finite or infinite. A finite derivation may be successful or failed.

**Definition 3.11**: 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 query is called the *answer constraint* and the associated neural network is called the *answer network*.

**Definition 3.12**: A derivation is *failed* if it is finite, the last query has one or more atoms, and condition 2 of a derivation step does not hold.

**Definition 3.13**: A PROCLANN-refutation is a successful derivation by PROCLANN-IR with $n$ finite steps. We say that the refutation has *length* $n$.

**Definition 3.14**: An unrestricted PROCLANN-refutation is a PROCLANN-refutation, except that we drop the requirement that the substitutions $\vartheta_i$ and $\varepsilon_i$ be the most general $\alpha$-unifier and the most general $\beta$-unifier. They are only required to be $\alpha$-unifiers and $\beta$-unifier respectively.

**Definition 3.15**: Let $P$ be a PROCLANN program. The *success set* of $P$ is the set of all ground atoms $A$ such that there exists a PROCLANN-refutation for $P \cup \{\leftarrow A\}$.

**Definition 3.16**: A *computed answer* of a successful derivation is the union of a solution state $\theta_N$ of the answer network $N$ and the composition of the most general $\alpha$-unifiers $\theta_h = \vartheta_1 \cdots \vartheta_m$, where $\vartheta_i$ is the most general $\alpha$-unifier generated in the $i$-th step in the derivation.

The PROCLANN refutation procedure works by searching for a PROCLANN-refutation out of many possible derivations. Note that, in a derivation step, there can be more than one clause in the program that satisfies condition 2 of a derivation step. We call these clauses *candidate clauses*. The solution computed depends upon the choice of candidate clause, to which the evaluation is committed. PROCLANN does not specify which candidate clause to select if there is a choice. It relies on a *fair*[2] scheduling algorithm [15] in the underlying implementation to select and commit

[2] Shapiro *et al.* [35, 36] define the notion of "stable" implementation of Concurrent Prolog and show how fairness can be controlled by the programmer. Parlog programmers do not have to rely on a stable implementation. Fairness can be programmed using the `var` primitive [4, 5].

to a candidate clause. Thus PROCLANN does not support tree search on constraints. In addition, it is possible that PROCLANN chooses a wrong clause in a derivation step that leads to a failed derivation even when there exists a successful derivation for the original query. This is typical of committed-choice languages. Nevertheless, PROCLANN is sound and weakly complete.

### 3.4 Soundness and Weak Completeness

The soundness of PROCLANN-IR follows essentially from the soundness criterion of the ANN constraint-solver.

**Theorem 3.17**: (Soundness of the PROCLANN Computation Model) Let $P$ be a PROCLANN program, $\mathcal{T}$ be a first-order theory that axiomatizes the constraint domain $\mathcal{D}$ and $\leftarrow Q$ a query. If the query $\leftarrow Q$ has a successful derivation with computed answer $\theta$ by PROCLANN-IR, then

$$P \cup \mathcal{T} \models_{\mathcal{D}} \forall(Q\theta)$$

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

**Proof:** Let $Q = \vec{C}, A_1, \cdots, A_l$, where $\vec{C} = c_1, \cdots, c_k$ is a conjunction of primitive constraints defined in $\mathcal{D}$ and $A_1, \cdots, A_l$ is conjunction of atoms, $\vec{C_N}$ be the answer constraint, $\theta_N$ be the solution state of the answer network $N$ and $\theta_h = \vartheta_1 \cdots \vartheta_n$ be the composition of the most general $\alpha$-unifiers for a successful derivation. We have to show that $\forall((c_1 \wedge \cdots \wedge c_k \wedge A_1 \wedge \cdots \wedge A_l)\theta)$ is a logical consequence of $P \cup \mathcal{T}$, where $\theta = \theta_h \cup \theta_N$. The result is provable by induction on the length of the derivation.

Suppose first that $n = 1$. This means that $\leftarrow Q$ is a query of the form $\leftarrow c_1, A_1$. The program has a unit clause of the form $A \leftarrow c$ and $\vec{C_N} = c \cup c_1 \cup d(\varepsilon_1)$, where $\varepsilon_1$ is the most general $\beta$-unifier generated for the unification of $A$ and $A_1$. Since $\mathcal{T} \models_{\mathcal{D}} \forall(\vec{C_N}\theta_N)$ by the soundness criterion of the ANN adopted in PROCLANN and $A_1\theta_1 = A\theta_1$,

$$P \cup \mathcal{T} \models_{\mathcal{D}} \forall((\vec{C_N}\theta_N) \wedge (A_1\theta_1)).$$

Since $\vec{C_N}$ do not contain any $d$-variables, it can be re-written as follows:

$$\begin{aligned} P \cup \mathcal{T} &\models_{\mathcal{D}} \forall((\vec{C_N}(\theta_N \cup \theta_h)) \wedge (A_1\theta_1)) \\ &\models_{\mathcal{D}} \forall((\vec{C_N}\theta_1) \wedge (A_1\theta_1)) \\ &\models_{\mathcal{D}} \forall((\vec{C_N} \wedge A_1)\theta_1) \\ P \cup \mathcal{T} &\models_{\mathcal{D}} \forall((c_1 \wedge A_1)\theta_1) \ \text{( since } c_1 \in \vec{C_N}) \end{aligned}$$

Hence, $\forall(Q\theta)$ is a logical consequence of $P \cup \mathcal{T}$ for $n = 1$.

