The Journal of Functional and Logic Programming

The MIT Press

Volume 1998 Article 2
20 January, 1998

ISSN 1080–5230. MIT Press Journals, Five Cambridge Center, Cambridge, MA 02142-1493, USA; (617)253-2889; *journals-orders@mit.edu, journals-info@mit.edu*. Published one article at a time in LaTeX source form on the Internet. Pagination varies from copy to copy. For more information and other articles see:

- *http://www.cs.tu-berlin.de/journal/jflp/*
- *http://mitpress.mit.edu/JFLP/*
- *gopher.mit.edu*
- *ftp://mitpress.mit.edu/pub/JFLP*

The Journal of Functional and Logic Programming is a peer-reviewed and electronically published scholarly journal that covers a broad scope of topics from functional and logic programming. In particular, it focuses on the integration of the functional and the logic paradigms as well as their common foundations.

Semantics for using Stochastic Constraint Solvers in Constraint Logic Programming

Peter Stuckey Vincent Tam

20 January, 1998

Abstract

This paper proposes a number of models for integrating finite-domain stochastic constraint solvers into constraint logic programming systems to solve constraint-satisfaction problems efficiently. Stochastic solvers can solve hard constraint-satisfaction problems very efficiently, and constraint logic programming allows heuristics and problem breakdown to be encoded in the same language as the constraints; hence their combination is attractive. Unfortunately, there is a mismatch between the kind of information a stochastic solver provides and that which a constraint logic programming system requires. We study the semantic properties of the various models of constraint logic programming systems that make use of stochastic solvers, and give soundness and completeness results for their use. We describe an example system we have implemented using a modified neural network simulator, GENET, as a constraint solver. We briefly compare the efficiency of these models against the propagation-based solver approaches that are typically used in constraint logic programming.

1 Introduction

This paper proposes a framework for using stochastic constraint solvers within constraint logic programming (CLP), employing a number of models for integration. Our motivation for using such solvers is to solve constraint-satisfaction problems (CSPs) more efficiently. In the paper we concentrate on this case, although the theoretical results are applicable to any stochastic constraint solver and constraint domain.

A CSP (see, e.g., [Tsa93]) involves a finite set of variables, each of which has a finite domain (set) of possible values, and a constraint formula that limits the combination of values for a subset of variables. The task is to assign values to the variables so that the constraint formula is satisfied.

Although CSPs occur in a large number of applications, such as computer vision, planning, resource allocation, scheduling, and temporal reasoning, CSPs are, in general, NP-complete. Thus, a general algorithm designed to solve any CSP will necessarily require exponential time[1] in the worst case. Constraint logic programming systems have been successfully used to tackle a number of industrial CSP applications, such as car sequencing [DSH88], disjunctive scheduling [Hen89], and firmware design [DSH90]. Stochastic search methods have also had remarkable success in solving industrial CSPs [WT92a] and constraint-satisfaction optimization problems (CSOPs) [AK89, HT85].

Constraint logic programming systems use a constraint solver to direct a search for an answer to a goal. When the constraint solver determines that the constraints collected on some derivation path are unsatisfiable, the CLP system backtracks and tries a different derivation path. Thus the solver's key behavior is its correct determination of unsatisfiability. Typically, constraint solvers are incomplete, that is, they do not answer whether every constraint is satisfiable or unsatisfiable. For example, many real-number constraint solvers treat nonlinear constraints incompletely, and integer constraint solvers are also typically incomplete. The major requirement of the incomplete solver in a CLP system is determining unsatisfiability; that is, as often as possible, the solver returns *false* if a constraint is unsatisfiable.

Most constraint solvers in a CLP system that are used to solve CSPs employ the technique of *constraint propagation* to solve the set of constraints. Each constraint is used to restrict the possible values that variables within it can take. Restrictions caused by one constraint may propagate further restrictions from other constraints, and the propagation continues until no further restrictions are made. Constraint propagation is algorithmic in nature. To guarantee the finding of a solution, these solvers are augmented with some form of enumerative search. When the CSP is tight,[2] this search may be very costly.

Many of the methods traditionally used in solving CSPs are not propagation-

[1] Assuming $P \neq NP$.

[2] A *tight* CSP has few solutions over a large search space.

based. A major class of CSP-solving techniques are stochastic methods; for example, simulated annealing, neural networks, and evolutionary algorithms. These methods are designed to find solutions to CSPs using a relaxation-based search. The methods are, in general, not guaranteed to find a solution, and they typically involve a resource bound on the search that determines how long the method will take and the likelihood of finding a solution. Such solvers do not usually determine when a constraint is unsatisfiable, but these kinds of solvers can be considerably more efficient than propagation-based solvers on large or hard instances of CSPs. The problem we examine is how such solvers can be used efficiently in CLP systems.

Earlier approaches that considered incorporating stochastic solvers into CLP systems were restricted to one of the models we discuss. Lee and Tam [LT95] required the program to execute only a single derivation, so that backtracking could never occur. At the end of the single derivation, the stochastic solver (the GENET algorithm discussed later) determines a solution. Both Illera and Ortiz [IO95] and Kok et al. [KMMR96] used genetic algorithms to search for a good solution to a constraint that was collected by a CLP system; hence they used a stochastic method to tackle a CSOP. All of these papers show how using stochastic methods instead of enumeration methods traditional to CLP can lead to substantial benefits. None of these works used the stochastic constraint solver to control the search for a successful derivation; rather, it was used only as a solution finder, hence most of the issues dealt with in this paper do not arise. Besides, there are related works developed in artificial intelligence that incorporate constraint-reasoning techniques into stochastic solvers such as genetic algorithms (GA) to improve performance in solving CSPs. Bowen and Dozier [BD95] considered a hybrid GA incorporating arc revision to determine when to stop with failure in case the CSP is unsatisfiable. Riff Rojac [Roj96] studied how to embed constraint propagation into GAs for solving CSPs. Our works focus on a general theoretical framework for integrating stochastic constraint solvers into constraint logic programming systems, rather than only addressing an improvement in the stochastic solvers.

This paper is organized as follows. Section 2 briefly introduces some preliminaries for subsequent discussion. Section 3 describes various models for how a stochastic solver can be used within a CLP system, while Section 4 gives soundness and completeness results for these various models. In Section 5, we briefly describe GENET [WT92a, DTWZ94], a probabilistic artificial neural network (ANN) used as the constraint solver. We use

GENET to demonstrate the feasibility of our approach. We describe how we adapt the original model to support efficient incremental execution with backtracking on constraints. Section 6 gives our experimental results, and in Section 7 we conclude.

2 Preliminaries

Here we briefly introduce constraint logic programming. For more details, see [JM94]. A *constraint domain* $\mathcal{A}$ is a structure defining the language and meaning of the primitive constraints. A *primitive constraint* is of the form $r(t_1, \ldots, t_n)$, where r is a relation defined by $\mathcal{A}$, and $t_1, \ldots, t_n$ are terms for $\mathcal{A}$. For example, $X > Y + 2$ is a primitive constraint over the domain of integer constraints. We assume that the constraint domain includes the primitive constraint $=$, which is always interpreted as identity. A *constraint* is of the form $c_1 \wedge \ldots \wedge c_n$, where $n \geq 0$ and $c_1, \ldots, c_n$ are primitive constraints. When $n = 0$, we usually write the constraint as *true*.

We assume familiarity with first-order logic (see, for example, [Men87]). In particular, a *valuation* θ for a set S of variables is an assignment of values from the domain of $\mathcal{A}$ to the variables in S. Suppose $S = \{X_1, \ldots, X_n\}$; then θ may be written $\{X_1 \mapsto a_1, \ldots, X_n \mapsto a_n\}$, indicating that each X_i is assigned the value a_i. Let $vars(F)$ denote the set of free variables occurring in a formula F. Let $\bar{\exists}_V F$ represent the formula $\exists w_1 \ldots \exists w_k F$, where $\{w_1, \ldots, w_k\} = vars(F) - V$. Let $\tilde{\exists} F$ represent the formula $\bar{\exists}_\emptyset F$, the existential closure of F. If θ is a valuation for $S \supseteq vars(c)$, then it is a *solution* of c if the replacement of each variable by its value (as given by θ) yields a true statement; that is, $\mathcal{A} \models_\theta c$.

A constraint c is *satisfiable* if there exists a solution θ of c. Otherwise, it is *unsatisfiable*. A *constraint solver*, *solv*, for a constraint domain $\mathcal{A}$ takes as input any constraint c in $\mathcal{A}$ and returns *true*, *false*, or *unknown*. A solver *solv* is *correct* if whenever $solv(c)$ returns *true*, the constraint c must be satisfiable; and whenever $solv(c)$ returns *false*, the constraint c must be unsatisfiable. A constraint solver is *complete* if it is correct and always returns either *true* or *false* (never *unknown*).

