Using Stochastic Methods to Guide Search in CLP: a Preliminary Report*

J.H.M. Lee[1], H.F. Leung[1], P.J. Stuckey[2], V.W.L. Tam[2], and H.W. Won[1]

[1] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong. Email: {jlee,lhf,hwwon}@cse.cuhk.edu.hk
[2] Department of Computer Science, University of Melbourne, 3052 Parkville, Australia. Email: {pjs,vtam}@cs.mu.oz.au

Abstract. Recently Lee, Stuckey and Tam have shown the advantages of incorporating stochastic solvers into constraint logic programming (CLP) systems. Their approaches, while efficient, both suffer from some form of incompleteness and complication in semantics. This paper proposes a generalization of these previous efforts by using stochastic methods to guide and speed up the search of derivation trees for successful branches. By spending computational effort to exercise the stochastic solver at various nodes in the derivation tree, additional information is obtained to suggest (a) delaying exploration of unpromising subtrees and (b) visiting promising children first. Using these simple guidelines we give two example search strategies extending the basic depth-first search procedure used typically in CLP systems. Each extension exhibits a different degree of interaction and cooperation between the principal CLP solver and the stochastic solver. While encompassing all previous integration schemes, the generality of our framework also opens up myriad possibilities for using a stochastic solver to improve search efficiency. Last but not least, since the interaction is through the search strategy most semantic properties of constraint logic programming are inherited.

1 Introduction

Constraint logic programming (CLP) is a paradigm formed at the intersection of constraint processing and logic programming. While inheriting a strong theoretical background from logic programming, CLP offers an expressive and flexible language for problem specification and heuristics programming, allowing rapid program development for complex problems and enabling programs to be easily modified and extended. A wealth of CLP systems exist, such as CLP($\mathcal{R}$) [5] and CHIP [2], each differing in its constraint domain and serving in different areas of applications. Problems tackled by CLP systems include the important class of constraint satisfaction problems (CSP's) [9], which are well-known to be NP-hard in general. Although much work is in progress to improve the efficiency of constraint solving in the context of CLP, the techniques used in CLP systems are algorithmic in nature and seem unlikely (at least in the near future) to overcome the difficulties in solving large-scale and difficult instances of NP-hard problems.

Apart from the traditional algorithmic techniques, another major class of constraint solving strategies are stochastic methods, such as randomized search, local search,

* This project is supported in part by a CUHK Direct Grant, ARC Small Grant S4969494, and a University of Melbourne Young Asian Scholar Grant.

connectionist approaches, simulated annealing and genetic algorithms. These methods are in general not guaranteed to find a solution (even if there is one), and typically involve a resource bound on the search before the search has to be abandoned. Such methods are rarely able to determine when a constraint is unsatisfiable, but they can be considerably more efficient and exhibit better scalability than traditional solvers on large or hard CSP instances. An additional advantage of these stochastic methods is that answers returned by them are usually in the form of a valuation (assignment of values to variables) while some CLP systems return only a set of answer constraints which may require further manipulation to extract a solution.

An attractive proposal is to incorporate stochastic methods into CLP, yielding an efficient and expressive constraint programming language. Two earlier approaches consider replacing the usual propagation-based solver in CHIP-like CLP languages by stochastic solvers. PROCLANN [8] uses a stochastic solver based on artificial neural networks. The language allows only a single thread of execution so that backtracking never occurs. Constraints are collected during derivation and the stochastic solver is exercised at the end of a derivation as a solution finder. PROCLANN, being a committed-choice language, sacrifices completeness in its semantics and fails to handle problems that require tree search on constraints, such as disjunctive scheduling. Stuckey and Tam [11] study how a stochastic solver can be used in conjunction with tree search. By interpreting an unknown result (after a predefined resource limit) of a stochastic solver as failure the stochastic solver can be used to control search for a successful derivation in the derivation tree. A problem of this approach is that a reasonable resource limit is difficult to obtain. Too small a limit results in the possibility of *false-failure*, while too large a limit hinders upon execution efficiency. In both attempts, the stochastic solver is integrated too *deeply* into CLP. The resulting languages suffer from some form of incompleteness and complication in semantics.