Next suppose the result holds for computed answer $\theta_{n-1}$ which come from a successful derivation with $n-1$ steps. Suppose $\theta_n$ is the computed answer for a successful derivation with $n$ steps. Let $A \leftarrow \vec{C'}, B_1, \cdots, B_j$ be the first input clause and $A_m$ the selected atom of query $Q$. By the induction hypothesis, $\forall((\vec{C}_{n-1} \wedge \vec{C'} \wedge A_1 \wedge \cdots \wedge A_{m-1} \wedge B_1 \wedge \cdots \wedge B_j \wedge A_{m+1} \wedge \cdots \wedge A_l)\theta_n)$ is a logical consequence of $P \cup \mathcal{T}$, where $\vec{C}_{n-1}$ is the answer constraint for a successful derivation with $n-1$ steps. Thus, if $j > 0$, $((\vec{C'} \wedge B_1 \wedge \cdots \wedge B_j)\theta_n)$ is a logical consequence of $P \cup \mathcal{T}$. Consequently, $(A\theta_n)$ is a logical consequence of $P \cup \mathcal{T}$. $\forall((\vec{C}_{n-1} \wedge \vec{C'} \wedge A_1 \wedge \cdots \wedge A_l)\theta_n)$ is a logical consequence of $P \cup \mathcal{T}$.

Hence $\forall((c_1 \wedge \cdots \wedge c_k \wedge A_1 \wedge \cdots \wedge A_l)\theta)$ is a logical consequence of $P \cup \mathcal{T}$ since $\vec{C}_{n-1} \wedge \vec{C'}$ contains $c_1 \wedge \cdots \wedge c_k$. ■

The probabilistic completeness criterion of ANN leads to the *weak completeness* result of PROCLANN-IR. We begin with two very useful lemmas.

**Lemma 3.18**: (Mgu Lemma) Let $P$ be a PROCLANN program and $Q$ be a query. Suppose that $P \cup \{Q\}$ has an unrestricted PROCLANN-refutation. Then $P \cup \{Q\}$ has a PROCLANN-refutation of the same length such that, if $\vartheta_1, \cdots, \vartheta_n$ are the $\alpha$-unifiers from the unrestricted PROCLANN-refutation and $\vartheta'_1, \cdots, \vartheta'_n$ are the most general $\alpha$-unifiers from the PROCLANN-refutation, then there exists a $\alpha$-substitution $\gamma$ such that $\vartheta_1, \cdots, \vartheta_n = \vartheta'_1, \cdots, \vartheta'_n\gamma$.

**Proof:** The proof is by induction on the length of the unrestricted PROCLANN-refutation. Suppose that $n = 1$. Thus $P \cup \{Q\}$ has an unrestricted refutation $Q_0 = Q, Q_1 = \Box$ with input clause $H_1$ and $\alpha$-unifier $\vartheta_1$, where $\Box$ stands for the empty clause. Suppose $\vartheta'_1$ is a most general $\alpha$-unifier of the atom in $Q$ and the head of the unit clause $H_1$. Then $\vartheta_1 = \vartheta'_1\gamma$, for some $\gamma$. Furthermore, $P \cup \{Q\}$ has a PROCLANN-refutation $Q_0 = Q, Q_1 = \Box$ with input clause $H_1$ and the most general $\alpha$-unifier $\vartheta'_1$.

Now suppose the result holds for $n-1$. Suppose $P \cup \{Q\}$ has an unrestricted refutation $Q_0 = Q, Q_1, \cdots, Q_n = \Box$ of length $n$ with input clauses $H_1, \cdots, H_n$ and $\alpha$-unifiers $\vartheta_1, \cdots, \vartheta_n$. There exists a most general $\alpha$-unifier $\vartheta'_1$ for the selected atom in $Q$ and the head of $H_1$ such that $\vartheta_1 = \vartheta'_1\rho$, for some $\rho$. Thus $P \cup \{Q\}$ has an unrestricted refutation $Q_0 = Q, Q'_1, Q_2, \cdots, Q_n = \Box$ with input clauses $H_1, \cdots, H_n$ and $\alpha$-unifiers $\vartheta'_1, \rho\vartheta_2, \vartheta_3, \cdots, \vartheta_n$, where $Q_1 = Q'_1\rho$. By the induction hypothesis, $P \cup \{Q'_1\}$ has a PROCLANN-refutation $Q'_1, Q'_2, \cdots, Q'_n = \Box$ with the most general $\alpha$-unifiers $\vartheta'_1, \cdots, \vartheta'_n$ such that $\rho\vartheta_2, \cdots, \vartheta_n = \vartheta'_2, \cdots, \vartheta'_n\gamma$, for some $\gamma$. Thus $P \cup \{Q\}$ has a refutation $Q_0 = Q, Q'_1, \cdots, Q'_n = \Box$ with the most general $\alpha$-unifiers $\vartheta'_1, \cdots, \vartheta'_n$ such that $\vartheta_1 \cdots \vartheta_n = \vartheta'_1\rho\vartheta_2 \cdots \vartheta_n = \vartheta'_1 \cdots \vartheta'_n\gamma$. ■