An *atom*, A, is of the form $p(t_1, \ldots, t_n)$, where p is an n-ary *predicate symbol* and $t_1, \ldots, t_n$ are terms from the constraint domain. A *literal* is either an atom or a primitive constraint. A *goal* G is a sequence of literals $L_1, \ldots, L_n$. An empty sequence is denoted $\Box$. A *rule*, R, is of the form

A :- B, where A is an atom and B is a goal. A is called the *head* of R, and B is called the *body* of R. A *(constraint logic) program*, P, is a set of rules. We will sometimes consider a goal as a conjunction of atoms and primitive constraints.

Execution proceeds by mapping one state to another. A *state* is a pair written $\langle G \mid c \rangle$, where G is a goal and c is a constraint. A *derivation step* from $\langle G_0 \mid c_0 \rangle$ to $\langle G_1 \mid c_1 \rangle$, written $\langle G_0 \mid c_0 \rangle \Rightarrow \langle G_1 \mid c_1 \rangle$, is defined as follows: Let G_0 be of the form

$$L_1, \ldots, L_{i-1}, L_i, L_{i+1}, \ldots, L_m$$

where L_j, $1 \leq j \leq m$ are literals and L_i is the *selected literal*. There are two cases:

1. L_i is a primitive constraint. Then G_1 is $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$, and c_1 is $c_0 \wedge L_i$. If $solv(c_1) = false$, the state is *failed*.

2. L_i is an atom $p(t_1, \ldots, t_n)$. Let R be a renamed version of a rule in P, sharing no variables with $\langle G_0 \mid c_0 \rangle$ of the form

   $$p(s_1, \ldots, s_n) \texttt{:-} B_1, \ldots, B_k$$

   Then c_1 is c_0, and G_1 is

   $$L_1, \ldots, L_{i-1}, s_1 = t_1, \ldots, s_n = t_n, B_1, \ldots, B_k, L_{i+1}, \ldots, L_m$$

A derivation $\langle G_0 \mid c_0 \rangle \Rightarrow \langle G_1 \mid c_1 \rangle \Rightarrow \ldots$ is a sequence of derivation steps $\langle G_i \mid c_i \rangle \Rightarrow \langle G_{i+1} \mid c_{i+1} \rangle$, where at each stage $i \geq 0$ the state $\langle G_i \mid c_i \rangle$ is not failed, and the selected literal is given by some *selection strategy*. A derivation terminates either by reaching a failed state or an empty goal.

A derivation is *successful* if it is finite with last state $\langle \Box \mid c_n \rangle$, where $solv(c_n) \neq false$. A derivation is *finitely failed* if it is finite and at the last state is failed ($solv(c_n) = false$). Thus a derivation only terminates by success or by the solver detecting an unsatisfiable constraint; otherwise, the derivation is infinite. A derivation is *fair* if it is finitely failed or each literal appearing in the derivation is eventually selected. A selection strategy is fair if it always produces fair derivations.

A derivation for a goal G is a derivation from state $\langle G \mid true \rangle$. If

$$\langle G \mid true \rangle \Rightarrow \ldots \Rightarrow \langle \Box \mid c_n \rangle$$

is a successful derivation for G, then $\bar{\exists}_{vars(G)} c_n$ is an *answer* to the goal G.

It will later be useful to refer to derivations in a manner where the last step is always a derivation step of type 1. Let $\langle G_0 \mid c_0 \rangle \overset{(1)}{\Rightarrow} \langle G_1 \mid c_1 \rangle$ represent a sequence of any number of type 2 derivation steps followed by a type 1 derivation step. Clearly, for a derivation of the form $\langle G_0 \mid c_0 \rangle \overset{(1)}{\Rightarrow} \langle G_1 \mid c_1 \rangle \overset{(1)}{\Rightarrow} \ldots \overset{(1)}{\Rightarrow} \langle G_n \mid c_n \rangle$, the constraint solver is invoked on each of the constraints $c_1, c_2, \ldots, c_n$.

Typically, CLP systems execute by starting from an initial state $\langle G \mid \mathit{true} \rangle$ and building a derivation. Whenever a derivation step of type 2 is made, there may be a choice of which rule in P to use in the step. If this is the case, the system sets a *choicepoint* on the state $\langle G_0 \mid c_0 \rangle$ before the derivation step. After a derivation step of type 1 is made where $\mathit{solv}(c_1) = \mathit{false}$, then the execution *backtracks* to the state of the last choicepoint and begins a derivation from that point using an untried rule. If this is the last untried rule, then the choicepoint is removed.

The constraint solver typically used in a CLP system to solve CSPs are so-called *propagation-based solvers* (for example, see [Hen89]). Each variable is associated with a domain of possible values, and consistency techniques such as generalized arc-consistency are used to remove values that cannot be part of a solution from the domains of variables. A common technique, called *bounds propagation*, is to only ensure that the upper and lower bounds of each variable are possibly consistent. Since the result of propagation does not, in general, guarantee the existence of a solution, propagation-based solvers usually return either *false* or *unknown*. Typically, they return *true* only when every variable is explicitly constrained to take a unique value. For our examples, we denote the range of integers from l to u inclusive as $l..u$, the empty range as $\emptyset$, and $X :: r$ to associate with variable X its current range of values r.

Example 1 *Consider the following simple CLP program executing in a constraint domain of the integers from* 0 *to* 4*:*

```
p(X, Y, Z) :- X + Y + Z = 4, r(X, Y, Z).
r(X, Y, Z) :- X < 0.
r(X, Y, Z) :- X + Y + Z = 3.
r(X, Y, Z) :- X = Z.
```

The derivation tree in Figure 1 shows all possible derivations for the goal $p(X, Y, Z)$*, and is somewhat simplified to remove extra variables introduced*

by renaming. The selected literal at each stage is underlined. If a bounds-

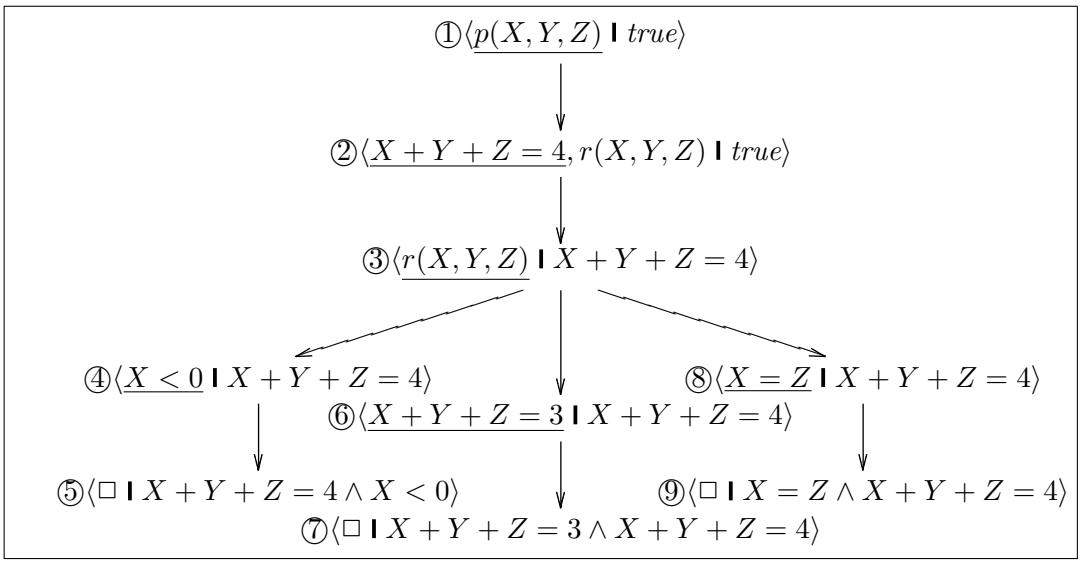

Figure 1: A derivation tree for a simple program

propagation solver were used in determining the derivations, then it would be invoked on the constraints in ③, ⑤, ⑦, and ⑨, and would act as follows:

State	*Truth*	*Bounds*
③	*unknown*	$\{X :: 0..4, Y :: 0..4, Z :: 0..4\}$
⑤	*false*	$\{X :: \emptyset, Y :: 0..4, Z :: 0..4\}$
⑦	*unknown*	$\{X :: 0..3, Y :: 0..3, Z :: 0..3\}$
⑨	*unknown*	$\{X :: 0..4, Y :: 0..4, Z :: 0..4\}$

For example, it does not detect that the constraint at ⑦ is false, and it does not guarantee an answer at ⑨. To fix these weaknesses, the propagation solver is usually executed with a labeling predicate that will set each variable to some value. For example, the following program and goal would add labeling to the above program:

```
value(0).
value(1).
value(2).
value(3).
```

```
value(4).
goal(X,Y,Z) :- p(X,Y,Z), value(X), value(Y), value(Z).
```

Executing this goal would create a derivation tree similar in shape to that in Figure 1, but at states ⑦ and ⑨, execution would continue by trying different values for X, then Y, then Z. The derivation tree below ⑦ would all lead to failed derivations, while underneath ⑨ the first successful derivation would be lead to a state

$$\langle \Box \mid X + Y + Z = 4 \wedge X = Z \wedge X = 0 \wedge Y = 4 \wedge Z = 0 \rangle$$

3 Using a Stochastic Solver

Consider the following simple example of a stochastic constraint solver for Boolean CNF constraints (based on an algorithm by Wu and Tang [WT92b]).