In this paper we introduce a general integration framework for CLP and stochastic solvers so as to speed up the search for successful branches. The framework is applicable to general CLP systems of any computation domain, and retains all the elegant semantic properties of CLP. Instead of acting as the principal solver of a CLP system, in our framework, the stochastic solver augments the usual solver of the CLP system. Given a CLP system with some (algorithmic) solver $solv$, a program P and a goal G of that system, we use a stochastic solver to guide the search of the corresponding derivation tree of P and G constructed using $solv$ according to the operational semantics of CLP. We use two guiding principles to speed up the search of solutions: (a) delay exploring unpromising subtrees and (b) explore promising children first. These guidelines open up different possibilities for interactions and cooperation between the principal solver of a CLP system and the stochastic solver to explore the search tree. The main contribution of the paper is the generality and naturalness of the resulting integration framework. The framework covers all the previously defined frameworks for integrating stochastic methods with CLP systems and, through its generality, suggests many possible schemes for making use of stochastic solvers to improve search.

This paper is organized as follows. Section 2 gives an overview of constraint logic programming, and in particular the notion of derivation trees. In Section 3 we briefly discuss stochastic methods for constraint solving and some of the issues that arise from

using the methods in conjunction with CLP systems. We outline the basic search strategy used in most CLP systems, and two ways in which we can use a stochastic solver to improve search in Section 4. Section 5 contains preliminary benchmarking results. In Section 6 we give a critical evaluation of our framework.

2 Constraint Logic Programming

We briefly introduce constraint logic programming. For more details, see [4]. A *constraint domain* $\mathcal{A}$ is a structure defining the meaning of the primitive constraints. A *primitive constraint* is of the form $r(t_1, \cdots, t_n)$ where r is a relation defined by $\mathcal{A}$ and $t_1, \cdots, t_n$ are *terms* from $\mathcal{A}$. A *constraint* is of the form $c_1 \wedge \cdots \wedge c_n$ where $n \geq 0$ and $c_1, \cdots, c_n$ are primitive constraints. If $n = 0$, c is interpreted as $true$.

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 $S = \{X_1, \cdots, X_n\}$. We write θ as $\{X_1 \mapsto a_1, \cdots, X_n \mapsto a_n\}$ to indicate that each X_i is assigned the value a_i. Let $vars(F)$ denote the set of variables occurring in a formula F. Let $\bar{\exists}_V F$ represent the formula $\exists w_1 \cdots \exists w_k F$ where $\{w_1, \ldots, w_k\} = vars(F) - V$. Let $\bar{\exists} F$ 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 $true$, or equivalently, $\mathcal{A} \models_\theta c$. A constraint c is *satisfiable* if there exists 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 $unknown$. Whenever $solv(c)$ returns $true$ the constraint c must be satisfiable, and whenever $solv(c)$ returns $false$ the constraint c must be unsatisfiable. An $unknown$ result indicates that the solver is unable to determine the satisfiability of c. This last case is typically not mentioned in the literature but is a problem inherent in all incomplete solvers. A constraint solver is *complete* if it only returns $true$ or $false$.

An *atom*, A, is of the form $p(t_1, \cdots, t_n)$ where p is a n-ary *predicate symbol* and $t_1, \cdots, t_n$ are terms. A *goal* G is a sequence of atoms $L_1, \cdots, L_m$. The empty sequence of atoms is denoted $\Box$. A *rule*, R, is of the form A:-$c \mid B$ where A is an atom, c is a constraint, and B is a goal. A is called the *head* of R, c is called the constraint of R and B is called the *body* of R. A *(constraint logic) program*, P, is a list of rules.