**Lemma 3.19**: (Lifting Lemma) Let $P$ be a PROCLANN program, $Q$ be a query and $\vartheta$ be a $\alpha$-substitution. Suppose that there exists a PROCLANN-refutation of $P \cup \{Q\}$ of the same length such that, if $\vartheta_1, \cdots, \vartheta_n$ are the most general $\alpha$-unifiers from the PROCLANN-refutation of $P \cup \{Q\theta\}$ and $\vartheta'_1, \cdots, \vartheta'_n$ are the most general $\alpha$-unifiers from the PROCLANN-refutation of $P \cup \{Q\}$, then there exists a $\alpha$-substitution $\gamma$ such that $\vartheta\vartheta_1 \cdots \vartheta_n = \vartheta'_1 \cdots \vartheta'_n\rho$.

**Proof:** Suppose the first input clause for the PROCLANN-refutation of $P \cup \{Q\vartheta\}$ is $H_1$, the first most general $\alpha$-unifier $\vartheta_1$ and $Q_1$ is the query which results from the first step. We may assume $\vartheta$ does not act on any variables in $H_1$. Now $\vartheta\vartheta_1$ is a $\alpha$-unifier for the head of $H_1$ and the atom in $Q$ which corresponds to the selected atom in $Q\vartheta$. The result of resolving $Q$ and $H_1$ using $\vartheta\vartheta_1$ is exactly $Q_1$. Thus we obtain an unrestricted PROCLANN-refutation of $P \cup \{Q\}$, which looks exactly like the given PROCLANN-refutation of $P \cup \{Q\}$, except the original query is different, of course, and the first unifier is $\vartheta\vartheta_1$. Now apply the mgu lemma. ■

We now state the first completeness result.

**Theorem 3.20**: The success set of a PROCLANN program is equal to its least Herbrand model.

**Proof:** The proof is omitted here but is exactly the same as that for theorem 8.3 in [24]. The proof makes use of fixed point theory and requires the definition of several notions not introduced at here. ■

The following lemma assists in the proof of the weak completeness result.

**Lemma 3.21**: Let $P$ be a PROCLANN program and $A$ an atom. Suppose that $\forall(A)$ is a logical consequence of $P$. Then there exists a PROCLANN-refutation of $P \cup \{\leftarrow A\}$ with the identity substitution as the computed answer.

**Proof:** Suppose $A$ has variables $X_1, \cdots, X_n$. Let $a_1, \cdots, a_n$ be distinct constants not appearing in $P$ or $A$ and let $\theta$ be substitution $\{X_1/a_1, \cdots, X_n/a_n\}$. Then it is clear that $A\theta$ is a logical consequence of $P$. Since $A\theta$ is ground, theorem 3.20 shows that $P \cup \{\leftarrow A\theta\}$ has a refutation. Since the $a_i$ do not appear in $P$ or $A$, by replacing $a_i$ by $x_i (i = 1, \cdots, n)$ in this refutation, we obtain a refutation of $P \cup \{\leftarrow A\}$ with the identity substitution as the computed answer. ■

**Theorem 3.22**: (Weak Completeness of the PROCLANN Computation Model) Let $P$ be a PROCLANN program, $\mathcal{T}$ be a first-order theory that axiomatizes the constraint domain $\mathcal{D}$ and $\leftarrow Q$ a query. If $P \cup \mathcal{T} \models_{\mathcal{D}} \forall(Q\theta)$ with $\theta = \theta_N \cup \theta_h$ a substitution, where $\theta_N$ contains only bindings for $d$-variables and $\theta_h$ contains only bindings for $h$-variables, then there exists a successful derivation for $\leftarrow Q$ with answer network $N$ such that

- there is a *non-zero* probability that $N$ will converge into the solution state $\theta_N$, and
- if $\theta'_h = \vartheta'_1 \cdots \vartheta'_m$ is the composition of $\alpha$-substitutions generated in the successful derivation, then there exist $\alpha$-substitution $\sigma$ such that $\theta_h = \theta'_h\sigma$.