BOOL_SOLVE(F, ϵ)
Let m be the number of clauses in F
$n := \ln \epsilon / \ln(1 - (1 - 1/m)^m)$
for $i := 1$ to n
    generate a random truth assignment θ
    if $F\theta$ is true return *true*
endfor
return *unknown*

This algorithm is a randomized polynomial time algorithm for solving a Boolean constraint F. The second argument, ϵ, gives a bound on its incompleteness. Applied to a satisfiable constraint F, the algorithm[3] will return *true* with probability $1 - \epsilon$ (this is quite a deep result of [WT92b]). So, ϵ is the probability that the solver returns *unknown* when the constraint F is satisfiable. All known complete solvers for this problem are exponential while this algorithm is polynomial. But this solver never returns *false*, because it can never determine that constraint F is unsatisfiable. Hence it seems difficult to use it in the context of a CLP system where we need failure to prune the search space.

Because stochastic solvers in general cannot determine unsatisfiability, when applied to an unsatisfiable constraint they will execute forever unless

[3] Assuming no individual clause is trivially *false* or *true*.

some limit is imposed upon them. Therefore they usually have a resource limit, for example, the iteration limit n in BOOL_SOLVE. Thus, we define a stochastic constraint solver as a solver that takes two arguments: a constraint, and a resource bound.

Definition 1 *A stochastic solver ssolv takes a constraint c and resource limit n, and either returns:*

true*, the amount of resources used to find a solution, and the solution itself. For example, $ssolv(c, n) = (true, m, \theta)$ where $m \leq n$, or*

unknown*, the resource limit, and a possibly meaningless valuation.*[4] *For example, $ssolv(c, n) = (unknown, n, \theta)$.*

We define three functions for extracting parts of the stochastic solver's answer:

$$ans((x_1, x_2, x_3)) = x_1$$
$$res((x_1, x_2, x_3)) = x_2$$
$$val((x_1, x_2, x_3)) = x_3$$

Stochastic solvers are generally randomized algorithms, hence they define a probability of finding a solution for some satisfiable constraint, c. We assume that if c is a satisfiable constraint, then $Pr(ans(ssolv(c, n)) = true) > 0$ if $n > 0$; that is, the solver always has a chance of finding a solution, given some resources. We assume that if c is an unsatisfiable constraint, $Pr(ans(ssolv(c, n)) = unknown) = 1$ whatever resource bound n is used, meaning the solver is correct and never returns *false*.[5] We also assume for technical convenience that $Pr(ans(ssolv(c, -1)) = unknown) = 1$. That is, given negative resources, the solver always returns *unknown*.

Because of the mismatch between the type of information that a stochastic solver provides and that required by a CLP system, several useful models arise for using a stochastic solver within a CLP system.

[4] In some cases, this valuation can have meaning; for example, in using the GENET solver discussed later, it could be the valuation that is closest to a solution.

[5] It is not difficult to extend the presentation here to handle solvers that return *false*, but it is orthogonal to the problems we tackle.

3.1 Model A: The Stochastic Solver as a Constraint Solver

The most obvious combination is to use the stochastic constraint solver to control the derivations. Clearly, because a stochastic constraint solver only returns *true* or *unknown*, it will never cause the execution to backtrack if it is used directly. Therefore, at some stage, we must assume that its failure to find a solution indeed indicates that the constraint is unsatisfiable. Thus we must sometimes treat the answer *unknown* as *false*. This will lead to incompleteness, but there are already a number of sources of incompleteness for CLP systems, such as depth-first search and unfair literal selection, so we should be willing to accept this approach if the resulting incompleteness is not too great.

More practically, even though the combination of propagation and enumeration is theoretically correct and complete, the actual behavior of most finite-domain constraint solving systems is to return *false* after a certain delay. This is because we cannot afford arbitrarily long consistency checks in practice. As a result, resource exhaustion may occur, whereby a (possibly consistent) constraint store is considered inconsistent after some time. Obviously, using stochastic solvers does not add any level of incompleteness with respect to this practical use of propagation-based finite-domain solvers.

Example 2 *Consider the program and goal of Example 1. Using a stochastic solver, we might expect the following answers for each state where it is invoked:*

State	*Truth*	*Possible Valuation*
③	*true*	$\{X \mapsto 1, Y \mapsto 1, Z \mapsto 2\}$
⑤	*unknown*	*meaningless* $\{X \mapsto 1, Y \mapsto 1, Z \mapsto 2\}$
⑦	*unknown*	*meaningless* $\{X \mapsto 1, Y \mapsto 1, Z \mapsto 2\}$
⑨	*true*	$\{X \mapsto 1, Y \mapsto 2, Z \mapsto 1\}$

Note that the valuations corresponding to unknown answers are meaningless. The unknown answers are treated as false, hence at ⑤ and ⑦, execution backtracks.

3.2 Model B: The Stochastic Constraint Solver as a Solution Finder at the End of a Derivation

The usual approach to solving CSPs in a CLP system employs a propagation solver. Propagation solvers are weak, and do not detect many cases of unsatisfiability. Usually, propagation solvers can only answer *false* or *unknown*, however, they can answer *true* when there is a unique value for each of the variables in the constraint (that is, the solver can check that a valuation is a solution). Because of this CLP, programs for solving CSPs typically have the following form:

```
goal(Vs) :- setupvars(Vs), constrain(Vs), labeling(Vs)
```

First the variables are declared, and then constrained. Finally, the labeling predicate performs an enumerative search for a valuation by setting each variable in turn to each of its possible values.

The enumerative search is usually where most of the computation time of the program is spent. Enumerative searches, even with clever heuristics to order the search, can sometimes perform very poorly; hence it is worth considering to use a stochastic solver as a valuation finder. For Model B, the constraint check in a derivation is performed by a propagation solver during each derivation step of type 1, and then whenever a successful derivation is found, e.g., when $\langle G_0 \mid true \rangle \Rightarrow \ldots \Rightarrow \langle \Box \mid c_n \rangle$, the stochastic solver is invoked on c_n to find a solution. The relaxation-based search of the stochastic solver replaces the enumerative search defined by labeling. In this way, the labeling part of the program is not required.

In addition, the propagation solver can communicate extra constraint information it derives from consistency techniques to the stochastic constraint solver to improve its behavior. The obvious example involves communicating the domains of variables. Whenever the propagation solver can reduce the domain of a variable, it can inform the stochastic solver of this reduced domain. This reduces the search space of the stochastic solver.

Example 3 *Consider the program and goal of Example 1, using a stochastic solver as a solution finder and the propagation solver to control the derivation. The propagation solver acts as in Example 1, and the stochastic solver is invoked at ⑦ and ⑨, resulting in:*

State	*Truth*	*Possible Valuation*
⑦	*unknown*	*meaningless* $\{X \mapsto 1, Y \mapsto 1, Z \mapsto 2\}$
⑨	*true*	$\{X \mapsto 1, Y \mapsto 0, Z \mapsto 2\}$

Note that the stochastic solver is not invoked at ⑤, because the constraint is already known to be false. At ⑦, the stochastic solver is invoked with the extra information that X, Y, and Z lie within the range 0..3*, which comes from the propagation solver. No enumeration of valuations is required.*

3.3 Model C: The Stochastic Solver Augmented by a Propagation Solver

This model is analogous to Model A. At each derivation step, the constraint is checked by first the propagation solver and then the stochastic solver. The role of the propagation solver is to detect unsatisfiability (which, when detectable, can be found much faster than with the stochastic solver). Again, the propagation solver can communicate extra constraint information to the stochastic solver (as in Model B, above).

