Extending GENET with lazy arc consistency

Peter Stuckey and Vincent Tam
Department of Computer Science,
University of Melbourne, 3052
Parkville, Australia.
{pjs,vtam}@cs.mu.oz.au

Technical Report 96/8

Abstract

Constraint satisfaction problems (CSP's) naturally occur in a number of important industrial applications, such as planning, scheduling and resource allocation. GENET is a neural network simulator to solve binary constraint satisfaction problems. GENET uses a convergence procedure based on a relaxed form of local consistency to find assignments which are locally minimal in terms of constraint violation. It uses heuristic learning to escape local minima which do not represent solutions. We describe a lazy arc consistency technique which is suitable for integration into the convergence procedure of GENET. We compare the efficiency of the GENET using lazy arc consistency against GENET, both alone and using a full arc consistency preprocessing step, on a number of hard or large instances of binary CSP's. GENET with lazy arc consistency betters the original GENET on instances of binary CSP's which are not arc consistent in their original formulation, and does not suffer the overhead of full arc consistency for problems whose original formulation is arc consistent.

1 Introduction

Constraint satisfaction problems (CSP's) occur in a large number of industrial applications such as planning, resource allocation, and scheduling. A CSP involves a set of variables, each of which has a finite and discrete domain of possible values, and a constraint formula, limiting the combination of values for a subset of variables. A binary CSP is a CSP involving constraints which restrict only one or two variables at a time. The task of solving a CSP is to assign values to the variables so that the constraint formula is satisfied.

Solving CSP's is, in general, NP-complete. Thus, a general algorithm designed to solve any CSP will necessarily require exponential time[1] in the worst case. Traditionally, there are a number of different approaches proposed to solve CSP's efficiently. For instances, constraint logic programming systems has been successfully used to tackle a number of industrial CSP applications such as car sequencing [3], disjunctive scheduling [13] and firmware design [4]. Stochastic search methods, such as simulated annealing, neural networks and evolutionary algorithms, have also had remarkable success in solving industrial CSP's [12] and CSP optimization problems [1, 5].

Among the stochastic solvers, GENET [12, 14] is a neural network simulator to solve binary CSP's with finite domains. It is shown to be efficient for solving certain hard or large instances of CSP's [14, 2] . GENET has been enhanced in a number of ways. It was augmented to handle *atmost* and *illegal* constraints in order to solve car-sequencing problems [2]. Lee, Leung and Won [7] describe an extended form of GENET, E-GENET, which has a generic representation scheme for handling general constraints. All the forms of GENET work, in general, by using a convergence procedure to achieving a relaxed form of local consistency, and then use learning to escape local minima. We propose a kind of lazy arc consistency which is suitable for integrating into the convergence procedure of GENET and E-GENET for any binary constraint to provide additional information for reducing the domain size of each variable, so as to improve the efficiency of the network convergence.

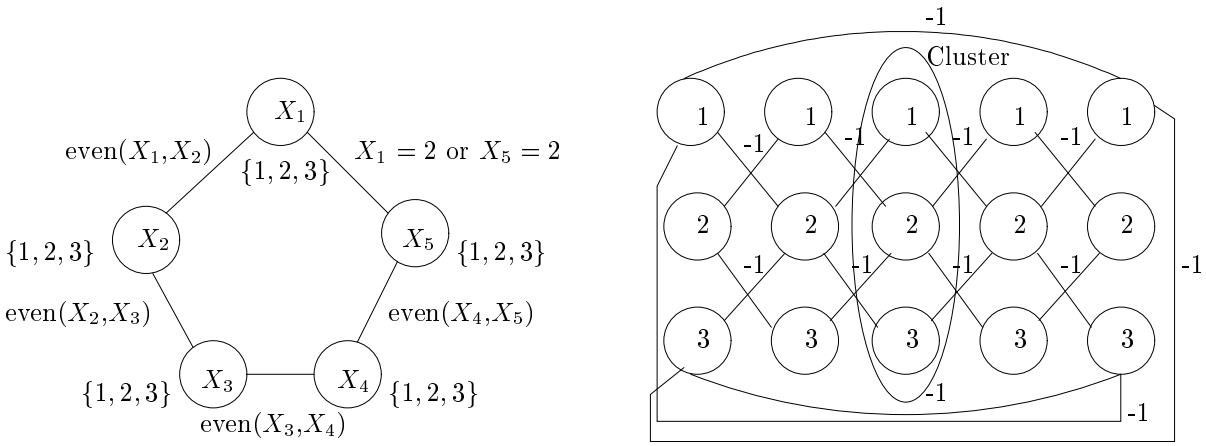
This paper is organised as follows. Section 2 describes the GENET model. We show through an example how the network convergence procedure of GENET works. In section 3 we define arc consistency, and give a straightforward procedure for converting a binary CSP which is not arc consistent into an equivalent problem which is arc consistent. In section 4, we describe the lazy arc consistency technique and show how we adopt the technique in the network convergence procedure of the GENET models. Section 5 describes our experimental results comparing GENET and versions using lazy and full arc consistency on a number of binary CSP's. Lastly, we summarize the contribution of our works and shed light on future directions in section 6.