Execution proceeds by mapping one state to another. A *state* is a pair written $\langle G \mid c \rangle$ where c is a constraint and G is a goal. 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 $L_1, \cdots, L_{i-1}, L_i, L_{i+1}, \cdots, L_m$ where L_j ($1 \leq j \leq m$) are atoms and L_i is the *selected literal*, an atom of the form $p(t_1, \cdots, t_n)$. Let R be a variant (by variable renaming) of a rule in P, sharing no variables with $\langle G_0 \mid c_0 \rangle$, which has the form $p(s_1, \cdots, s_n) \leftarrow c \mid B_1, \cdots, B_k$. Let $c_1 \equiv c_0 \wedge s_1 = t_1 \wedge \cdots \wedge s_n = t_n \wedge c$. There are two possibilities. Either $solv(c_1) \neq false$ in which case G_1 is $L_1, \cdots, L_{i-1}, B_1, \cdots, B_k, L_{i+1}, \cdots, L_m$, or $solv(c_1) = false$ in which case the state is failed and the goal becomes $G_1 = \Box$ to prevent further search.

A derivation from state $\langle G_0 \mid c_0 \rangle$ is a sequence $\langle G_0 \mid c_0 \rangle \Rightarrow \langle G_1 \mid c_1 \rangle \Rightarrow \cdots$ where at each stage $i \geq 0$ the selected literal is given by some *selection strategy* and $\langle G_i \mid c_i \rangle \Rightarrow \langle G_{i+1} \mid c_{i+1} \rangle$. A derivation for goal G is a derivation from state $\langle G \mid true \rangle$. A derivation is *successful* if it is finite and the last state $\langle \Box \mid c_n \rangle$ satisfies $solv(c_n) \neq false$. If

$\langle G \mid true \rangle \Rightarrow \cdots \Rightarrow \langle \Box \mid c_n \rangle$ is a successful derivation then $\bar{\exists}_{vars(G)} c_n$ is an *answer* to the goal G. A derivation is *finitely failed* if it is finite and at the last state $solv(c_n) = false$. Thus any derivation can be extended unless it is successful or the solver detects an unsatisfiable constraint.

A *derivation tree* for a state $\langle G \mid c \rangle$, program P and selection strategy S is defined as follows. The state $\langle G \mid c \rangle$ is the root node of the tree. For any node in the tree $\langle G \mid c \rangle$, if $p(t_1, \cdots, t_n)$ is the atom selected from G by S and $R_1, \ldots, R_m$ are variants of the rules in P of the form $p(s_1, \cdots, s_n) \leftarrow c \mid B_1, \cdots, B_k$ in the order the rules appear, then the i^{th} child node of $\langle G \mid c \rangle$ is the state obtained by applying a derivation step to state $\langle G \mid c \rangle$ selecting $p(t_1, \cdots, t_n)$ and using rule R_i. To each node $s \equiv \langle G \mid c \rangle$ in the derivation tree we associate a record of information $\{s.nochildren, s.child[]\}$ containing the number of children nodes n and an array (from 1 to n) of child nodes respectively.

3 Stochastic Solvers

In this section we define and characterize stochastic solvers and discuss their use in CLP. Stochastic algorithms are designed to find solutions to constraint problems, using a relaxation-based search. Stochastic solvers in general can never determine unsatisfiability, when applied to an unsatisfiable constraint. They will execute forever unless some limit is placed on them[3]. Hence they usually have a resource limit, such as a time bound or maximum number of iterations. Thus, we define a stochastic constraint solver as a solver that takes two arguments, a constraint and a resource bound.

A *stochastic solver* $ssolv$ takes as input a constraint c and resource limit n, and returns a triple $(Truth, Valuation, Usage)$. If $Truth$ is $true$, then $Valuation$ is a solution and $Usage \leq n$ is the amount of resources used to find the solution. If $Truth$ is $unknown$, then $Valuation$ is a possibly meaningless valuation and $Usage$ is simply the input resource limit.