**Proof:** Suppose $Q$ is the query $\leftarrow A_1, \cdots, A_k$. $\forall((A_1 \wedge \cdots \wedge A_k)\theta)$ is a logical consequence of $P$. By lemma 3.21, there exists a refutation of $P \cup \mathcal{T} \cup \{\leftarrow A_i\theta\}$ such that the computed answer is the identity, for $i = 1, \cdots, k$. We can combine these refutations into a refutation of $P \cup \mathcal{T} \cup \{Q\theta\}$ such that the computed answer is the identity.

Suppose the sequence of the most general $\alpha$-unifier of the refutation of $P \cup \mathcal{T} \cup \{Q\theta\}$ is $\vartheta_1, \cdots, \vartheta_n$. Then $Q\theta_h\vartheta_1, \cdots, \vartheta_n = Q\theta_h$. By the lifting lemma, there exists a PROCLANN-refutation of $P \cup \mathcal{T} \cup \{Q\}$ with the most general $\alpha$-unifiers $\vartheta'_1, \cdots, \vartheta'_n$ such that $\theta_h\vartheta_1 \cdots \vartheta_n = \vartheta'_1 \cdots \vartheta'_n\sigma$, for some $\alpha$-substitution $\sigma$. Hence, $\theta_h = \theta'_h\sigma$. The answer network $N$ will have non-zero probability to converge to the solution state $\theta_N$, which is guaranteed by the the probabilistic completeness criterion of the ANN adopted in PROCLANN. ■

Theorem 3.22 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. Note that our completeness results are existential in the sense that we can guarantee the existence of a successful derivation but PROCLANN may not find it. This is a consequence of the committed-choice nature of PROCLANN; hence the name weak completeness.

### 3.5 Probabilistic Non-determinism

One distinguished feature of PROCLANN is that while it is a committed-choice language but yet it supports both *don't-care* and *don't-know* non-determinisms, allowing the generation of multiple answers to a query. A PROCLANN computation can be divided into two parts: network generation and solution generation. Don't-care non-determinism is exhibited in the first part, in which the program is used as a network generator using goal reduction on guarded clauses. The neural network, while being built up incrementally, is also subject to network training. A successful derivation leads to an answer network. Don't-know non-determinism originates from the second part of the execution, in which the answer network is further executed using the network convergence procedure. *As in the case of Prolog*, PROCLANN users can request answers one after another. Upon request, PROCLANN *only has to re-start the second part of the execution*. In fact, the solution generation part is relatively inexpensive with respect to the overall cost of execution since much time is spent in the construction of the network, where a form of consistency check is performed. The implication is that we can generate subsequent answers at relatively low cost. In most conventional logic programming languages, the cost of getting subsequent answers grows exponentially in general. Last but not least, answers are produced randomly; hence the name *probabilistic non-determinism*.