Example 4 *Consider the program and goal of Example 1 using a stochastic solver together with the propagation solver to control the derivation. The result for each solver is shown below:*

State	*Solver*	*Truth*	*Bounds or Valuation*
③	*prop*	*unknown*	$\{X :: 0..4, Y :: 0..4, Z :: 0..4\}$
③	*stoch*	*true*	$\{X \mapsto 1, Y \mapsto 1, Z \mapsto 2\}$
⑤	*prop*	*false*	$\{X :: \emptyset, Y :: 0..4, Z :: 0..4\}$
⑤	*stoch*	*not invoked*	
⑦	*prop*	*unknown*	$\{X :: 0..3, Y :: 0..3, Z :: 0..3\}$
⑦	*stoch*	*unknown*	*meaningless* $\{X \mapsto 1, Y \mapsto 1, Z \mapsto 2\}$
⑨	*prop*	*unknown*	$\{X :: 0..4, Y :: 0..4, Z :: 0..4\}$
⑨	*stoch*	*true*	$\{X \mapsto 1, Y \mapsto 0, Z \mapsto 2\}$

Note that again the stochastic solver is not invoked at ⑤, and no enumeration of valuations is required; when invoked at ⑦, the stochastic solver has extra constraint information from the propagation solver.

3.4 Model D: The Stochastic Solver Used in Parallel with a Propagation Solver

The stochastic solver is used to give information about success—answering either *true* or *unknown*. The propagation solver is used to give information about failure, answering *false* or *unknown*. Constraints are sent to both

solvers independently by the run-time engine. When the stochastic solver answers *true*, the run-time engine continues the derivation immediately, without waiting for the propagation solver. When the propagation solver answers $false$, the run-time engine backtracks immediately, without waiting for the stochastic solver. Otherwise, the run-time engine waits until both answer *unknown* before the execution continues. A diagram of the interaction between the solvers and the run-time engine is shown in Figure 2.

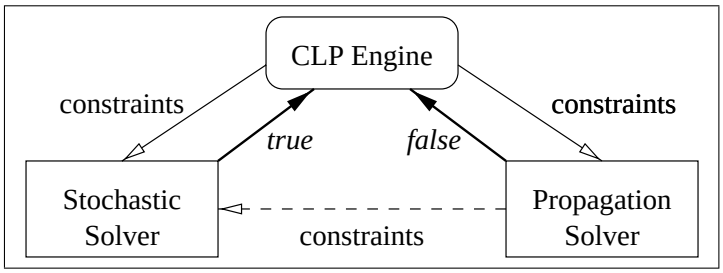

Figure 2: A stochastic solver in parallel with a propagation solver

The combination of solvers working asynchronously gives advantages to both: first, the stochastic solver does not need to calculate every failure (as in Model C), but additionally, the propagation solver does not need to complete propagation of constraints for every satisfiable constraint store. Consider executing a goal $\langle c_1, c_2, c_3, c_4 \mid c\rangle$ where $c \wedge c_1 \wedge c_2 \wedge c_3$ is satisfiable, but $c \wedge c_1 \wedge c_2 \wedge c_3 \wedge c_4$ is not. Then suppose the stochastic solver quickly finds solutions for $c \wedge c_1$, $c \wedge c_1 \wedge c_2$, and $c \wedge c_1 \wedge c_2 \wedge c_3$ (for example, if its solution for c is also a solution for these constraints). Before the propagation solver can finish propagation for $c \wedge c_1$, the execution has continued adding the constraints c_2, c_3, and c_4. The propagation solver may then be able to detect the unsatisfiability of $c \wedge c_1 \wedge c_2 \wedge c_3 \wedge c_4$ before it completes the propagation required for solving $c \wedge c_1 \wedge c_2 \wedge c_3$. Overall, the failure of the derivation will be detected faster.

3.5 Usage Strategies

When a stochastic constraint solver is required to communicate with the run-time engine, as in Models A, C, and D, we need a way of specifying at each derivation step of type 1 how many resources should be used. Additionally, we need to specify whether at this point we will treat an *unknown* answer

as $false$. Similarly, for Model B, we need a way of specifying how many resources are used at the end of each derivation. This is the role of a usage strategy.

A *usage strategy* U is a function on derivations that determines how a stochastic solver is to be used. Given a derivation D ending in a derivation step of type 1, $\langle G \mid true\rangle \stackrel{(1)}{\Rightarrow}^* \langle G_n \mid c_n\rangle$, the usage strategy $U(D)$ gives a pair defining:

1. whether an *unknown* result is considered *unknown* or *false*, and
2. a resource limit R to be used in solving c_n.

Define $ans((x_1, x_2)) = x_1$ and $res((x_1, x_2)) = x_2$. Many usage strategies are possible, for example:

- *fixed limit*—the simplest usage strategy, denoted $U_{fl(R)}$.

  $$U_{fl(R)}(D) = (false, R)$$

  where in each step, R resources are allocated and failure to find a solution with R resources is considered to be "proof" that the constraint is unsatisfiable.

- *derivation limit*—a more complicated strategy, denoted $U_{dl(R)}$.

  $$\begin{aligned} U_{dl(R)}(\langle G \mid true\rangle) &= (false, R), and \\ U_{dl(R)}(D \stackrel{(1)}{\Rightarrow} \langle G_{n+1} \mid c_{n+1}\rangle) &= (false, m) \end{aligned}$$

  where $U_{dl(R)}(D) = (false, n)$, and $m = n - res(ssolv(c_n, n))$. This encodes a strategy where resources are available for use in each derivation R. After each step in the derivation, the resources used $res(ssolv(c_n, n))$ are subtracted from the remaining resources.

- *lookahead limit*—a more complicated usage strategy, denoted $U_{ll(R1,R2)}$. This strategy yields $R2$ resources to be used on each derivation before we assume that the result *unknown* means *false*; however, at each individual constraint-solving step, only $R1$ resources may be used to find a solution. If, after using $R1$ resources, satisfiability has not been shown,

an *unknown* result is treated as *unknown*, until the total resource limit $R2$ is reached:

$$\begin{aligned} U_{ll(R1,R2)}(\langle G \mid true\rangle) &= (unknown, R1) \\ U_{ll(R1,R2)}(D \Rightarrow \langle G_{n+1}, c_{n+1}\rangle) &= (v, m) \\ m &= min(R1, Rem) \\ v &= false \text{ if } m = Rem \text{ else } unknown \\ \text{where } Rem &= R2 - \sum_{i=1}^{n} res(ssolv(c_i, k_i)) \\ \text{and } k_i &= res(U_{ll(R1,R2)}(\langle G \mid true\rangle \stackrel{*}{\Rightarrow} \langle G_i, c_i\rangle)) \end{aligned}$$

Many other strategies are possible, for example, where the fixed limit varies with the length of the derivation. Note that Model B can be seen as a particular case of Model C, given a usage strategy U_b for Model B (which clearly only assigns resources for successful derivations), then the following usage strategy for Model C yields equivalent derivations:

$$\begin{aligned} U_{c2b}(\langle G \mid true\rangle \stackrel{(1)}{\Rightarrow}^* \langle \Box \mid c\rangle) &= U_b(\langle G \mid true\rangle \stackrel{(1)}{\Rightarrow}^* \langle \Box \mid c\rangle) \\ U_{c2b}(\langle G \mid true\rangle \stackrel{(1)}{\Rightarrow}^* \langle G_n \mid c_n\rangle) &= (unknown, -1) \text{ when } G_n \neq \Box \end{aligned}$$

3.6 The Formal Models

Given that we have defined a usage strategy U, we can more formally define each of the above models. Let *ssolv* be the stochastic solver, and *psolv* be the propagation solver. The solver used to solve c_n for the last derivation step in the derivation

$$D = \langle G \mid true\rangle \stackrel{(1)}{\Rightarrow}^* \langle G_n \mid c_n\rangle$$

is given by the following models:

Model A

$$\begin{aligned} asolv(c_n) &= true \text{ if } ans(ssolv(c_n, res(U(D)))) = true \\ &= ans(U(D)) \text{ otherwise} \end{aligned}$$

Model B

$$\begin{aligned} bsolv(c_n) &= psolv(c_n) \text{ when } G_n \neq \Box \\ &= psolv(c_n) \cap asolv(c_n) \text{ otherwise} \end{aligned}$$

Model C

$$csolv(c_n) = psolv(c_n) \cap asolv(c_n)$$

Model D

$$dsolv(c_n) = psolv(c_n) \cap ans(ssolv(c_n, res(U(D))))$$

The $\cap$ operation is a form of conjunction for four-valued logic, as shown in the table:

$\cap$	*true*	*false*	*unknown*
true	*true*	$\perp$	*true*
false	$\perp$	*false*	*false*
unknown	*true*	*false*	*unknown*

The $\perp$ is a contradiction that can only happen in practice when *psolv* answers *true* and *asolv* answers *false*. For example, using a fixed-limit strategy in Model B or C, the *asolv* may answer *false* for a satisfiable constraint c_n owing to resource exhaustion, while the *psolv* answers *true* when there is only one value remaining in the domain of each variable. For the correctness of iterative deepening for Model C, it is treated as *false*. Note that *asolv*, *bsolv*, and *csolv* are, in general, incorrect solvers.