2 A Constraint Solver : GENET

GENET [12, 14], a generic neural network simulator, can be used to solve binary CSP's. A GENET network consists of a cluster of nodes for each variable V_i in a CSP. Each node $(V_i = d)$ denotes an assignment of the value d to the corresponding variable V_i, there is one node for each value in the domain of the variable $domain(V_i)$. And each constraint is represented by a set of inhibitory connections between nodes with incompatible values.

To illustrate how a GENET network is constructed for a CSP, we use a simple but tight binary CSP as an example. Assume there are five finite domain variables V_1 to V_5, all with domain $domain(V_i) = \{1, 2, 3\}$. There are two kinds of primitive constraints: (1) $V_i + V_{i+1}$ is even and (2) either $V_1 = 2$ or $V_5 = 2$.

[1] Assuming $P \neq NP$.



(a) Constraint Network of a tight CSP

(b) Network Structure of the CSP

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

Figure 1 (a) is the constraint network of the example defined in the sense of Mackworth [9]. Figure 1 (b) is the corresponding network topology in GENET. Each column of 3 nodes represents a cluster for each variable. Each node represents an assignment of a value to the variable of the cluster. In the figure the cluster for V_3 is circled. The 3 nodes represent the assignments $V_3 = 1$, $V_3 = 2$ and $V_3 = 3$.

Connection weights $weight(V_i, d_i, V_j, d_j)$ between two nodes $V_i = d_i$ and $V_j = d_j$ are initialized as follows: If $V_i = d_i \wedge V_j = d_j$ is inconsistent for some binary constraint, then $weight(V_i, d_i, V_j, d_j) = -1$ otherwise $weight(V_i, d_i, V_j, d_j) = 0$. For example $V_1 = 2$ and $V_2 = 1$ violates the first constraint because their sum is odd, hence $weight(V_1, 2, V_2, 1) = -1$. In Figure 1 (b) each of the arcs represents a weight of -1 and no arc between two nodes means the weight is 0.

At any moment, only one node in a cluster is *on*. The variable represented by the cluster is said to be *assigned* the value given by the node which is on. Let $value(V_i)$ denotes the value currently assigned to V_i. This means that the state of the entire network represents a valuation for the variables, that is an assignment of a unique value to each variable. The input $input(V_i, d)$ to each node $V_i = d$ is defined as the weighted sum of all the weights connecting it to other nodes which are on, that is $input(V_i, d) = \sum_{j \neq i} weight(V_i, d, V_j, value(V_j))$. The nodes' states and the connection weights of a GENET network are changed by the network convergence procedure which is stated in pseudo-code as follows.

```
network_converge(L_cc):{true, unknown}
    conv_cycle := 0
    foreach variable V_i                              /* for each var */
        value(V_i) := random_select(domain(V_i));     /* randomly select an assignment */
    endforeach
    repeat
        repeat                          /* A Convergence Cycle */
```

```
        converge := true;
        foreach variable V_i
            converge := converge & state_update(V_i);
        endforeach
    until (converge)        /* until no change in any assigned value */
    if (Σ_{V_i} input(V_i, value(V_i)) < 0) then learn();    /* learning */
    conv_cycle++;
until (Σ_{V_i} input(V_i, value(V_i)) = 0 || conv_cycle > L_cc)
if (conv_cycle > L_cc) then return unknown      /* resource exhausted */
return true                                     /* converge to a solution */
```