## 4 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 Prolog system takes the role of the frontend CLP system for constraint generations using goal reductions. The backend ANN-based constraint-solver, incremental GENET (I-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 I-GENET to add the $d$-variable $\mathtt{X}^{\mathtt{D}}$ into the network by creating a cluster of $|\mathtt{D}|$ nodes. The goal reduction part of PROCLANN is performed naturally by Prolog. Constraints are passed to I-GENET in a derivation step via the `addConstraints/1` predicate. For example, the following is the revised syntax of the `checkAdj/3` predicate of the graph-coloring program:

```
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 code.

We compare the performance of our prototype implementation and the current state of art of CLP implementation, the Cosytec CHIP version 4.0.1 language on two CSP's. Our benchmarks include the $N$-Queens problem and some simple and hard graph-coloring problems. We also compare PROCLANN with the forward-checking algorithm with dynamic variable ordering and the same algorithm augmented with conflict-directed back-jumping on an instance of exceptionally hard problems (EHP's). Results show that CHIP is more efficient than PROCLANN on the $N$-Queens problem which contains much symmetry. On all instances of the graph-coloring problem under test, PROCLANN out-performs CHIP. The efficiency of PROCLANN is best demonstrated in the hard graph-coloring problems. EHP's are designed to defeat forward-checking algorithms, which form the core part of CHIP. PROCLANN is not hindered by the instance of EHP that we test. All benchmarking is performed on a SUN SPARCstation 10 model 30. Time unit is in seconds of CPU time.

### 4.1 $N$-Queens

The $N$-Queens problem is to place $N$ queens onto a $N \times N$ chessboard so that none of the queens attack each other. Two queens attack each other if they are placed on the same column, row or diagonal. The $N$-Queens problem is a common benchmark for CSP's since the number of queens can be increased without limit.

#### 4.1.1 Benchmarking

The PROCLANN $N$-Queens program is shown in figure 4. The CHIP counterpart is adopted from the one that comes with the Cosytec package. Note that the first-fail principle [17] is used for labeling in the CHIP program. PROCLANN execution is probabilistic in nature. The recorded time for PROCLANN represents timing for average of ten runs.

Figure 7 reflects a comparison between PROCLANN and CHIP. The recorded time for CHIP is the time taken to obtain the *first* solution. The curve for CHIP exhibits irregularity, with a sudden peak at 90-Queens. Figure 8 has the 90-Queens anomaly removed. The PROCLANN time rises exponentially from 10- to 140-Queens while the CHIP time does not show any significant increase. At 149- and 150-Queens, CHIP does not return in 24 hours. PROCLANN works at 149-Queens but 150-Queens exceeds the capacity of the Prolog engine part of PROCLANN.

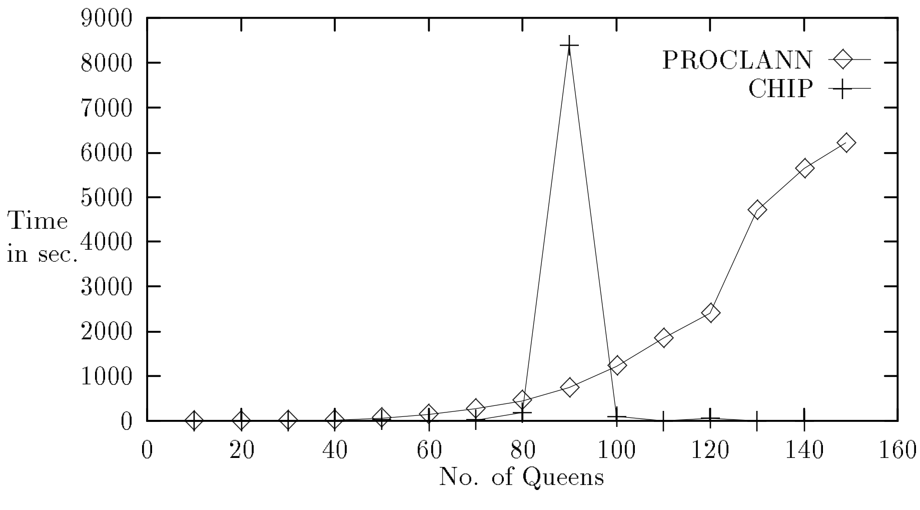



Figure 7: A First Comparison between PROCLANN and CHIP.

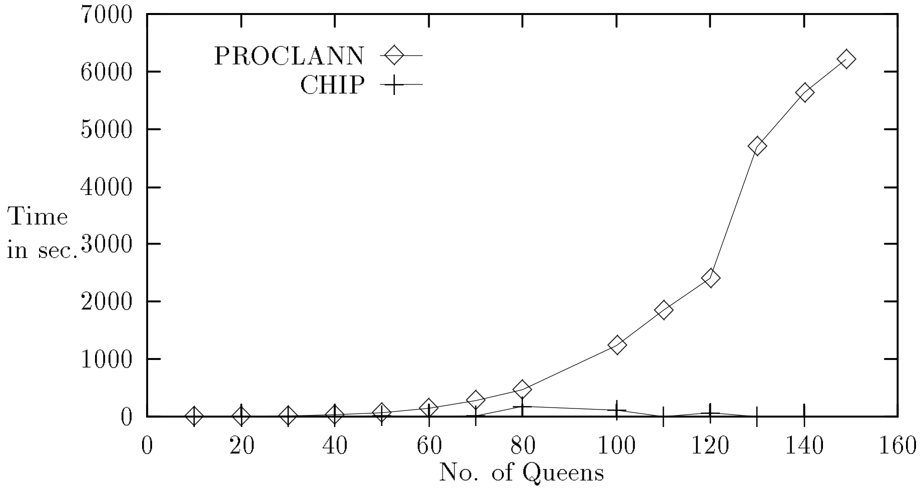



Figure 8: Ignoring the 90-Queens Anomaly.

#### 4.1.2 Analysis

Similar to Prolog, CHIP employs a left-to-right depth-first search strategy to explore the derivation tree of a query. The time to find the first solution depends on the distribution of solutions in the tree, and in particular, the location of the first solution branch in the tree. If the branches to the left of the first solution branch are "bushy" and "deep," then much time is spent in backtracking before reaching the first solution. The CHIP approach is incredibly sensitive to data [7]. Any random change in the definition of constraints can result in drastic different runtime behavior. This explains the anomaly at 90-Queens.

