Algorithms for Stochastic CSPs

Thanasis Balafoutis and Kostas Stergiou

Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece

Abstract. The Stochastic CSP (SCSP) is a framework recently introduced by Walsh to capture combinatorial decision problems that involve uncertainty and probabilities. The SCSP extends the classical CSP by including both decision variables, that an agent can set, and stochastic variables that follow a probability distribution and can model uncertain events beyond the agent's control. So far, two approaches to solving SCSPs have been proposed; backtracking-based procedures that extend standard methods from CSPs, and scenario-based methods that solve SCSPs by reducing them to a sequence of CSPs. In this paper we further investigate the former approach. We first identify and correct a flaw in the forward checking (FC) procedure proposed by Walsh. We also extend FC to better take advantage of probabilities and thus achieve stronger pruning. Then we define arc consistency for SCSPs and introduce an arc consistency algorithm that can handle constraints of any arity.

1 Introduction

Representation and reasoning with uncertainty is an important issue in constraint programming since uncertainty is inherent in many real combinatorial problems. To model such problems, many extensions of the classical CSP have been proposed (see [9] for a detailed review). The Stochastic CSP (SCSP) is a framework that can be used to model combinatorial decision problems involving uncertainty and probabilities recently introduced by Walsh [10]. The SCSP extends the classical CSP by including both decision variables, that an agent can set, and stochastic variables that follow a probability distribution and can model uncertain events beyond the agent's control. The SCSP framework is inspired by the stochastic satisfiability problem [6] and combines some of the best features of traditional constraint satisfaction, stochastic integer programming, and stochastic satisfiability.

The expressional power of the SCSP can help us model situations where there are probabilistic estimations about various uncertain actions and events, such as stock market prices, user preferences, product demands, weather conditions, etc. For example, in industrial planning and scheduling, we need to cope with uncertainty in future product demands. As a second example, interactive configuration requires us to anticipate variability in the users' preferences. As a final example, when investing in the stock market, we must deal with uncertainty in the future price of stocks.

F. Benhamou (Ed.): CP 2006, LNCS 4204, pp. 44–58, 2006.

SCSPs have recently been introduced and only a few solution methods have been proposed. In the initial paper, Walsh described a chronological backtracking and a forward checking procedure for binary problems [10]. These are extensions of the corresponding algorithms for CSPs that explore the space of policies in a SCSP. Alternatively, scenario-based methods, which solve a SCSP by reducing it to a sequence of CSPs, were introduced in [7]. This approach carries certain advantages compared to algorithms that operate on the space of policies. Most significantly, it can exploit existing advanced CSP solvers, without requiring the implementation of (potentially complicated) specialized search and propagation techniques. As a consequence, this approach is not limited to binary problems. However, the number of scenarios in a SCSP grows exponentially with the number of stages in the problem. Therefore, the scenario-based approach may not be applicable in problems with many stochastic variables and many stages.

In this paper we develop algorithms for SCSPs following the initially proposed approach based on the exploration of the policy space. We first identify and correct a flaw in the forward checking (FC) procedure proposed by Walsh. We also extend FC to take better advantage of probabilities and thus achieve stronger pruning. Then we define arc consistency (AC) for SCSPs and introduce an AC algorithm that can handle constraints of any arity. This allows us to implement a MAC algorithm that can operate on non-binary problems. Finally, we present some preliminary experimental results.

2 Stochastic Constraint Satisfaction

In this section we review the necessary definitions on SCSPs given in [10] and [7]. A *stochastic constraint satisfaction problem* (SCSP) is a 6-tuple$< X, S, D, P, C, \Theta >$ where X is a sequence of n variables, S is the subset of X which are stochastic variables, D is a mapping from X to domains, P is a mapping from S to probability distributions for the domains of the stochastic variables, C is a set of e constraints over X, and Θ is a mapping from constraints to threshold probabilities in the interval $[0, 1]$. Each constraint is defined by a subset of the variables in X and an, extensionally or intensionally specified, relation giving the allowed tuples of values for the variables in the constraint. A hard constraint, which must be always satisfied, has an associated threshold 1, while a "chance constraint" c_i, which may only be satisfied in some of the possible worlds, is associated with a threshold $\theta_i \in [0, 1]$. This means that the constraint must be satisfied in at least a fraction θ_i of the worlds.

For the purposes of this paper, we will follow [10] and assume that the problem consists only of a single global chance constraint which is the conjunction of all constraints in the problem. This global constraint must be satisfied in at least a fraction θ of the possible worlds. We will also assume that the stochastic variables are independent (as in [10]). This assumption limits the applicability of the SCSP framework but it can be lifted, as in other frameworks for uncertainty handling, such as *fuzzy* and *possibilistic* CSPs [4].