We are going to use a stochastic constraint solver in searching a derivation tree. In order to do this we need a way of specifying at each node in the tree how many resources should be used. This is the role of a usage strategy. A *usage strategy* U is a function on derivations which determines how a stochastic solver is to be used. Given a derivation D, $\langle G \mid true \rangle \Rightarrow^* \langle G_n \mid c_n \rangle$, the usage strategy $R = U(D)$ gives a resource limit R to be used in solving c_n. Many usage strategies are possible. For example, the simplest usage strategy is the fixed limit strategy $U_{fl(R)}(D) = R$ where, in each step, R resources are allocated. Many other strategies are possible, for example, where the fixed limit varies with the length of the derivation.

4 Search Strategies

Given a goal G and a program P, the aim of a CLP system is to find answers to the goal using the program by constructing and exploring a derivation tree for G and P. The standard search strategy used in a CLP system is depth-first. Execution begins by visiting the root node. Visiting a node involves visiting each of the children of the node

[3] Some rare stochastic solvers [12] are able to return failure in some cases. It is not difficult to extend our presentation to handle this class of stochastic solvers but we do not consider them in this paper for simplicity.

and then returning to the parent. Children of a node s are visited in the order they are stored in the array $s.child[]$ of s. This can be expressed using the following pseudo-code.

```
visit(⟨G ❙ c⟩)
    if solv(c) = false then return
    if G ≡ □ then success(c)
    else
        for i := 1 to ⟨G ❙ c⟩.nochildren
            visit(⟨G ❙ c⟩.child[i])
    return
```

The aim of the search is to find a successful state $\langle G \mid c \rangle$ where $G \equiv \Box$ and c is possibly satisfiable, *i.e.* not determined as unsatisfiable by the solver. When this is found the procedure success returns an answer constraint to the user. We assume that execution either continues if the user asks for another solution (or all solutions), or halts inside the success procedure.

One of the disadvantages of many constraint solvers that are used within CLP systems is that they are incomplete. When the constraint solver $solv$ is applied to a constraint c, $solv(c)$ will sometimes answer $unknown$. This indicates that the solver cannot determine satisfiability or unsatisfiability of the constraint. In particular a large class of commonly used constraint solvers (propagation-based solvers) rarely answer $true$, usually returning $unknown$ or $false$.

In the next two subsections we look at two ways of using stochastic constraint solvers to improve the search strategy of the CLP system. Note that the stochastic constraint solver is used *in addition* to the underlying constraint solver $solv$, which defines the derivation tree to be searched.

4.1 Delay Exploring Unpromising Subtrees

When the constraint solver $solv$ returns $unknown$ at a state $\langle G \mid c \rangle$ and the constraint c is in fact unsatisfiable, any states occurring in the subtree rooted by $\langle G \mid c \rangle$ will in fact involve constraints that are unsatisfiable. Hence exploring this subtree will never discover an answer, but the solver cannot determine this. For weak constraint solvers such as the propagation-based solvers, this can mean exploring large subtrees that would not need to be searched if the solver were complete. For this reason it can be worthwhile making use of a stochastic constraint solver $ssolv$.

Stochastic constraint solvers return $true$ when they find a solution. If this is the case then the solver guarantees that the subtree rooted at the current node in the search tree is worth investigating, at least in the sense that it may contain an answer. When it returns $unknown$ there is no such guarantee. If we trust in the stochastic solver to usually find a solution if there is one, then this is evidence that the subtree should not be explored, *i.e.* the subtree is not promising. Hence we can *delay* visiting this subtree until after other parts of the derivation tree are explored.

An extreme case of delayed exploration is to delay indefinitely. Clearly this search strategy is incomplete in the sense that it can miss successful derivations, because sometimes the stochastic constraint solver will return $unknown$ when the constraint is

satisfiable. However, there are already a number of sources of incompleteness for CLP systems, such as depth-first search and unfair literal selection. We should be willing to accept this approach if the resulting incompleteness is not too great. The following algorithm enforces this search.

```
visit-fail(⟨G ❙ c⟩, D, U)
    if solv(c) = false then return explored
    result := explored
    if G ≡ □ then success(c)
    else
        if ssolv(c, U(D)) = (unknown, _, _) then return delayed
        for i := 1 to ⟨G ❙ c⟩.nochildren
            if visit-fail(⟨G ❙ c⟩.child[i], D ⇒ ⟨G ❙ c⟩.child[i], U) = delayed then
                result := delayed
    return result
```