4 Semantics

When using a stochastic solver, we sometimes assume an *unknown* result is $false$; therefore, the standard theoretical results for the CLP scheme [JL87] are not applicable. In this section we show what theoretical results hold for the use of the various models.

4.1 Success

Regardless of which model and usage strategy U is used, we have the following soundness result for successful derivations.

Theorem 1 *For Models A, B, C, or D, if* $\langle G \mathbin{\mathsf{I}} true\rangle \Rightarrow^* \langle \Box \mathbin{\mathsf{I}} c\rangle$ *is a derivation achieved using stochastic solver ssolv and usage strategy U, then* $P, \mathcal{A} \models c \rightarrow G$.

Proof of Theorem 1 We prove that for a derivation $\langle G_0 \mathbin{\mathsf{I}} c_0\rangle \Rightarrow^n \langle G_n \mathbin{\mathsf{I}} c_n\rangle$ of length n that

$$P, \mathcal{A} \models G_n \wedge c_n \rightarrow G_0 \wedge c_0$$

The essential result is that for a single derivation step $\langle G_j \mathbin{\mathsf{I}} c_j\rangle \Rightarrow \langle G_{j+1} \mathbin{\mathsf{I}} c_{j+1}\rangle$,

$$P, \mathcal{A} \models G_{j+1} \wedge c_{j+1} \rightarrow G_j \wedge c_j$$

Say the step is of type 1. Literal L_j is just transferred from G_j to c_{j+1}, in which case

$$\models G_{j+1} \wedge c_{j+1} \leftrightarrow G_j \wedge c_j$$

Say the step is of type 2. Then

$$P \models p(s_1, \ldots, s_n) \leftarrow B_1 \wedge \ldots \wedge B_k$$

and since $\mathcal{A}$ treats equality as identity,

$$\mathcal{A} \models s_1 = t_1 \wedge \ldots \wedge s_n = t_n \rightarrow (p(s_1, \ldots, s_n) \leftrightarrow p(t_1, \ldots, t_n))$$

Together we have that

$$P, \mathcal{A} \models p(t_1, \ldots, t_n) \leftarrow s_1 = t_1 \wedge \ldots \wedge s_n = t_n \wedge B_1 \wedge \ldots \wedge B_k$$

Clearly, from the form of $\langle G_j \mathbin{\mathsf{I}} c_j\rangle$ and $\langle G_{j+1} \mathbin{\mathsf{I}} c_{j+1}\rangle$,

$$P, \mathcal{A} \models G_{j+1} \wedge c_{j+1} \rightarrow G_j \wedge c_j$$

Proof of Theorem 1 □

Because stochastic solvers are able to return solutions to constraints, we have the following slightly stronger soundness result as a corollary.

Corollary 1 *For Models A, B, C, or D, if* $\langle G \mathbin{\mathsf{I}} true\rangle \Rightarrow^* \langle \Box \mathbin{\mathsf{I}} c\rangle$ *is a derivation achieved using stochastic solver ssolv and usage strategy U, and* $ssolv(c, k) = (true, l, \theta)$, *then* $P, \mathcal{A} \models_\theta G$.

Given that we are using a probabilistic solver, we cannot expect to have a strong completeness result. In practice, CLP systems are already incomplete because of the fixed search strategy, so completeness may not be seen as vital. However, we would like to have some form of completeness.

Let us define $\mathit{Prff}(\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \stackrel{*}{\Rightarrow} \langle G_n \mathbin{\mathsf{I}} c_n\rangle, U)$ to be the probability that the solver applied to c_n using usage strategy U returns *false* when c_n is, in fact, satisfiable. Given the sequence

$$\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \stackrel{(1)}{\Rightarrow} \langle G_1 \mathbin{\mathsf{I}} c_1\rangle \stackrel{(1)}{\Rightarrow} \ldots \stackrel{(1)}{\Rightarrow} \langle \Box \mathbin{\mathsf{I}} c_n\rangle$$

where each c_i is satisfiable, the overall probability that this sequence succeeds using strategy U is

$$(1-\mathit{Prff}(\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \stackrel{(1)}{\Rightarrow}{}^{*} \langle G_1 \mathbin{\mathsf{I}} c_1\rangle, U))\times\ldots\times(1-\mathit{Prff}(\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \stackrel{(1)}{\Rightarrow}{}^{*} \langle \Box \mathbin{\mathsf{I}} c_n\rangle, U))$$

As long as there is a nonzero probability of finding a solution at each step, there is a finite probability that the sequence will succeed.

We can guarantee this if the usage strategy provides enough resources so that the solver at each stage has a nonzero probability of finding a solution. We call such combinations of solvers and usage strategies *nonstarving*. Because we assume for any satisfiable constraint c that $Pr(ans(ssolv(c,n)) = \mathit{true}) > 0$ if $n > 0$, any fixed-limit strategy is nonstarving.

For Model D, we have the usual completeness result for CLP, because when using Model D we never assume that the stochastic solver has determined unsatisfiability when this is not actually the case.

Proposition 1 (Jaffar and Lassez [JL87]) *If $P, \mathcal{A} \models \tilde{\exists} G$, then when using a correct solver, there exists a successful derivation of the form $\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \Rightarrow^{*} \langle \Box \mathbin{\mathsf{I}} c\rangle$.*

For the other models, we only have a weak (probabilistic) form of completeness.

Theorem 2 *Suppose program P is executed under Model A, B, or C using the nonstarving stochastic solver ssolv and the usage strategy U. If $P, \mathcal{A} \models \tilde{\exists} G$, then there is a nonzero probability that there is a successful derivation of the form $\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \Rightarrow^{*} \langle \Box \mathbin{\mathsf{I}} c\rangle$.*

Proof of Theorem 2 From Proposition 1, there exists a successful derivation using a correct solver. Employing the reasoning above, the probability of this derivation failing using a nonstarving usage strategy and a stochastic solver is nonzero.

Proof of Theorem 2 □

For the Models A, B, and C, we would like to ensure a greater degree of completeness. There exist usage strategies that can ensure this. Suppose

$$Prff(\langle G \mid true\rangle \stackrel{(1)}{\Rightarrow}^{m} \langle G_m \mid c_m\rangle, U) \leq (\frac{1}{2})^{m+k}$$

for all derivations containing m derivation steps of type 1, where $k \geq 1$. Then the probability that a derivation with l steps of type 1, where all constraints are satisfiable and the derivation is failed, is less than

$$(\frac{1}{2})^{k+1} + (\frac{1}{2})^{k+2} + \ldots + (\frac{1}{2})^{k+l}$$

In other words, the derivation will succeed using *ssolv* and a usage strategy U with at least probability $1 - (\frac{1}{2})^k$.

Consider the following stochastic solver, which is based on the stochastic solver described in the beginning of the previous section.

```
BOOL_SOLVE(c, n)
for i := 1 to n
      guess a truth assignment θ for c
      if Fθ is true return (true, i, θ)
endfor
return (unknown, n, θ)
```

For each constraint solve, the probability of an *unknown* answer for a satisfiable formula c is (from [WT92b]):

$$\epsilon = (1 - (1 - 1/m)^m)^n$$

where m is the size of c in terms of the number of clauses. We will assume that each primitive constraint is a single clause; thus the size of constraint c_m is less than or equal to m. Now $0.25 \leq (1 - 1/m)^m < 0.5$, hence $\epsilon < 0.75^n$.

If the usage strategy U chooses resources to be used at the m^{th} step in a derivation as

$$\begin{array}{rcl} n & > & (m+k)\ln(1/2)/\ln(3/4) \\ \text{e.g.,}\quad n & > & 2.5(m+k) \end{array}$$

Then

$$Prff(\langle G|true\rangle \Rightarrow^m \langle G_m|c_m\rangle, U) \leq (\frac{1}{2})^{m+k}$$

Thus we can expect[6] a probability of at least $1-(\frac{1}{2})^k$ of finding any derivation that should succeed. Note that the overall resources used by a derivation are quadratic in its length when using this strategy.

4.2 Finite Failure

Unfortunately, because of the nature of the solvers, we are unable to have any soundness result for finite failure for Models A, B, and C. This is because we cannot ensure that there is no *false failure*.