We will sometimes denote decision variables by xd_i and stochastic variables by xs_i. Accordingly, the sets of decision and stochastic variables in the problem will be denoted by Xd and Xs respectively. The domain of a variable x_i will be denoted by $D(x_i)$, and the variables that participate in a constraint c_i will be denoted by $var(c_i)$. We assume that in each constraint c_i the variables in $var(c_i)$ are sorted according to their order in X.

The backtracking algorithms of [10] explore the space of policies in a SCSP. A *policy* is a tree with nodes labelled with value assignments to variables, starting with the values of the first variable in X labelling the children of the root, and ending with the values of the last variable in X labelling the leaves. A variable whose next variable in X is a decision one corresponds to a node with a single child, while a variable whose next variable in X is a stochastic one corresponds to a node that has one child for every possible value of the following stochastic variable. Leaf nodes take value 1 if the assignment of values to variables along the path to the root satisfies all the constraints and 0 otherwise. Each path to a leaf node in a policy represents a different possible *scenario* (set of values for the stochastic variables) and the values given to decision variables in this scenario. Each scenario has an associated probability; if xs_i is the i-th stochastic variable in a path to the root, v_i is the value given to xs_i in this scenario, and $\text{prob}(xs_i \leftarrow v_i)$ is the probability that $xs_i = v_i$, then the probability of this scenario is: $\prod_i \text{prob}(xs_i \leftarrow v_i)$.

The satisfaction of a policy is the sum of the leaf values weighted by their probabilities. A policy satisfies the constraints iff its satisfaction is at least θ. In this case we say that the policy is *satisfying*. A SCSP is satisfiable iff it has a satisfying policy. The optimal satisfaction of a SCSP is the maximum satisfaction of all policies. Given a SCSP, two basic reasoning tasks are to determine if the satisfaction is at least θ and to determine the maximum satisfaction.

The simplest possible SCSP is a one-stage SCSP in which all of the decision variables are set before the stochastic variables. This models situations in which we must act now, trying to plan our actions in such a way that the constraints are satisfied (as much as possible) for whatever outcome of the later uncertain events. Alternatively, we may demand that the stochastic variables are set before the decision variables. A one stage SCSP is satisfiable iff there exist values for the decision variables so that, given random values for the stochastic variables, the constraints are satisfied in at least the given fraction of worlds. In a two stage SCSP, there are two sets (blocks) of decision variables, Xd_1 and Xd_2, and two sets of stochastic variables, Xs_1 and Xs_2. The aim is to find values for the variables in Xd_1, so that given random values for Xs_1, we can find values for Xd_2, so that given random values for Xs_2, the constraints are satisfied in at least the given fraction of worlds. An m stage SCSP is defined in an analogous way to one and two stage SCSPs.

SCSPs are closely related to *quantified* CSPs (QCSPs). A QCSP can be viewed as a SCSP where existential and universal variables correspond to decision and stochastic variables, respectively. In such a SCSP, all values of the stochastic variables have equal probability and the satisfaction threshold is 1.

3 Forward Checking

Forward Checking for SCSPs was introduced in [10] as an extension of the corresponding algorithm for CSPs. We first review this algorithm and show that it suffers from a flaw. We then show how this flaw can be corrected and how FC can be enhanced to achieve stronger pruning.

Figure 1 depicts the `FC` procedure of [10]. `FC` instantiates the variables in the order they appear in X. On meeting a decision variable, `FC` tries each value in its domain in turn. The maximum value is returned to the previous recursive call. On meeting a stochastic variable, `FC` tries each value in turn, and returns the sum of all the answers to the subproblems weighted by the probabilities of their occurrence. On instantiating a decision or stochastic variable, `FC` checks forward and prunes values from the domains of future variables which break constraints. If the instantiation of a decision or stochastic variable breaks a constraint, the algorithm returns 0. If all variables are instantiated without breaking any constraint, `FC` returns 1.