The second argument D holds the derivation from root to the node being visited, while the third argument U gives the usage strategy for the stochastic solver. In effect the stochastic solver $ssolv$ is used like the constraint solver $solv$, but an $unknown$ answer is treated as $false$. The result of a call to visit-fail is either $explored$ meaning that the subtree has been entirely explored, or $delayed$ meaning that some nodes in the subtree were not explored. This information can be used to build more complicated search strategies.

4.2 Explore Promising Children First

A stochastic solver is able to give a solution to the current constraints if it succeeds. This is contrary to many CLP solvers: when the solvers answer $true$, they may only have a guarantee that a solution exists but not hold the solution explicitly. And for many problems a stochastic solver can find a solution where the CLP solver just returns $unknown$. Given we have a solution to the constraints in the state, how can we use this to improve search?

In general a *lookahead* process is defined by giving an *evaluation function* eval which takes a state and a solution and returns a ranking (the smaller the better) of how "promising" the state is to explore. At each state if the stochastic solver determines a solution then the children of the state are placed in a *priority queue* depending on their evaluated ranking. A simple evaluation function eval is defined as follows. Given a solution θ to c there are three simple relationships between θ and a constraint occurring in a child node $c \wedge c'$:

(a) *solution*, if θ is a solution of $c \wedge c'$ ($\mathcal{A} \models_\theta c \wedge c'$);
(b) *not a solution*, if θ makes $c \wedge c'$ false ($\mathcal{A} \models_\theta \neg(c \wedge c')$); and
(c) *possibly a solution*, otherwise. (Note this requires that c' has variables not in c).

It is straightforward to determine the above relationships given the valuation θ and the constraint c'. An obvious evaluation function is to rank those children in case (a) as 1, those in case (c) as 2, and those in case (b) as 3. The search will then visit children for

which we have a solution before others. We shall give another example of an evaluation function in Section 5. The following algorithm enforces this behavior.

```
visit-look(⟨G ▌ c⟩, D, U)
    if solv(c) = false then return
    if G ≡ □ then success(c)
    else
        Q := emptyque()
        if ssolv(c, U(D)) = (true, θ, _) then
            for i := 1 to ⟨G ▌ c⟩.nochildren
                s := ⟨G ▌ c⟩.child[i]
                Q := enque(Q, [s, eval(s, θ)])
        else
            for i := 1 to ⟨G ▌ c⟩.nochildren
                s := ⟨G ▌ c⟩.child[i]
                Q := enque(Q, [s, i])
        while ¬ empty(Q)
            ([s, w], Q) := deque(Q)
            visit-look(s, D ⇒ s, U)
    return
```

The stochastic solver is invoked at each node of the derivation tree. If it returns a solution then each child s is evaluated (using eval) to determine a weight w. The children are then placed in a priority queue Q, together with their weight $[s, w]$. If no solution is available the children are just ordered using their child number. The abstract operations for the priority queue are emptyque, empty, enque and deque, the functions of which are self-explanatory. Execution proceeds by removing the lowest weight child from the priority queue and visiting it until the queue becomes empty.

The advantage of lookahead is that for little overhead we may be able to use the stochastic solver to direct the search first to subtrees that are likely to have answer nodes, because we have some handle on at least one answer of the current constraint. This strategy is useful for applications that need to find the first answer to a goal quickly. Execution of the stochastic solver is not free. There is no reason to exercise the stochastic solver and/or order the children at every node of the derivation tree. It is straight-forward to generalize visit-look to handle simple program annotations, by which programmers specify when to invoke the stochastic solver and when to rank child branches.