Similar to other iterative improvement algorithms, the GENET network convergence procedure (network_converge) has a limit L_{cc} imposed on the number of convergence cycles used. A convergence cycle is defined in network_converge as the execution of the inner **repeat** loop once. Initially, a value for each variable V_i is selected randomly from its domain $domain(V_i)$ by the function $random\_select(domain(V_i))$. Then, the state update rule (state_update(V_i)) is applied to each variable until there is no change in the value assigned to each variable, that is the network is at a local minima. The sum of inputs of the values assigned to all the variables is checked. If this sum is less than zero then the assignment violates some of the constraints and is therefore not a solution. In this case a heuristic learning rule (learn()) is applied to penalize violating value-pairs in the assignment and execution continues. Execution stops if the sum of inputs of the values is zero, (a solution has been found) or the number of convergence cycle exceeds the limit L_{cc} (resources have been exhausted). If resources have been exhausted, network_converge returns *unknown* indicating its failure to find a solution. Otherwise, it returns *true*. Note that when network_converge returns *unknown* the CSP may have a solution or no solution.

The state update rule and heuristic learning rule are stated in pseudo-code as follows:

```
state_update(V_i):{true, false}
    old_value := value(V_i);
    foreach d ∈ domain(V_i)
        input(V_i, d) := 0;
        foreach variable V_j ≠ V_i
            penalty := weight(V_i, d, V_j, value(V_j));
            input(V_i, d)+= penalty;
        endforeach
    endforeach
    Let D := {d_1, ··· , d_k} be values for which each input(V_i, d_i) is maximum;
    if old_value ∈ D then return true;
    else value(V_i) := random_select(D);
    endif
    return false;
```

The procedure state_update starts by saving the old value assigned to variable V_i. Then, it calculates the input of each value for the variable V_i and forms a set D which contains the values with maximum inputs (that is least conflict) for the variable. If the old value is contained in D, then state_update returns *true*, since the variable V_i does not need to change value. Otherwise, a value in D is randomly selected and assigned to V_i, and state_update returns $false$.

```
learn()
    foreach variable V_i
        foreach variable V_j ≠ V_i
```

if $(weight(V_i, value(V_i), V_j, value(V_j)) < 0)$ **then**
$weight(V_i, value(V_i), V_j, value(V_j)) -= 1;$
endforeach
endforeach

The procedure learn() works by searching through each pair of nodes $V_i = value(V_i)$ and $V_j = value(V_j)$ of the current assignment and penalizes those that violate constraints (i.e. $weight(V_i, value(V_i), V_j, value(V_j)) < 0$) by reducing the connection weights by 1. In this way, the pairs of variable assignments which violate a constraint are discouraged from being selected again by state_update since their inputs are reduced by the learning rule.

Consider an example problem follows GENET structure which corresponds to a CSP with $V_1 < V_2, V_2 < V_3$ and each variable has domain $\{1, 2, 3\}$. Figure 2(a) shows the GENET structure of such CSP with the initial assignment $(V_1 = 2, V_2 = 2, V_3 = 1)$. Arcs in the diagram show pairs of nodes where the weights have value -1. No arc indicates weight of 0. Updating the value for

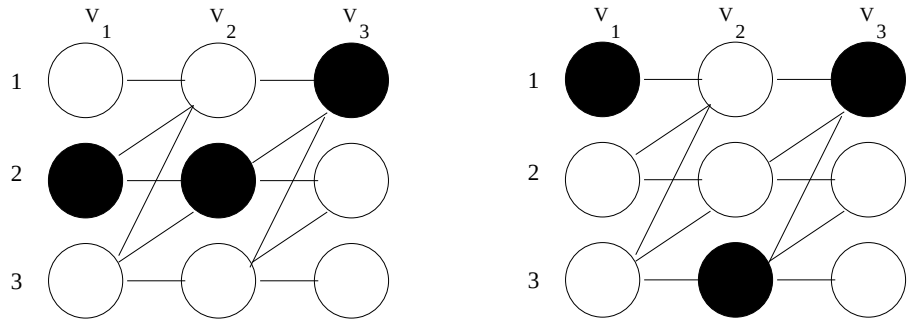


Figure 2: The GENET structure of an example CSP

V_2, we in turn calculate the inputs for each nodes as follows:

$$\begin{array}{lllll}
[V_2 = 1] & \text{is adjacent to } V_1 = 2 & \text{weight } (-1) & \\
 & \text{is adjacent to } V_3 = 1 & \text{weight } (-1) & input(V_2, 1) = -2 \\
[V_2 = 2] & \text{is adjacent to } V_1 = 2 & \text{weight } (-1) & \\
 & \text{is adjacent to } V_3 = 1 & \text{weight } (-1) & input(V_2, 2) = -2 \\
[V_2 = 3] & \text{is adjacent to } V_3 = 1 & \text{weight } (-1) & input(V_2, 3) = -1
\end{array}$$