Procedure `FC`(i, θ_l, θ_h)
if $i > n$ **then** return 1
$\theta := 0$
for each $v_j \in D(x_i)$
  if prune$(i, j) = 0$ **then**
    if check$(x_i \leftarrow v_j, \theta_l)$ **then**
      if $x_i \in Xs$ **then**
        $p :=$ prob$(x_i \leftarrow v_j)$
        $q_i := q_i - p$
        θ:=θ+p×FC(i+1,(θ_l-θ-q_i)/p,(θ_h-θ)/p)
        `restore`(i)
        if $\theta + q_i < \theta_l$ **then** return θ
        if $\theta > \theta_h$ **then** return θ
      else
        $\theta := \max(\theta, FC(i+1, \max(\theta, \theta_l), \theta_h))$
        `restore`(i)
        if $\theta > \theta_h$ **then** return θ
    else `restore`(i)
return θ

function `check`$(x_i \leftarrow v_j, \theta_l)$
 for $k := i + 1$ to n
  dwo := true
  for each $v_l \in D(x_k)$
   if prune$(k, l) = 0$ **then**
    if inconsistent$(x_i \leftarrow v_j, x_k \leftarrow v_l)$ **then**
     prune$(k, l) := i$
     if $x_k \in Xs$ **then**
      $q_k := q_k$ - prob$(x_k \leftarrow v_l)$
      if $q_k < \theta_l$ **then** return false
    else dwo := false
  if dwo **then** return false
 return true

Fig. 1. The `FC` algorithm of [10]

In Figure 1, a 2-dimensional array prune(i, j) is used to record the depth at which the value $v_j \in D(x_i)$ is removed by forward checking. Each stochastic variable xs_i has an upper bound, q_i, on the probability that the values left in $D(xs_i)$ can contribute to a solution. This is initially set to 1. The upper and lower bounds, θ_h and θ_l are used to prune search. By setting $\theta_l = \theta_h = \theta$, we can determine if the optimal satisfaction is at least θ. By setting $\theta_l = 0$ and $\theta_h = 1$, we can determine the optimal satisfaction.

The calculation of these bounds in recursive calls is done as follows. Suppose that the current assignment to a stochastic variable returns a satisfaction of θ_0. We can ignore other values for this variable if $\theta + p\times\ \theta_0 \geq \theta_h$. That is, if $\theta_0 \geq (\theta_h - \theta)/p$. This gives the upper bound in the recursive call to `FC` on a stochastic variable. Alternatively, we cannot hope to satisfy the constraints adequately if $\theta + p\times\ \theta_0 + q_i \leq \theta_l$ as q_i is the maximum that the remaining values can contribute to the satisfaction. That is, if $\theta_0 \leq (\theta_l - \theta - q_i)/p$. This gives the lower bound in the recursive call to `FC` on a stochastic variable. Finally, suppose that the current assignment to a decision variable returns a satisfaction of θ. If this is more than θ_l, then any other values must exceed θ to be part of a better policy. Hence, we can replace the lower bound in the recursive call to `FC` on a decision variable by $\max(\theta, \theta_l)$. Procedure `restore`, which is not shown, is called when a tried assignment is rejected and when a backtrack occurs, to restore values that have been removed from future variables and reset the value of q_i for stochastic variables.

Checking forwards fails if any variable has a domain wipeout (dwo), or (crucially) if a stochastic variable has so many values removed that we cannot hope to satisfy the constraints. When forward checking removes some value v_j from xs_i, `FC` reduces q_i by prob$(xs_i \leftarrow v_j)$, the probability that xs_i takes the value v_j. This reduction on q_i is undone on backtracking. If `FC` ever reduces q_i to less than θ_l, it backtracks as it is impossible to set xs_i and satisfy the constraints adequately.

3.1 A Flaw in FC

As the next example shows, this last claim can be problematic. When the current variable is a stochastic one, there are cases where, even if q_i is reduced to less than θ_l, the algorithm should continue going forward instead of backtracking because the satisfaction of the future subproblem may contribute to the total satisfaction. The example considers the case where we look for the maximum satisfaction.

Example 1. Consider a problem consisting of one decision variable xd_1 and two stochastic variables xs_2, xs_3, all with $\{0, 1\}$ domains. The probabilities of the values are shown in Figure 2a where the search tree for the problem is depicted. There is a constraint between xd_1 and xs_2 disallowing the tuple $<xd_1 \leftarrow 0, xs_2 \leftarrow 1>$. There is also a constraint between xs_2 and xs_3 disallowing the tuple $<xs_2 \leftarrow 1, xs_3 \leftarrow 0>$. Assume that we seek the maximum satisfaction of the problem. That is, initially $\theta_l = 0$ and $\theta_h = 1$.