5 Experimental Results

In this section we detail a number of preliminary experiments using the ECLiPSe [3] system augmented with a GENET stochastic solver [1] to illustrate the benefits that are possible. The two guiding principles for using a stochastic solver to control search have already been shown to be beneficial. The visit-fail procedure is equivalent to an approach of Stuckey and Tam described in [11], while visit-look is a generalization of an integration framework by Lee, Leung and Won [7]. In the following we give preliminary results looking at extensions suggested by the generality of the framework described herein. Timings given are median execution time in CPU seconds.

Disjunctive Graph Coloring Problems

The disjunctive graph coloring problem is to color a graph with a number of hyper-arcs connecting nodes. A hyper-arc $\{(i_1, j_1), \ldots, (i_n, j_n)\}$ specifies that at least one of the pairs of nodes (i_k, j_k) ($1 \leq k \leq n$) must be colored differently. The disjunctive graph coloring problem has wide applicability in time-tabling, scheduling and production planning. By modeling disjunction as choices, we use a variant of visit-fail to delay visiting unpromising branches in the search process.

Consider the execution of a stochastic solver. In order to return $unknown$, the solver must use up all its resources. Hence failure is, in general, expensive to detect using a stochastic solver. This is where lookahead can help. We can use the valuation returned by the stochastic solver to choose to explore branches which are *less likely to fail*. This helps to possibly avoid some of the overhead of failure in the search. The visit-fail-look search procedure is a simple combination of visit-fail and visit-look.

The evaluation function eval used in visit-fail-look is defined as follows. The first child (as ordered in the child node array) entailed by the valuation of the stochastic solver, which may not exist, is given the highest priority. The rest of the children is assigned a lower priority. We can rephrase the definition in the context of disjunctive graph coloring problems: if one pair of a hyper-arc is already colored differently in the solution held in the stochastic solver before the hyper-arc is considered, then we should explore the child corresponding to this pair first. In this case the current solution continues to be a solution of the problem with the added constraint, and no further constraint solving is required.

Nodes	Colors	visit	visit-fail	visit-fail-look
10	3	19.96 (6428)	0.975 (4)	0.035 (0)
20	4	1362 (65535)	0.405 (1)	0.055 (0)
30	5	0.060 (0)	0.110 (0)	0.105 (0)
40	5	0.085 (0)	0.110 (0)	0.155 (0)
50	5	0.105 (0)	0.180 (0)	0.185 (0)

The table shows execution times and, in parentheses, the median number of backtracks required to find a solution. Where significant backtracking is required in the vanilla search procedure visit, using the stochastic solver is advantageous, and in these cases, using lookahead together with the stochastic solver further improves the search behavior.

Job-Shop Scheduling Problems

An $n \times m$ job shop scheduling problem consists of n jobs, each made up of a sequence of m tasks with some duration. Each task has to be performed on one of m machines. The aim is to find a start time for every task such that every task starts after its immediate predecessor has completed, and no two tasks run on the same machine concurrently. A straightforward encoding of the problem consists of a variable T_{ij} for the start time of each task j of job i. The precedence relationships are encoded as a conjunction of constraints of the form $T_{ij} + d_{ij} \leq T_{ij+1}$, where d_{ij} is the duration of task ij. The machine usage constraints are encoded as a search ensuring that for any two tasks ij and kl requiring the same machine, one is after or before the other: $T_{ij} + d_{ij} \leq T_{kl}$ or $T_{kl} + d_{kl} \leq T_{ij}$. Usually the full problem is a minimization task. Here we use the closely related problem of finding a solution within a certain overall time limit.

Primitive constraints occurring in the above formulation of job-shop scheduling are very simple and are completely handled by a propagation-based solver. There is no advantage in using a stochastic solver to delay visiting unpromising branches. However it may be worthwhile in controlling the search for a solution using lookahead. The strategy we use is to invoke the stochastic solver *just once*, after the precedence constraints are added, to determine a (random) valuation $\{T_{ij} \mapsto t_{ij}\}$, which is just a solution to the precedence constraints but not yet a solution to the entire problem. The valuation is then used to control the search: for each choice in the machine usage constraints we calculate $t_{ij} + d_{ij} - t_{kl}$ and $t_{kl} + d_{kl} - t_{ij}$. The eval function gives a higher priority to the choice closest to the "solution" of the precedence constraints.