Hence, 3 is the new value selected for variable V_2 by the state update rule. Figure 2(b) shows the GENET structure for the same CSP with the initial value assignment as $(V_1 = 1, V_2 = 3, V_3 = 1)$. The inputs of all the values in the GENET network are shown as a 3×3-matrix as follows:

$$\begin{bmatrix} 0^* & -2 & -1^* \\ 0 & -1 & -1 \\ -1 & -1^* & -1 \end{bmatrix}$$

Since the current values (marked by asterisk) each have maximum input in their clusters, they are not changed by state_update, that is the assignment is a local minimum. But the sum of inputs of the values assigned to the variables is $0 + (-1) + (-1) = -2$ is not equal to zero, hence

the assignment is not a solution. Thus, the heuristic learning rule is applied to the network and penalizes the only connected assignment-pair ($V_2 = 3, V_3 = 1$) by reducing the connection weight by 1 to -2. Thus, the inputs to $V_2 = 3$ and $V_3 = 1$ are now -2 which causes $V_2 = 2$ to be selected in the next state update of V_2. Alternatively, $V_3 = 2$ or $V_3 = 3$ may be chosen when updating V_3. In this way, the heuristic learning rule helps to escape from the local minimum.

3 Arc consistency

A binary CSP is *arc consistent* when for each variable V_i and value d in its domain $domain(V_i)$ then for each other variable V_j there exists value $d_j \in domain(V_j)$ such that $V_i = d \wedge V_j = d_j$ is compatible with the constraint between V_i and V_j.

Arc consistency gives us a way of removing useless values of the domains of variables which are guaranteed not to appear in any solution. Any value d for variable V_i clearly cannot take part in a solution of the CSP unless it is compatible with some value d_j for each of the other variables V_j. Hence, values which are not arc consistent can be removed.

Given a network representing a binary CSP we can preprocess it to create an arc consistent network by removing values from domains which are not arc consistent. We are guaranteed that any solution to the original network is a solution of the arc consistent network. The following pseudo code performs the preprocessing. It is a straightforward test of each possible variable and value. Note that once a value of a variable is found to be inconsistent, and removed from the domain of the variable, we need to recheck all the other variables and their values which may no longer be arc consistent.

```
arc_consistent()
    repeat
        changes := false
        foreach variable V_i
            foreach d ∈ domain(V_i)
                foreach variable V_j ≠ V_i
                    possibly_inconsistent := true
                    foreach d_j ∈ domain(V_j)
                        if weight(V_i, d, V_j, d_j) == 0 then possibly_inconsistent := false
                    endforeach
                    if possibly_inconsistent then
                        domain(V_i) := domain(V_i) − {d}
                        changes := true
                        break
                    endif
                endforeach
            endforeach
        endforeach
    until ¬changes
```

For example given the network of Figure 2, applying the arc_consistent procedure examines each node $V_1 = i$ in turn. Examining $V_1 = 3$ versus the variable V_2 clearly each possible value of V_2 is inconsistent with $V_1 = 1$ hence it is removed from the domain of V_1. Examining V_2 the node $V_2 = 1$ is inconsistent with each (remaining) value for V_1 and the node $V_2 = 3$ is inconsistent with all values of V_3 hence both are removed. Now examining V_3 both nodes $V_3 = 1$ and $V_3 = 2$

are inconsistent with the remaining node $V_2 = 2$ for V_2 hence they are removed. In the next iteration $V_1 = 2$ is inconsistent with remaining node $V_2 = 2$ for V_2 and is removed. No other changes are made. Finally in the last iteration no changes are found. Clearly now the only possible assignment is in fact a solution (of course this is not necessarily the case).