Definition 2 *A goal G is falsely failed for a usage strategy U and a stochastic solver ssolv if there is a sequence of states*

$$\langle G \mathbin{\mathsf{I}} true\rangle \Rightarrow \ldots \Rightarrow \langle \Box \mathbin{\mathsf{I}} c_n\rangle$$

where each c_i is satisfiable, but the execution of this derivation using ssolv and U fails.

Because a program P is only made up of implications, it has no negative consequences. To reason about failure logically, we need to consider the program as defining each predicate as equivalent to the disjunction of its rule bodies. This is the role of the program completion P^*, which is originally due to Clark [Cla78].

The definition of the n-ary predicate symbol p in the program P is the formula:

$$\forall x_1 \ \ldots \forall x_n \ p(x_1,\ldots,x_n) \leftrightarrow B_1 \vee \ldots \vee B_m$$

where each B_i corresponds to a rule in P of the form:

$$p(t_1,\ldots,t_n) \text{ :- } L_1,\ldots,L_k$$

[6] We have not taken into account the dependencies between successive constraint solves in this simplistic analysis.

and B_i is:

$$\exists y_1 \ \ldots \exists y_j \ (x_1 = t_1 \wedge \ldots \wedge x_n = t_n \wedge L_1 \wedge \ldots \wedge L_k)$$

where $y_1, \ldots, y_j$ are the variables in the original rule and $x_1, \ldots, x_n$ are variables that do not appear in any rule. Note that if there is no rule with head p, then the definition of p is simply:

$$\forall x_1 \ \ldots \forall x_n \ p(x_1, \ldots, x_n) \leftrightarrow \mathit{false}$$

as $\bigvee \emptyset$ is naturally considered to be *false*. The *completion*, P^*, of a constraint logic program P is the conjunction of the definitions of the user-defined predicates in P.

For Model D, of course, we obtain the usual "soundness of finite failure" result for CLP.

Proposition 2 (Jaffar and Lassez [JL87]) *Using a correct solver, if every derivation for the goal $\langle G \mid \mathit{true} \rangle$ is finitely failed, then $P^*, \mathcal{A} \models \neg\tilde{\exists} G$.*

In contrast to Models A and C, we do have strong completeness results for finite failure, provided the usage strategy satisfies a basic condition: a usage strategy is *eventually failing* if for each infinite or finite derivation D, for each finite prefix D_i of D there is a longer prefix D_j where $U(D_j) = (\mathit{false}, -)$. That is, eventually on every derivation the strategy will consider an *unknown* answer by the solver to be *false*.

The key to the result is the following result for complete solvers (see [JMMS96]).

Lemma 1 *If solv is a complete solver with regard to $\mathcal{A}$, P is a canonical program, and $P^*, \mathcal{A} \models \neg\tilde{\exists} G$, then every fair derivation for G is finitely failed.*

A canonical program is one where the immediate fixpoint operator $T_P^{\mathcal{A}}$ corresponding to P reaches its greatest fixpoint at the first infinite ordinal, i.e., $gfp(T_P^{\mathcal{A}}) = T_P^{\mathcal{A}} \downarrow \omega$. This is a technical condition that holds for all realistic programs (see [JS86] for more information).

Note that the following result does not usually hold for CLP systems, because the solvers are incomplete.

Theorem 3 *Let P be a canonical program. If $P^*, \mathcal{A} \models \neg\tilde{\exists} G$, then for Models A and C, if we use a fair selection strategy, then any derivation for G using the stochastic solver ssolv and the eventually failing usage strategy U is finitely failed.*

Proof of Theorem 3 Suppose, to the contrary, there is a successful or infinite derivation

$$\langle G \mid true\rangle \overset{(1)}{\Rightarrow}^* \langle G_i \mid c_i\rangle \overset{(1)}{\Rightarrow}^* \langle G_j \mid c_j\rangle \Rightarrow^* \ldots$$

Using Lemma 1 and a complete solver, the derivation is finitely failed; hence there exists state $\langle G_i \mid c_i\rangle$ where $\mathcal{A} \models \neg\tilde{\exists} c_i$. Because the usage strategy is eventually failing, there exists state $\langle G_j \mid c_j\rangle$ where $U(\langle G \mid true\rangle \Rightarrow^* \langle G_j \mid c_j\rangle = (false, -)$. Since $\models c_j \rightarrow c_i$ and the stochastic solver must return *unknown* at this state, the derivation fails.

Proof of Theorem 3 □

We can remove the technical restriction on P if we use just the theory of the constraint domain, $th(\mathcal{A})$, that is, all first-order sentences true in $\mathcal{A}$.

Theorem 4 *If $P^*, th(\mathcal{A}) \models \neg\tilde{\exists} G$, then for Models A and C, if we use a fair selection strategy, then any derivation for G using the stochastic solver ssolv and the eventually failing usage strategy U is finitely failed.*

Proof of Theorem 4 As for Theorem 3, since an appropriate version of Lemma 1 holds for $th(\mathcal{A})$ without the canonicality condition.

Proof of Theorem 4 □

4.3 Iterative Deepening

For Models A, B, and C, the possibility of false failure on all successful derivations means that a goal that has a solution may falsely fail. In particular, given a derivation-limit usage strategy for Models A or C, we may expect to falsely fail on all successful derivations if the limit is too small. This leads to the idea of reexecuting the goal with more resources, to avoid the false failure and find a success. Thus we consider a form of the iterative-deepening approach.

Iterative deepening works by reexecuting a finitely failed goal, using greater resources. For example, we can double the limits in the usage strategy. So when a goal G fails using the fixed-limit strategy $U_{fl(1000)}$, we reexecute the goal using the usage strategy $U_{fl(2000)}$. If that execution fails to find an answer, then we reexecute with the usage strategy $U_{fl(4000)}$, etc.

The idea driving this approach is that a solution will be found quicker if we automatically use the minimal number of resources required to avoid false failure. If we give too many resources to solve a problem, execution will be slowed, because failure is determined by resource exhaustion; hence, giving too many resources slows down execution. Iterative deepening automatically determines the (nearly) least number of resources required to find a solution. When using an iterative-deepening approach, there is no possibility of finite failure, because a goal with no solutions will run infinitely.

Technically, we also need to ensure that no infinite derivations are possible, so that, for example, when using usage strategy $U_{fl(R)}$, we never consider derivations of length greater than R. Otherwise, we cannot guarantee a goal will either succeed or fail. In practice, the resource limits will usually manage this, but in theory, we must use a *depth-bounded* usage strategy. A depth-bounded strategy is such that there exists a monotonically increasing function $l(R)$ in terms of the argument of the usage strategy (or arguments, if there are more than one), such that for any derivation of length $m \geq l(R)$, $U(\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \overset{m}{\Rightarrow} \langle G_m \mathbin{\mathsf{I}} c_m\rangle) = (\mathit{false}, -1)$. In other words, after some length of derivation, the strategy allows only negative resources to the solver, ensuring it returns *unknown*, and this causes failure. For example, we can define a depth-bounded fixed-limit strategy as

$$\begin{aligned} U_{dbfl(R)}(\langle G \mathbin{\mathsf{I}} \mathit{true}\rangle \overset{(1)}{\Rightarrow}^{m} \langle G_m \mathbin{\mathsf{I}} c_m\rangle) \quad &= (\mathit{false}, R) \quad \text{if } m < R \\ &= (\mathit{false}, -1) \quad \text{if } m \geq R \end{aligned}$$

Thus any derivation of length $\geq R$ will be failed.

Using an iterative-deepening approach, we can ensure that eventually we will find a success if one exists. Note that iterative deepening does not make sense in the context of Model D, because it never falsely fails.

Theorem 5 *If $P, \mathcal{A} \models \tilde{\exists} G$, then executing goal G under Models A or C using iterative deepening and a nonstarving and depth-bounded usage strategy U and the solver ssolv will yield a successful derivation.*

Proof of Theorem 5 Because the usage strategy is depth bounded, every derivation tree for G is finite; after some finite limit, all derivations fail. The only way the iterative-deepening procedure terminates is through success; hence it remains to show that a successful derivation is eventually found. Such a derivation exists by Proposition 1, and eventually, because the bound

limit is monotonically increasing, the successful derivation will be within the depth bound. By Theorem 2, this derivation has a nonzero probability of being a successful derivation for U and $ssolv$. Execution continues until it becomes a successful derivation.

Proof of Theorem 5 □

5 A Constraint Solver: GENET

We have used GENET [WT92a, WT91], a generic neural network simulator, to demonstrate the feasibility of our proposal. GENET is used to solve binary CSPs with finite domains. A GENET network consists of a cluster of nodes for each domain variable in a CSP. Each node denotes a value (label) in the corresponding domain variable, and each constraint is represented by a set of inhibitory connections between nodes with incompatible values. GENET works by updating each variable using the min-conflict heuristic until a valuation that is a local minimum in terms of conflict is found. It then applies a heuristic learning rule which penalizes constraints that are violated in the current local minimum. Updating of variables proceeds as before. We count one resource as a single update to each variable.