In general, CHIP out-performs PROCLANN on finding the first solution from 10- to 140-Queens. PROCLANN execution can be divided into two parts: network building and solution finding by ANN. The former consists of creating clusters of nodes and making inhibitory connections between nodes. The latter is realized by executing the I-GENET convergence procedure. Figure 9 shows the breakdown of the total processing time of PROCLANN for the $N$-Queens problem into these two components. In making inhibitory connections, a form of consistency checking is performed. For the $N$-Queens problem, PROCLANN has to build large networks which grow exponentially in size. Thus PROCLANN spends considerable time in building the network, which consumes most of the time for finding the first solution in figure 8. The time for exercising the network convergence procedure is, in general, relatively small. For CHIP, however, the execution time always depends on the amount and depth of the backtracking required for the branches to the left of the first solution. Since the constraints for the $N$-Queens problems are symmetric, for example $X \neq Y + 1$ and $Y \neq X + 1$ where $X$ and $Y$ are $d$-variables representing the row numbers of queens, the solutions occur in pairs and are symmetrical about the central vertical axis of the associated search tree. Therefore, the first solution must be located in the left subtree if solutions exist. In the worst case,

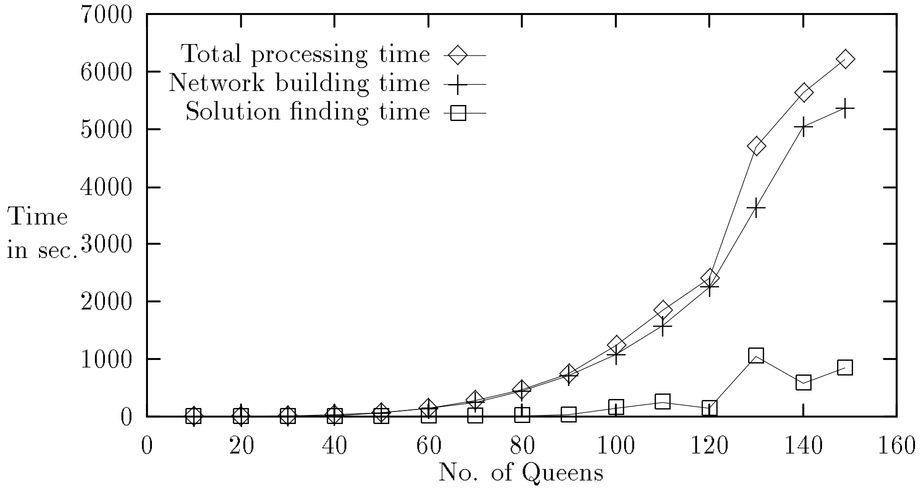



Figure 9: Breakdown of PROCLANN Execution Time.

CHIP have to traverse the whole left subtree before it can find the first solution.

Tsang [38] points out that benchmarks on different algorithms produced using the $N$-Queens problem must be interpreted with caution since the $N$-Queens problem has very specific features: it is a binary CSP in which every variable is constrained by every other variable, which need not be the case in other CSP's. More importantly, in the $N$-Queens problem, each label for every variable conflicts with at most three values of each other variable, regardless of the number of variables (i.e. $N$) in the problem. For example, $< 1, 2 >$ has conflict with $< 2, 1 >, < 2, 2 >, < 2, 3 >$. In the 8-Queens problem, for example, when 2 is assigned to queen 1, there are 5 out of 8 values that queen 2 can take. In the $1,000,000$-Queens problem, however, there are $999,997$ out of $1,000,000$ values that queen 2 can take after $< 1, 2 >$ has been committed to. Therefore, constraints get looser as $N$ grows larger. Such features may not be shared by many other CSP's.

## 4.2 Graph-coloring

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 computer applications. Production scheduling, construction of examination timetables, and the storages of goods can all be stated as graph-coloring problems [40]. We compare PROCLANN and CHIP on some simple randomly generated graph-coloring problems and a set of hard graph-coloring problems described in [21].

### 4.2.1 Benchmarking

The PROCLANN graph-coloring program is shown in figure 5. The results are presented in tables 1 and 2 respectively. The CPU time of PROCLANN reported in the tables is the average time over ten runs. The first-fail principle [17] is also used for labeling in the CHIP program.

Table 1 gives the timing recorded for both CHIP and PROCLANN to solve the simple graph-coloring problems from 10- to 250-vertices. They both manage to solve the problems in a timely manner. PROCLANN out-performs CHIP in all runs. Figure 10 shows the comparison between CHIP and PROCLANN on these simple problems. The execution time for PROCLANN grows much slower than that of CHIP. PROCLANN requires, in general, less than half of the CHIP time to find a solution in all the cases. Table 2 records timing for four hard graph-coloring problems. CHIP cannot solve either of the problems with 125 vertices within 48 hours. For problems with 250 vertices, CHIP runs out of memory on our machine with 64M memory. PROCLANN never fails in the ten runs.

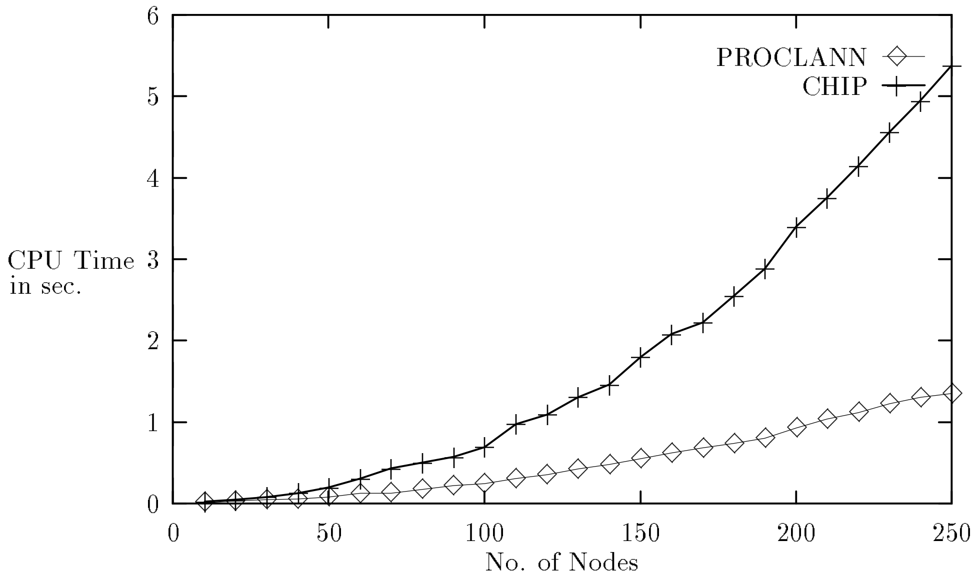



Figure 10: Comparison between CHIP and PROCLANN on simple graph coloring problems.

### 4.2.2 Analysis

The number of colors used for coloring the graph is kept constant at 10 from 110 to 250 vertices. When the number of vertices increases, the size of the search tree grows exponentially. Hence, the CHIP time shows an exponential growth since CHIP employs a left-to-right depth-first search of the derivation tree. For the hard instances of the graph-coloring problem, PROCLANN can

| graph | | CHIP | PROCLANN |
|---|---|---|---|
| vertices | colors | CPU time | CPU time |
| 10 | 3 | 0.020 | 0.0165 |
| 20 | 4 | 0.050 | 0.033 |
| 30 | 5 | 0.090 | 0.05 |
| 40 | 5 | 0.130 | 0.058 |
| 50 | 6 | 0.200 | 0.083 |
| 60 | 6 | 0.310 | 0.125 |
| 70 | 7 | 0.430 | 0.137 |
| 80 | 8 | 0.5 | 0.183 |
| 90 | 9 | 0.57 | 0.2165 |
| 100 | 9 | 0.7 | 0.2415 |
| 110 | 10 | 0.98 | 0.3085 |
| 120 | 10 | 1.1 | 0.35 |
| 130 | 10 | 1.31 | 0.4255 |
| 140 | 10 | 1.46 | 0.4755 |
| 150 | 10 | 1.81 | 0.55 |