Arc consistency is in general an expensive operation. For the simple example network it removes many nodes, but in many instances it cannot remove any nodes. Applying arc consistency as a preprocessing step for GENET is worthwhile if the reduction in search space yields an improvement in the convergence of the GENET algorithm that gains more time than the arc consistency requires. This may not always be the case (see Empirical results section), and hence motivates the use of lazy arc consistency.

4 Lazy Arc Consistency

The network convergence procedure of GENET trys to attain a relaxed form of local consistency which minimizes the number of constraint violations that a variable has with the other variables in the CSP. It does not remove values (label) from any domain variable (cluster). Thus, it is attractive to have an arc consistency technique to remove incompatible values from the domain variables to help the network convergence procedure of GENET achieve local consistency faster. We propose a "lazy" form of the arc consistency technique which only enforces arc consistency for the value (node) assigned to each domain variable. It can readily be achieved by the network convergence procedure of the GENET model without much extra computation while gaining a reasonable amount of useful information to improve the efficiency of the search.

The GENET network convergence procedure with lazy arc consistency we propose is re-written in pseudo-code as follows:

```
state_update(V_i):{true, false}
    old_value := value(V_i);
    if (inconsistent(V_i) := true)
    then domain(V_i) := domain(V_i) − old_value;
    foreach variable V_j ≠ V_i
        possibly_inconsistent(V_j) = true;
    endforeach
    foreach d ∈ domain(V_i)
        input(V_i, d) := 0;
        foreach variable V_j ≠ V_i
            penalty := weight(V_i, d, V_j, value(V_j));
            if (penalty = 0) then possibly_inconsistent(V_j) := false;
            input(V_i, d)+= penalty;
        endforeach
    endforeach
    foreach variable V_j ≠ V_i
        if (possibly_inconsistent(V_j)) then inconsistent(V_j) := true;
    endforeach
    Let D := {d_1, ··· , d_k} be values for which each input(V_i, d_i) is maximum;
    if old_value ∈ D then return true;
    else value(V_i) := random_select(D);
    endif
    return false;
```

The above algorithm is almost identical to that given in section 2. New code is boxed. Notice

that only a single line is added to the inner loop. Lazy arc consistency plays its role in removing arc inconsistent values assigned to each variable.

The algorithm works as follows. If the current value assigned to V_i has been detected as arc inconsistent, then $inconsistent(V_i)$ is $true$. In this case the value is removed from the domain of V_i. Then the algorithm assumes for all the other variables (V_j's), current values ($value(V_j)$) may be possibly arc inconsistent with the current variable V_i. In the calculation of $input$, when it identifies that at least one of the values of V_i is compatible with the value currently assigned to another variable V_j, it immediately resets the flag for possible arc inconsistency of V_j to be $false$. After the inputs are calculated, the algorithm marks all the remaining possibly arc inconsistent values as arc inconsistent for subsequent processing. These values will be removed when the variables involved are updated. Computation proceeds by selecting the value with maximum input in the usual way.

Consider the same example CSP which we discussed in the first section. The CSP has three variables: $V_1 < V_2, V_2 < V_3$ and each variable has domain $\{1, 2, 3\}$. The GENET structure of the CSP is shown in figure 2(a) with the initial value assignment $\{V_1 = 2, V_2 = 2, V_3 = 1\}$. Now, let us consider the updating of the value for V_2 by the new state update rule with lazy arc consistency. We, in turn, assume inconsistency for each other variables, calculate the inputs for each nodes and remove incompatible values if possible in the following way:

1. For each other variable (V_1,V_3), set $possibly\_inconsistent(V_i) = true$.
2. Calculate the inputs for each node:

$[V_2 = 1]$	is adjacent to $V_1 = 2$	weight (-1)	
	is adjacent to $V_3 = 1$	weight (-1)	$input(V_2, 1) = -2$
$[V_2 = 2]$	is adjacent to $V_1 = 2$	weight (-1)	
	is adjacent to $V_3 = 1$	weight (-1)	$input(V_2, 2) = -2$
$[V_2 = 3]$	not adjacent to $V_1 = 3$		$possibly\_inconsistent(V_1) = false$
	is adjacent to $V_3 = 1$	weight (-1)	$input(V_2, 3) = -1$