`FC` will first instantiate xd_1 to 0 and forward check this assignment. As a result, value 1 of xs_2 will be deleted and the dashed nodes will be pruned. Then the algorithm will explore the non-pruned subtree below $xd_1 \leftarrow 0$ and eventually will backtrack to xd_1. At this point θ will be 0.5 (i.e. the satisfaction of the explored subtree). Now when `FC` moves forward to instantiate xs_2, θ_l will be set to $\max(\theta_l, \theta) = \max(0, 0.5) = 0.5$. The subtree below $xs_2 \leftarrow 0$, weighted by prob$(xs_2 \leftarrow 0)$, gives 0.5 satisfaction. When assigning 1 to xs_2, `check` will return false because value 0 of xs_3 will be removed and the remaining probability in the domain of xs_3 will be $0.4<\theta_l$. Therefore, FC will backtrack and terminate, incorrectly returning 0.5 as the maximum satisfaction. Clearly, the maximum satisfaction, which is achieved by the policy depicted with bold edges, is 0.7.

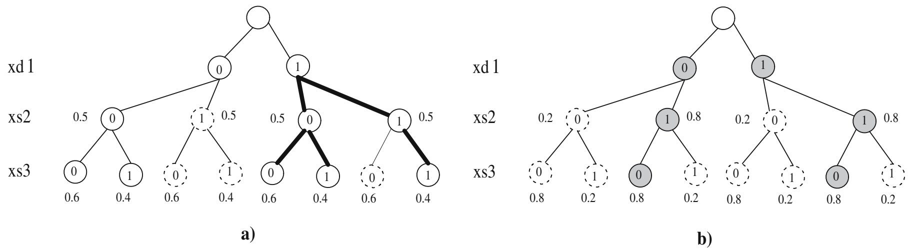

Fig. 2. Search trees of Examples 1 and 2

Function `check` correctly returns a failure when the current variable is a decision one and for some future stochastic variable xs_i forward checking reduces q_i below θ_l. In this case, there is not point in exploring the subtree below the current assignment. However, when the current variable is a stochastic one and for some future stochastic variable, q_i falls below θ_l, it is not certain that the currently explored policy cannot yield satisfaction greater than the threshold. What we need is a way to determine if the maximum satisfaction offered by the current stochastic variable is enough to lift the total satisfaction over the lower satisfaction bound or not. Therefore, we need to take into account the following quantities: 1) the already computed satisfaction of the previously assigned values of the current variable, 2) the maximum satisfaction of the subtree below the current assignment, 3) the sum of the probabilities of the following values of the current variable (i.e. the maximum satisfaction that they can contribute). If the sum of these quantities is lower than θ_l then the current assignment can be safely rejected. Otherwise, we must continue expanding it. This idea is formulated in more detail further below, after we describe a simple way to enhance the pruning power of FC.

3.2 Improving FC

We can save search effort by performing stronger pruning inside function `check`. When making forward checks and removing values from future stochastic variables, the FC algorithm of [10] exploits only a "local" view of the future problem. But as values are removed from future stochastic variables, the maximum possible satisfaction of the current assignment is reduced. FC fails to exploit this because it considers value removals from any future stochastic variable as "independent" of value removals from other future stochastic variables. However, it is possible that enough values are removed from a number of stochastic variables so that the maximum possible satisfaction of the current assignment cannot exceed

θ_l. The maximum possible satisfaction of an assignment v_j to the current variable x_i is equal to $\prod_{s=i+1}^{n} \sum_{t=1}^{|D(x_s)|} \text{prob}(x_s \leftarrow v_t)$ (weighted by the probability of $x_i \leftarrow v_j$ if x_i is stochastic), where only values that have not been pruned are considered. In words, we sum the probabilities of the remaining values for all future stochastic variables and multiply the sums. Before explaining how we can exploit this, we present an example that demonstrates the savings in search effort that can be achieved through such reasoning.

Example 2. Consider a problem consisting of one decision variable xd_1 and two stochastic variables xs_2, xs_3, all with $\{0, 1\}$ domains. The probabilities of the values are shown in Figure 2b where the search tree of the problem is depicted. There is a constraint between xd_1 and xs_2 disallowing the tuples $\langle xd_1 \leftarrow 0, xs_2 \leftarrow 0\rangle$ and $\langle xd_1 \leftarrow 1, xs_2 \leftarrow 0\rangle$. There is also a constraint between xd_1 and xs_3 disallowing the tuples $\langle xd_1 \leftarrow 0, xs_3 \leftarrow 1\rangle$ and $\langle xd_1 \leftarrow 1, xs_3 \leftarrow 1\rangle$. Assume that we are looking for the maximum satisfaction.