| 160 | 10 | 2.08 | 0.6165 |
| 170 | 10 | 2.23 | 0.683 |
| 180 | 10 | 2.55 | 0.742 |
| 190 | 10 | 2.89 | 0.8 |
| 200 | 10 | 3.4 | 0.933 |
| 210 | 10 | 3.75 | 1.033 |
| 220 | 10 | 4.15 | 1.1245 |
| 230 | 10 | 4.56 | 1.2245 |
| 240 | 10 | 4.95 | 1.3 |
| 250 | 10 | 5.38 | 1.35 |

Table 1: Benchmarks on simple graph coloring problems.

| graph | | CHIP | PROCLANN |
|---|---|---|---|
| vertices | 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 2: Benchmarks on hard graph coloring problems.

solve all the problems while CHIP fails to return answer for any of them. The size of the GENET networks for the graph-coloring problems is much smaller than that for the $N$-Queens problem. Thus, PROCLANN spends relatively little time in network building. Most of the PROCLANN time is spent in the network convergence to find solutions. The branches to the left of the first solutions for these hard problems usually require much backtracking. Thus, CHIP performs much backtracking before it finds the first solution in the search tree. This accounts for CHIP's failure on finding solution for any of these hard CSP's.

### 4.3 Exceptionally Hard Problem

This section discusses an instance of exceptionally hard problems (EHP's) [31], which consists of 50 variables, each with domain $\{1, 2, 3, 4, 5, 6, 7, 8\}$. The constraint graph for this problem is connected. The tightness of the problem is 0.06.

#### 4.3.1 Benchmarking

The forward-checking algorithm with dynamic variable ordering (fc-dvo) [29], which always chooses variables with the smallest current domain, cannot solve the problem after 711 million consistency checks. However, the same algorithm augmented with conflict-directed back-jumping method (fc-cbj-dvo) [30] finds a solution with 9588 consistency checks. PROCLANN solves the problem with 2448 convergence cycles in 3.24 seconds on average of 10 runs.

#### 4.3.2 Analysis

The EHP at hand contains solution(s) which is very sparse in the search space. Thus, it is a very tight problem. It is an instance of CSP's designed especially to defeat fc-dvo. The performance of PROCLANN is not affected by this hard instance of CSP's.

## 5 Concluding Remarks

### 5.1 Contribution

The contribution of this paper is six-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. Second, we study the GENET model and its dynamics. We show that GENET satisfies the objective criteria required for incorporation in the PROCLANN framework. Third, 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 logic programming and artificial neural networks. Fourth, we prove important semantic properties of PROCLANN, namely soundness and weak completeness. The latter is probabilistic in nature. Fifth, we introduce non-determinism into a committed-choice language. This is a *novel* feature of PROCLANN. Sixth, a prototype implementation of PROCLANN is built using I-GENET and its efficiency compares well against forward-checking algorithms. The prototype is 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.

### 5.2 Limitations

GENET can only handle binary constraints over finite domains and a limited number of special general constraints. 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 [38]. We have not yet investigated its effect on efficiency.

The GENET algorithm is semi-decidable, just as the behavior of most other random-based algorithms. It cannot be used to detect inconsistent CSP's. To avoid infinite looping, we usually impose 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.

PROCLANN is a committed-choice language. It cannot handle CSP's which require tree search on constraints, an example of which is disjunctive scheduling. A remedy is to define appropriate disjunctive constraints, such as "$X_1 = 2$ or $X_5 = 2$," tailored for each application.

### 5.3 Future Work

There are a few directions for future work. First, the inherent data parallelism in GENET suggests implementation of 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 I-GENET running in the massively parallel backend. Second, the I-GENET model supports only binary constraints. A GENET architecture for solving general constraints, such as the `illegal` and the `atmost` constraints, in the car-sequencing problem is presented in [8]. It should be interesting to check if the model can be adapted for other 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. The suitability of other constraint domains for PROCLANN are topics of further studies.