In the original GENET model, the entire network must be constructed before computation starts. However, a constraint solver in a CLP system must support efficient incremental execution, because new primitive constraints are being added to an existing solvable constraint during a derivation. Therefore, an efficient incremental version of GENET is necessary. A naive but efficient incremental GENET, called I-GENET, adds new primitive constraints and variables to the network as they are collected. Its incrementality originates from the re-use of the connection weights, which are computed using the heuristic learning rule in previous cycle. Thus, the network is *trained* while it is being built incrementally. A more detailed discussion of the incremental GENET is contained in [Tam95].

The I-GENET model only allows a monotonic increase to the constraint represented by the network. It cannot "remove" the primitive constraints added to the network; hence it cannot be used to solve CSPs that require a tree search on constraints, such as the disjunctive scheduling problem in [Hen89] (although disjunctive constraints can be directly encoded into the GENET network [LLW95]). To obviate this shortcoming, we extend I-GENET to "undo" the effect of a most-recently added primitive constraint.

This is done by disconnecting the inhibitory connections made for that primitive constraint. In this way, we backtrack on the primitive constraints added to the GENET network in a chronological order. We use a trailing stack to keep track of the most recently added primitive constraints for backtracking.

6 Experimental Results

To compare the different models of stochastic solvers, we have used ECLiPSe version 3.5.1 and the GNU C Compiler version 2.6.3, running under SunOS 5.3. To build prototypes for Models A, B, and C, we used the backtrackable I-GENET as the stochastic constraint solver, and the ECLiPSe `fd` library as a propagation solver. Note that the first-fail principle (i.e., choose the variable with the smallest domain to be instantiated first) is used with forward checking to improve the performance of the search in the ECLiPSe system. We also include another example of Model A, using a version of GENET that is extended to incorporate lazy arc consistency [ST96] to reduce the domains of variables. The lazy arc consistency is a "lazy" notion of arc consistency [Tsa93] that is most natural for GENET. Essentially, it enforces arc consistency only on the current variable assignments obtained by the GENET solver. Any arc-inconsistent label $(X_i = v)$ found by the lazy arc-consistency technique will be removed from the domain of the variable X_i. Using this solver (denoted as Model A$'$) has an effect similar to Model C, where the reduction of domains comes from the propagation solver.

Owing to the difficulty of implementing parallel asynchronous procedures within ECLiPSe, the Model D solver is simulated. At each constraint solve we execute the propagation solver and the stochastic solver, and take as the solver time the minimum of the times if both return *true* or *false*, the time of the solver that returns *true* or *false* if only one does so, and the maximum if both return *unknown*. This provides an optimistic time for Model D because, for example, we do not take into account when a solver may still be running from the previous constraint solve.

We give results of preliminary experiments comparing the different models using usage strategy $U_{fl}(1000)$, and compare our systems versus a traditional propagation-based CLP approach on a set of CSPs with and without disjunctive constraints. The Hamiltonian path calculation and disjunctive graph coloring are examples of CSPs with disjunctive constraints, while N-queens and permutation-generation problems are examples of CSPs without

disjunctive constraints. In all the test cases, the CPU times for the different models are the medians over 10 successful runs to find a solution. All CPU times are measured in seconds. For the different models each run was terminated if the constraint solvers used 1,000,000 resources without finding a solution. Also, 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 that an execution took more than 10 hours.

6.1.1 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* [Pra76]. The Hamiltonian path problem is a practical CSP that is very similar to the route-planning problems faced by many circus, traveling road companies, and traveling salespeople. In any case, it can be regarded as a nonoptimizing version of the well-known "traveling salesman problem," in which the salesman does not need to return to the original city.

The following shows the relevant clauses in an ECLiPSe program for ordering the vertices from 1 to n in a graph to calculate the Hamiltonian path.

```
ordering([], _).
ordering([arc(X,L)|T], CLimit) :-
        before(X, L, CLimit),
        ordering(T, CLimit).
before(X, [], CLimit) :- addConstraint([X = 1], CLimit).
before(X, [H|T], CLimit) :- addConstraint([X = H + 1], CLimit)
      ;
      before(X, T, CLimit).
```

The variables X and H stand for different vertices in the graph. The CLimit denotes the convergence cycle limit for the GENET solver. The arc(X,L) specifies a relation between the variable X and a list L of variables, in which the vertex denoted by X has an arc with the other vertex denoted by any element in the list L. The ordering predicate takes a list of arc relations together with the resource CLimit, and assigns a value from 1 to n to each variable X. The before predicate specifies that the vertex denoted by variable X is either the first one (X = 1) in the ordering, or it is preceded by some

Times	ECLiPSe	A	B	C	D	A′
10	0.040	4.195	0.085	2.565	0.04	0.460
20	1090	58.27	1825	50.65	—	31.75
30	?	5493	?	4287	—	2687

Table 1: Hamiltonian path CPU times

Backtracks	ECLiPSe	A	B	C	D	A′
10	11	11	11	15.5	11	11
20	276012	21	276012	20	—	20
30	?	46	?	46	—	46

Table 2: Hamiltonian path backtracks

element in the list L. The addConstraint(CL, CLimit) is a predicate that makes an external function call to add the list CL of constraints into the GENET solver with the resource CLimit, which may return either *true*, *false*, or *unknown*.

Tables 1 and 2 show the performance of ECLiPSe and the different models on Hamiltonian path problems using fixed-limit strategies with limits set at 1,000; Table 1 shows CPU seconds for each benchmark under the different models, and Table 2 shows the average number of backtracks in the search. The first two problems are coded from some interesting real-life examples in graph theory [Pra76]. The last problem is a modified example obtained from [LLW95].

In general, the stochastic Models A and C outperform ECLiPSe and Model B on all but the smallest example. This is because Models A and C determine unsatisfiability much earlier than does the ECLiPSe. Also, the number of possible branches visited by Models A and C in the search tree is much smaller than that for the ECLiPSe, because of this early resource exhaustion. Models B and D fail to improve on ECLiPSe, because in this example no labeling is required. For these problems, Model C betters Model A, showing that the propagation solver can provide useful information for the stochastic solver during the search. In these problems, the improved stochastic solver (Model A′) finds useful domain reduction information without the overhead of communication with the propagation solver.

Graph		CPU Time					
Nodes	Colors	ECLiPSe	A	B	C	D	A′
10	3	19.96	1.580	—	1.645	5.235	1.820
20	4	1362	0.690	—	0.770	1135	0.770
30	5	0.050	0.115	0.095	0.210	0.080	0.120
40	5	0.070	0.145	0.105	0.220	0.110	0.155
50	5	0.090	0.200	0.140	0.345	0.160	0.225

Table 3: Disjunctive graph-coloring CPU times

Graph		Backtracks					
Nodes	Colors	ECLiPSe	A	B	C	D	A′
10	3	6428	4	—	4	6428	4
20	4	65535	1	—	1	65535	1
30	5	0	0	0	0	0	0
40	5	0	0	0	0	0	0
50	5	0	0	0	0	0	0

Table 4: Disjunctive graph-coloring backtracks

6.1.2 Disjunctive Graph-Coloring Problems

The disjunctive graph-coloring problem is to color a (possibly nonplanar) graph with a number of hyper-arcs that connect nodes. A hyper-arc of the form $\{(i_1, j_1), (i_2, j_2)\}$ specifies that at least one of the pairs of nodes $(i_k, j_k), 1 \leq k \leq 2$ must be colored differently. The disjunctive graph-coloring problem has wide applicability in assembling timetables, schedules, and production plans.

Tables 3 and 4 show results for ECLiPSe and the different models on a set of small-sized disjunctive graph-coloring problems. In general, each of the approaches where the stochastic solver controls the search (Models A and C) outperforms ECLiPSe when the problem requires backtracking. This is because these models do much less backtracking by avoiding unpromising branches with early resource exhaustion. Also, Models A and C do not require a backtracking enumeration search. This is an example where no useful domain reduction information is discovered by Model C or model A′;

Number of Queens	ECLiPSe	A	B	C	D	A′
10	0.040	0.200	0.195	0.320	0.205	0.290
20	0.220	3.665	2.195	4.200	3.540	4.020
30	0.880	23.79	10.11	27.43	21.67	23.91
40	0.810	93.12	33.58	96.04	82.96	95.82
50	63.42	253.3	73.09	268.8	234.8	257.1
60	3.580	594.0	146.4	625.5	552.4	605.8
70	66.67	1172	273.9	1242	1124	1231

Table 5: N-queens problem

hence they do not improve on Model A, and suffer some overhead. Model D just beats ECLiPSe for the cases where there is a lot of backtracking, but this may be owing to the optimistic simulation.