The table below shows results for a number of randomly generated 10×5 (j2 and j4) and 10×10 (j1 and j3) job-shop scheduling problems for three kinds of execution: visit, visit-look choosing the choice closest to the solution of the precedence constraints, and anti-visit-look choosing the opposite choice!

Problems	visit	visit-look	anti-visit-look
j1	1.15	53.37	59.34
j2	1.69	6.36	4.50
j3	591.77	56.94	54.39
j4	15.80	573.72	4.46

For the small problems the ordinary depth-first left-right search can quickly find a solution. In these problems the overhead of calling the stochastic solver even once swamps the execution time and no benefit is found. For problems where the ordinary search is poor, the search directed by a solution to the precedence constraints can benefit. Surprisingly, as shown by j4, the wrong choice can be substantially faster. The visit procedure in a sense performs a search ordered by the (random) order in which tasks are specified. We would expect that visit-look would perform better, since it uses some knowledge of the problem (a solution of the precedence constraints) to guide the search. Why anti-visit-look can be beneficial is that it quickly drives the search towards failure. Hence the leftmost branches tend to be short. Still it is difficult to see why it should be better than visit-look which would not need to visit these search paths at all if there is a solution "close" to the solution of the precedence constraints.

6 Concluding Remarks

An important feature of logic programming is its strong theoretical background. It is easy to check that standard CLP semantics are retained in our framework for two reasons. First, the shape of a derivation tree is completely determined by the underlying resolution principle used in a particular CLP system. Second, the stochastic solver is used simply to guide the search of the derivation tree. Therefore, the soundness and completeness of the underlying resolution procedure of the CLP system remain valid. Note that visit-fail and its variants suffer from a form of search-incompleteness since unknown is interpreted as failure during search. The term "completeness" is overloaded in CLP semantics and in search. Completeness in CLP semantics concerns the existence of successful derivations for logical consequences; whereas search-completeness concerns whether the entire search space is explored eventually.

The integration framework contributes to CLP by enabling CLP systems to tackle some difficult problems; hence reinforcing the practicality of declarative languages. The integration contributes to the stochastic community by offering a simple, expressive, and flexible programming language frontend. The simplicity and generality of the integration scheme based on guiding search opens up myriad possibilities of different level of cooperation between the principal CLP solver and the stochastic solver. In fact, we demonstrate some of these possibilities by simply combining and modifying the basic search procedures described. In a sense we have a mechanism for combining the relaxation based search of stochastic methods, with enumerative search used in CLP.

Feasibility studies of various instances of the framework confirm empirically that the instances are useful on some difficult benchmarks. Although we deal with programs in the integer domain throughout the paper, our integration framework is applicable to CLP systems over any constraint domain, as long as a stochastic solver for that domain is available. A longer and more detailed version of this paper appears as [6].

References

1. 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.
2. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.
3. ECRC Common Logic Programming System (ECLiPSe) version 3.5 User Manual.
4. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, Volume 19 / 20, May 1994.
5. J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, **14**(3) 339–395, July 1992.
6. J.H.M. Lee, H.F. Leung, P.J. Stuckey, V.W.L. Tam, and H.W. Won, Using Stochastic Methods to Guide Search in CLP: a Preliminary Report. Technical Report, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, 1996.
7. J.H.M. Lee, H.F. Leung and H.W. Won. Integration of Stochastic Solvers into Constraint Logic Programming. Technical Report, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, 1996.
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. P. Stuckey and V. Tam, Models for using stochastic solvers in constraint logic programming. To appear in *Proceedings of the Eighth Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, Aachen, Germany, September 1996.
12. P. Stuckey and V. Tam, Extending GENET with Lazy Arc Consistency. Technical Report 96/8, Department of Computer Science, University of Melbourne, Parkville, Australia, 1996.

This article was processed using the LaTeX macro package with LLNCS style