3. Mark value $V_3 = 1$ as inconsistent
4. Select 3 as the new value for V_2

Hence, value 3 is the new value for V_2. Since the value $V_3 = 1$ is found to be inconsistent with all the values in V_2, it is marked as inconsistent. Then, when we update V_3, the execution of the new state update rule proceeds as follow:

1. Delete the inconsistent value 1 from the domain of V_3
2. For each other variable (V_1,V_2), set $possibly\_inconsistent[V_i] = true$.
3. Calculate the inputs for each node:

$[V_3 = 1]$	removed		
$[V_3 = 2]$	not adjacent to $V_1 = 2$		$possibly\_inconsistent(V_1) = false$
	is adjacent to $V_2 = 3$	weight (-1)	$input(V_3, 2) = -1$
$[V_3 = 3]$	is adjacent to $V_2 = 3$	weight (-1)	$input(V_3, 3) = -1$

4. mark value $V_2 = 3$ as inconsistent
5. randomly choose one node, say $V_3 = 3$

Hence, 3 is the new value selected for variable V_3 by the new state update rule and the node $V_2 = 3$ is slated for removal.

Lazy arc consistency is "lazy" since it only checks that the value assigned to the variable being updated is arc consistent with other variables in the network. Such laziness avoids a significant proportion of the computation required by the full arc consistency technique while providing extra useful information for the GENET convergence procedure to reduce the domain sizes of variables in the network. Notice that if an assignment node is never selected it is never checked for arc consistency. Hence the only inconsistencies found are those which are relevant to the assignment nodes which are of interest to the search. Thus the lazy arc consistency technique requires less work to gain a reasonable amount of useful domain information for the GENET network as compared to full arc consistency, although clearly it may gain less information than full arc consistency.

5 Experimental Results

To demonstrate the efficiency of lazy arc consistency on GENET, we have used ECLiPSe Version 3.5.1 and the GNU C Compiler Version 2.6.3 running on SUNOS 5.3, to build prototypes for of GENET, an improved GENET with lazy arc consistency (denoted LAZY) and GENET with full arc consistency preprocessing step (denoted FULL).

The solvers are embedded in a constraint logic programming (CLP) system (see e.g. [6]) which gives a common language for expressing CSP's and allows the handling of disjunctive CSP problems by using search over the constraints of the CSPs. The system builds CSPs incrementally, starting from a CSP with no constraints, the constraints are added one by one. Each CSP built in this process is checked for satisfiability using the stochastic solver (GENET, LAZY or FULL). This checking is essential for efficiency when the problem is disjunctive. When the system finds no solution for an incrementally built CSP, the system backtracks to where there was last a choice of which constraint to add, and then proceeds with the other possibility. For non-disjunctive CSPs this approach has some overhead, so for these problems, we just collect all the constraints and then apply the stochastic solver.

We give results of preliminary experiments comparing the three models: GENET, LAZY and FULL, using a fixed amount of 1000 convergence cycles per solve. Out of interest we also compare the systems versus a traditional propagation-based CLP approach to solving CSPs (using the ECLiPSe fd library). We compare the systems on on a set of CSP's with and without disjunctive constraints. The arbitrary and N-queens problems as examples of CSP's without disjunctive constraints. The hamiltonian path calculation and disjunctive graph coloring are examples of CSP's with disjunctive constraints.

In all the test cases, the CPU time for the different models are the median over 10 successful runs to find a solution. All runs for the different models were terminated if the GENET, LAZY or FULL solver used 1000000 convergence cycles without finding a solution. And we aborted the execution of any ECLiPSe program if it took more than 10 hours. In any case, we use a "—" symbol to mean there were no successful runs and a "?" symbol to denote the execution took > 10 hours.

5.1 Non-disjunctive Binary CSP's

Arbitrary Problem
The arbitrary problem we define defines an ordering of its variables as follows : $X_0 < X_1, X_1 <$

$X_2, \cdots, X_{i-1} < X_i$ where i is the problem size. Problem size is measured in terms of the number of constraints in the problem.