## Acknowledgement

We are indebted to Lai Wan Chan, Ho Fung Leung, and Edward Tsang for fruitful discussions. We benefit from discussion with Andrew Davenport, who also provides pointers to our benchmark problems. Patrick Prosser supplies the source of the EHP. We are grateful to the anonymous referees of both TAI'95 and IJAIT for their insightful comments. This research was supported in part by a RGC Earmarked Grant (CUHK 70/93E) of the Hong Kong University Grants Committee and a Direct Grant of the Chinese University of Hong Kong.

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

[3] K. Binder and D.W. Heermann. *Monte Carlo Simulation in Statistical Physics.* Springer-Verlag, 1988.

[4] J.D. Broda and S. Gregory. PARLOG for discrete event simulation. In *Proceedings of the Second International Logic Programming Conference*, pages 301–312, 1984.

[5] K.L. Clark and S. Gregory. Parlog: Parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.

[7] Y. Cras. Using constraint logic programming in services: A few short tales. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 3–16, 1994.

[8] 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 *Proceedings of AAAI'94*, 1994.

[9] 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.

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

[11] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. Applications of CHIP to industrial and engineering problems. In *Proceedings of the 1st International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, pages 887–892, 1988.

[12] 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.

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

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

[15] S. Gregory. *Parallel Logic Programming in PARLOG: The Language and Its Implementation*. Addison-Wesley, 1987.

[16] J. Hammersley and D.C. Handscomb. *Monte Carlo Methods*. Chapman and Hall, Inc., 1964.

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

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

[19] J. Jaffar and J-L. Lassez. Constraint logic programming. Technical Report 72, Department of Computer Science, Monash University, Clayton, Victoria, Australia, 1986.

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

[21] 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.

[22] M.D. Johnston and H.M. Adorf. Learning in stochastic neural networks for constraint satisfaction problems. In *Proceedings of NASA Conference on Space Telerobotics*, Pasadena, CA, 1989.

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

[24] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

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

[26] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:797–281, 1982.

[27] B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[28] M. Perrett. Using constraint logic programming techniques in container port planning. *ICL Technical Journal*, 7(3):537–545, May 1991.

[29] P. Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 262–267, 1993.

[30] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.

[31] P. Prosser. Exceptionally hard problems. The comp.constraints newsgroup, September 1994.

[32] M.O. Rabin. Probabilistic algorithms. In J.F. Traub, editor, *Algorithms and Complexity, Recent Results and New Directions*, pages 21–39. Academic Press, 1976.

[33] 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.

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

[35] E. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report CS83-06, Weizmann Institute of Science, Israel, February 1983.

[36] E.Y. Shapiro and C. Mierowsky. Fair, biased, and self-balancing merge operators. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 83–90, 1984.

[37] V.W.L. Tam. Integrating artificial neural networks and constraint logic programming. Master's thesis, Department of Computer Science, The Chinese University of Hong Kong, Hong Kong, 1995.

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

[39] 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.

[40] P. Van Hentenryck. *Consistency Techniques in Logic Programming.* PhD thesis, University of Namur, Belgium, 1987.

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

[42] P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 745–759, 1991.

[43] 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.

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