6.1.3 The *N*-Queens Problem

The N-queens problem is to place N queens onto an $N \times N$ chess board so that no queens can be attacked. A queen can be attacked if it is on the same column, row, or diagonal as any other queen. It is a standard benchmark for CSPs, since the size N can be increased without limit.

Table 5 shows the CPU time of ECLiPSe and the different models on N-queens problems where N ranges from 10 to 70. In general, ECLiPSe outperforms the stochastic approaches, because the problems do not require backtracking on constraints, plus, using the first-fail principle significantly improves the performance of the enumerative search strategy. Model B performs best among all the models, because the problem does not involve any search on constraints and it does not need to expend resources on derivation steps, except for the last one. Model D in this case is essentially a slightly faster version of Model C, because for almost all of the computation, the stochastic solver is the slower solver. Again, Models A′ and C do extra work without gaining any useful information for these problems. Thus, they are slower than Model A.

Length of List	ECLiPSe	A	B	C	D	A′
9	0.010	0.210	0.075	0.190	0.185	0.210
10	0.020	0.325	0.145	0.455	0.375	0.355
20	8.440	6.470	1.210	6.295	5.665	7.035
30	?	50.61	5.880	45.69	43.22	52.77

Table 6: Permutation generation

6.1.4 Permutation Generation

Given a permutation f on the integers from 1 to n, we define the monotonies of f as a vector $(m_1, m_2, \ldots, m_{n-1})$, where m_i equals 1 if the value of the $i+1$-th element in the permutation is greater than that of the i-th element and 0 otherwise, and the vector $(a_1, a_2, \ldots, a_{n-1})$ of advances of f such that $a_i = 1$ if the integer $f(i)+1$ is placed in the permutation on the right of $f(i)$, and 0 otherwise. The problem is to construct all the permutations of a given range that admit a given vector of monotonies and a given vector of advances.

The following monotonie(M, L, CLimit) predicate shows how to ensure monotony in the permutation list L, given a vector M with 0s and 1s and the resource CLimit to the GENET solver.

```
monotonie([], [X], _).
monotonie([1|Lm], [X1, X2|Lv], CLimit) :-
        addConstraint([X2 ¿ X1], CLimit),
        monotonie(Lm, [X2|Lv], CLimit).
monotonie([0|Lm], [X1, X2|Lv], CLimit) :-
        addConstraint([X2 ≤ X1], CLimit),
        monotonie(Lm, [X2|Lv], CLimit).
```

Table 6 shows the performance of ECLiPSe and the different models on the permutation generation, given a number of vectors of different sizes for monotonies and advances. The first example is taken from [Hen89], while the others are arbitrary examples of permutation lists with lengths from 10 to 30. For the first two test cases, ECLiPSe performs better than our models because the search space is relatively small. However, when the length of the permutation list grows larger, the stochastic approaches outperform ECLiPSe. In particular, Model B performs best, because the problems do

not require backtracking on constraints. Model D again essentially acts as a slightly faster version of Model C. For these problems, Model C, in general, performs better than Models A and A′, as it gains useful information from the propagation solver, which cannot be deduced by Model A′.

7 Conclusion

The integration of relaxation-based search methods and enumerative search methods is an important aim, because each method has advantages over the other in solving different forms of CSPs. In this paper we propose one approach to such an integration: stochastic constraint solvers represent the relaxation-based search methods, while CLP with a propagation solver and some kind of labeling program represents enumerative search.

Using a stochastic solver within a constraint logic programming system relies upon inferring failure information from the solvers' inability to find a solution. This inference which may be false immediately gives rise to questions about what semantics results continue to hold. In this paper we have shown which results continue to apply.

We have also demonstrated the benefits that can arise from using stochastic constraint solvers within CLP. First, the stochastic constraint solvers may be able to be applied to a larger class of problems, because disjunction can be handled by the built-in search mechanism of CLP. Second, empirical results confirm the advantages of using stochastic methods, in particular in conjunction with other constraint-solving approaches.

Summarizing the empirical results, we have that using a stochastic solver to control search (Models A and C) is of benefit for disjunctive problems where the search space of constraints is large. For nondisjunctive problems, Model B is most effective and outperforms enumerative search in many instances. Cooperating solvers (Models C and D) can result in significant improvements, and usually do not add too much overhead. Model D is probably only advantageous over both Model B and Model C in a few cases. Improving underlying stochastic solvers (Model A′) is another worthwhile approach. No model is uniformly better, and the user should be prepared to experiment.

This paper certainly does not complete the question of what is the best approach to integrating relaxation-based and enumerative searches. There are other approaches that need to be examined. Another obvious area re-

quiring further investigation is the use of stochastic methods to improve the finding of optimal or good solutions in CLP systems.

References

[AK89] E. H. L. Aarts and J. H. M. Korst. Boltzmann machines for traveling salesman problems. *European Journal of Operational Research*, 39(39):79–95, 1989.

[BD95] James Bowen and Gerry Dozier. Solving constraint satisfaction problems using a genetic/systematic seach hybrid that realizes when to quit. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 122–129, San Mateo, CA, 1995. Morgan Kaufmann.

[Cla78] K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322, New York, 1978. Plenum Press.

[DSH88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of European Conference on Artificial Intelligence*, pages 290–295, Amsterdam, 1988. North-Holland.

[DSH90] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:75–93, 1990.

[DTWZ94] 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 American Association for Artificial Intelligence '94*, pages 325–330, Cambridge, MA, 1994. MIT Press.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Cambridge, MA, 1989. The MIT Press.

[HT85] J. J. Hopfield and D. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[IO95] A. R.-A. Illera and J. Ortiz. Labeling in clp(fd) with evolutionary programming. In Maria Alpuente and I. Sessa Maria, editors, *Proceedings of GULP-PRODE 95*, pages 569–590, Salerno, Italy, 1995. Poligraf Press.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *POPL'87: Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, 1987. ACM Press.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[JMMS96] Joxan Jaffar, Michael J. Maher, Kim Marriott, and Peter J. Stuckey. Semantics of constraint logic programs. Technical Report 39, Department of Computer Science, University of Melbourne, 1996.

[JS86] J. Jaffar and P. J. Stuckey. Canonical logic programs. *Journal of Logic Programming*, 2(3):143–155, 1986.

[KMMR96] J. Kok, E. Marchiori, M. Marchiori, and C. Rossi. Evolutionary training of CLP-constrained neural networks. In *Proceedings of Practical Application of Constraint Technology Conference*, pages 129–142, London, 1996. The Practical Application Company.

[LLW95] J. H. M. Lee, H. F. Leung, and H. W. Won. Extending GENET for non-binary CSPs. In *Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, New York, 1995. IEEE.

[LT95] Jimmy H. M. Lee and Vincent Tam. A framework for integrating artificial neural networks and logic programming. *International Journal on Artificial Intelligence Tools*, 4(1–2):3–32, June 1995.

[Men87] E. Mendelson. *Introduction to Mathematical Logic*, 3rd edition, Pacific Grove, CA, 1987. Wadsworth and Brooks.

[Pra76] Ronald E. Prather. *Discrete Mathematical Structures for Computer Science*, Boston, MA, 1976. Houghton Mifflin.

[Roj96] María Cristina Riff Rojas. From quasi-solutions to solution: An evolutionary algorithm to solve CSP. In *Proceedings of Principles and Practice of Constraint Programming (CP96)*, volume 118 of *Lecture Notes in Computer Science*, pages 367–381. Springer-Verlag, 1996.

[ST96] Peter Stuckey and Vincent Tam. Extending GENET with lazy arc consistency. Technical Report 96/8, Department of Computer Science, University of Melbourne, 1996.

[Tam95] 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.

[Tsa93] Edward Tsang. *Foundations of Constraint Satisfaction*. London, 1993. Academic Press.

[WT91] C. Wang and E. Tsang. Solving satisfaction problems using neural networks. In *Proceedings of IEEE Second International Conference on Artificial Neural Networks*, New York, 1991. IEEE Computer Society.

[WT92a] C. J. Wang and E. P. K. Tsang. *A Generic Neural Network Approach for Constraint Satisfaction Problems*, pages 12–22. Berlin, 1992. Springer-Verlag.

[WT92b] L. C. Wu and C. Y. Tang. Solving the satisfiability problem by using randomized approach. *Information Processing Letters*, 41:295–299, 1992.