The following table shows comparison of the CPU time for the set of arbitrary problems we tested. The domain size for each problem is either the same as problem size or its double. The table is divided into two halves. The first column of each half is the time in seconds for GENET to find a solution, the second column is for LAZY and the third column is for FULL. The numbers in brackets are the average numbers of values removed by LAZY or FULL.

	dom. size					
prob. size n	$n+1$			$2n$		
	GENET	LAZY	FULL	GENET	LAZY	FULL
10	0.455	0.080 [58.5]	0.270 [110]	0.610	0.340 [33.5]	0.450 [110]
20	37.14	1.995 [283]	5.905 [420]	32.44	10.69 [156]	14.14 [420]
30	304.1	12.78 [643]	41.87 [930]	267.8	123.1 [431]	408.1 [930]
40	1767	57.35 [1199]	161.9 [1640]	1749.76	562.3 [869.5]	1667 [1640]
50	8938	188.7 [2024]	508.9 [2550]	9632	2232 [1528]	5019 [2550]

In both cases of $n+1$ and $2n$, FULL betters GENET, which shows that the arc consistency preprocessing step gains useful information for the GENET network convergence procedure. But LAZY always spends less time than FULL and finds a significant proportion of the arc inconsistent values, showing the efficiency of the lazy arc consistency. Notice as the domain size gets larger the number of arc inconsistencies found by LAZY is reduced since the search never trys some values. The advantages of full arc consistency for GENET is more obvious on small and hard instances of CSP's (such as $n+1$) than larger and easier CSP's (such as $2n$). This is because the advantages gained by the removal of values is traded off by the time spending on consistency checks for the value-pairs as the search space increases on larger CSP's.

N-queens Problem

The N-queens problem is to place N queens onto a $N \times N$ chessboard so that no queens can attack another. A queen can attack another if it is on the same column, row or diagonal as the other queen. It is a standard benchmark for CSPs since the size N can be increased without limit. For this problem, the initial formulation is arc consistent hence neither LAZY nor FULL can improve the convergence behaviour. To investigate the effect of lazy arc consistency on CSP's which are arc consistent, we compare the timings of GENET, LAZY and FULL on the 10- to 50-Queens problems. The following table shows the median CPU time for finding a solution.

No. of Queens	GENET	LAZY	FULL
10	0.145	0.160	0.190 (0.015)
20	2.025	2.025	2.270 (0.220)
30	9.655	11.14	12.53 (1.250)
40	31.58	29.98	34.34 (4.615)
50	72.57	72.11	79.46 (10.78)

The figure in parentheses is the median CPU time for the full arc-consistency preprocessing step. The CPU time for finding a solution for GENET and LAZY do not differ much showing that lazy arc consistency does not take up much computation even in the worst case where it does not gain any extra useful information. This is reflected in the fact that only one extra line in the inner loop of input calculation together with a few testing statements are added in the pseudo-code for lazy arc consistency. The timing for FULL shows that full arc consistency can take up a significant part of time for inefficient computation when arc consistency is unable to gain any useful domain reduction information for the CSP at hand.

No. of Queens	GENET	LAZY	FULL
10	0.280	0.250	0.600 (0.315)
20	3.945	4.015	27.80 (23.42)
30	25.36	24.46	308.9 (280.0)
40	93.33	97.24	1793 (1688)
50	268.0	259.0	7045 (6746)

The second table shows the same benchmarks when the CSP is checked for satisfiability after each constraint is added; thus treating the problem as if it were disjunctive. Now FULL is totally un-competitive since instead of a single arc-consistency check it performs one per constraint and this dominates the computation time.

5.2 Disjunctive Binary CSP's

Hamiltonian Path Problems
Given a graph G of n vertices, the hamiltonian path problem is to find the hamiltonian path of G, in which *each* of n vertices in G is visited *once and only once* [10]. The hamiltonian path problem is a practical CSP which is very similar to the route planning problems faced by sales departments. It can be regarded as an non-optimizing version of the well-known traveling salesman problem, in which the salesman does not need to return to the original city.

Problems	ECLiPSe	GENET	LAZY	FULL
10	0.220	4.195	0.460 [54.5]	0.855 [90]
20	339.0	58.27	31.75 [119.5]	63.99 [380]
30	?	5493	2687 [469]	3084 [870]

The table above show the performance of ECLiPSe and the three models of GENET on hamiltonian path problems using 100 convergence cycles per solve for the first two cases and 1000 convergence cycles per solve for the last case. Statistics are CPU seconds execution time. The average number of values removed by arc consistency are shown in square brackets. The first two problems are coded from some interesting real-life examples in graph theory [10]. And the last one is a modified example obtained from [7].

In general, the stochastic solvers GENET, LAZY and FULL outperform ECLiPSe on all but the smallest example. This is because the search space for GENET, LAZY and FULL is much smaller than for the ECLiPSe because they determine unsatisfiability much earlier. This is one of the motivations for using stochastic solvers for disjunctive CSPs.

For these problems, the improved solvers LAZY and FULL, in general, perform better than the original GENET. This shows the arc consistency actually finds useful domain reduction information to prune the search space. While LAZY is clearly the best solver in terms of execution time – it does not find as many inconsistent values as FULL. This is to be expected, but it show the targeted nature of the inconsistencies found by LAZY, only those involved, in the search.

Disjunctive Graph-coloring Problems
The disjunctive graph-coloring problem is to color a graph, possibly non-planar, with a number of hyper-arcs connecting nodes. A hyper-arc $\{(i_1, j_1), (i_2, j_2)\}$ specifies that at least one of the pairs of nodes (i_k, j_k) must be colored differently. The disjunctive graph-coloring problem has wide applicability in time-tabling, scheduling and production planning. It is another example of a problem whose original statement is arc consistent.

Graph		CPU time			
Nodes	Colors	ECLiPSe	GENET	LAZY	FULL
10	3	41.22(6428)	1.580(4)	1.820(4)	1.580(4)
20	4	1514(65535)	0.690(1)	0.770(1)	0.740(1)
30	5	0.090(0)	0.115(0)	0.120(0)	0.220(0)
40	5	0.120(0)	0.145(0)	0.155(0)	0.270(0)
50	5	0.180(0)	0.200(0)	0.225(0)	0.390(0)

The table above show results for ECLiPSe and the three different models of GENET on a set of small-sized disjunctive graph-coloring problems. The numbers in parentheses are the number of backtracks, or choices made in finding a first solution to the disjunctive problem. Note that ECLiPSe introduces many backtracks in a separate labelling stage not performed by the stochastic solvers. In general, the approach where the stochastic solver controls the search (GENET, LAZY and FULL) out-performs ECLiPSe on the hard instances of the CSP's when the problem requires backtracking. This is because they do much less backtracking since they do not require a backtracking enumeration search.

This is an example when no useful domain reduction information is discovered by LAZY or FULL since the original problem is arc consistent. Hence neither of them improve on GENET, and suffer some overhead. Interestingly, in the first two examples where a large number of convergence cycles are required compared to the number of constraint additions, the overhead of LAZY (which is per convergence cycle) is greater than FULL (which is per constraint addition). For the easier problems LAZY outperforms FULL by a considerable margin.

6 Concluding remarks

In this paper, we consider a relaxed form of local consistency, lazy arc consistency. Adding lazy arc consistency to GENET is of considerable benefit, because it avoids the expense of full arc consistency and gains most of the benefit. In particular in cases where the problem is dynamically changing, for example in a CLP system handling disjunctive CSP's, the benefit is significant. This is demonstrated in our benchmarks for hamiltonian path problems. Even for non-disjunctive CSP's such as N-queens, lazy arc consistency does not introduce significant overhead where full arc consistency can. We are now considering how to generalize the lazy arc consistency for general CSP's, so that the same techniques can be applied to handle binary and non-binary constraints in extended versions of GENET such as E-GENET. Also, the integration of lazy arc consistency into another stochastic methods is an interesting future direction.

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 *Proceedings of AAAI'94*, 1994.

[3] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Procs. of the European Conf. on Art. Int.*, 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] J.J. Hopfield and D. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[6] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, Volume 19 / 20, May 1994.

[7] J.H.M. Lee, H.F.Leung and H.W. Won. Extending GENET for Non-Binary CSP's. In *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence*, 338–343, November 1995.

[8] J.H.M. Lee and V.W.L. Tam. A Framework for Integrating Artificial Neural Networks and Logic Programming *International Journal on Artificial Intelligence Tools* 4(1&2), 3–32, June, 1995.

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

[10] Ronald E. Prather *Discrete Mathematical Structures for Computer Science.* Houghton Mifflin,1976.

[11] 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, 1995.

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

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

[14] 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*, 295–299, 1991.

[15] L.C. Wu and C.Y. Tang. Solving the satisfiability problem by using randomized approach. In *Information Processing Letters* 41, 295–299, North-Holland, 1992.