`FC` will first instantiate xd_1 to 0 and forward check this assignment. As a result, values 0 and 1 will be removed from the $D(xs_2)$ and $D(xs_3)$ respectively. Since q_2 and q_3 do not fall below θ_l, the algorithm will continue to make the instantiations $xs_2 \leftarrow 1$ and $xs_3 \leftarrow 0$. After backtracking to xd_1, the current satisfaction θ for xd_1 will be 0.64. Now `FC` will instantiate xd_1 to 1, forward check the assignment, remove values 0 and 1 from the domains of xs_2 and xs_3, and proceed to instantiate the stochastic variables. Similarly as before, the satisfaction of assignment $xd_1 \leftarrow 1$ will be 0.64. Therefore, `FC` will return the maximum satisfaction among the values of xd_1, which is 0.64. To find this, `FC` needs to visit six nodes in the search tree (the gray nodes in Figure 2b).

Consider again the point when after the satisfaction of assignment $xd_1 \leftarrow 0$ has been computed, the algorithm instantiates xd_1 to 1. Forward checking removes values 0 and 1 from $D(xs_2)$ and $D(xs_3)$ respectively, and as a result the maximum possible satisfaction of assignment $xd_1 \leftarrow 1$ is equal to $\text{prob}(xs_2 \leftarrow 1) \times \text{prob}(xs_3 \leftarrow 0) = 0.64$. This is not greater than the satisfaction of assignment $xd_1 \leftarrow 0$, and therefore, the algorithm need not proceed to instantiate the stochastic variables. Since there is no other value in $D(xd_1)$, we can determine that the satisfaction of the problem is 0.64. In this way, the problem is solved visiting four instead of six nodes.

Figure 3 depicts the improved `check` function of `FC`. The identified flaw is corrected in lines 10-13 where we differentiate between the case where the current variable is a stochastic one and the case where it is a decision one. In both cases we first compute ζ_i; the maximum satisfaction that the current assignment can yield. This is computed as the product of the sums of probabilities of the values that are left in the domains of the future stochastic variables. In this way we get a better estimation of the maximum satisfaction that the current assignment can provide and the efficiency of the algorithm, compared to the version given in [10], is improved. Note that ζ_i is computed each time `FC` has filtered the domain of a future variable. Alternatively, we can compute it once after `FC` has finished with all future variables. In this case we can save repeating some computations but may perform redundant consistency checks.

function check($x_i \leftarrow v_j, q_i, \theta_l, \theta$)
$q_i := q_i$ - prob($x_i \leftarrow v_j$)
for $k := i + 1$ to n
  dwo := true
  for each $v_l \in D(x_k)$
    if prune$(k, l) = 0$ **then**
      if inconsistent($x_i \leftarrow v_j, x_k \leftarrow v_l$) **then**
        prune$(k, l) := i$
        if $x_k \in Xs$ **then**
          $\zeta_i := \prod_{s=i+1}^{n} \sum_{t=1}^{|D(x_s)|}$ prob($x_s \leftarrow v_t$)
          if $x_i \in Xs$ **then**
            if $\zeta_i \times prob(x_i \leftarrow v_j) + \theta + q_i < \theta_l$ **then** return false
          else
            if $\zeta_i < \theta_l$ **then** return false
      else dwo := false
  if dwo **then** return false
return true

Fig. 3. The improved check function of FC

If the current variable is a decision one and ζ_i falls below θ_l then we return false as is is not possible to extend the current assignment in a way that the threshold is satisfied. If the current variable is a stochastic one then we multiply ζ_i with the probability of the current assignment, add the satisfaction (θ) yielded by previously tried assignments to the current variable, add the sum of probabilities (q_i) of the remaining values for the current variable, and compare the resulting quantity with θ_l. If it is lower then we return fail because we know that there is no way to extend the current assignment, so that the threshold is satisfied, even if the current assignment and the remaining assignments to the current variable yield the maximum possible satisfaction.

4 Arc Consistency

Arc consistency (AC) is an important concept in CSPs since it is the basis of constraint propagation in most CSP solvers. In this section we first define AC for SCSPs and then describe an AC algorithm for SCSPs that can handle constraints of any arity. We show that, apart from the case of domain wipeout, failure can also be determined when enough values are removed from stochastic variables. We also introduce a specialized pruning rule that can be used to remove values from certain decision variables.

Before defining AC, we give a definition of consistency for values of decision variables. To do this, we adjust the corresponding definitions for QCSPs given in [2,3]. Intuitively, a value $v \in D(xd_i)$ is inconsistent if the assignment $xd_i \leftarrow v$ cannot participate in any satisfying policy.

Definition 1. A value $v \in D(xd_i)$ is *consistent* iff there is a satisfying policy, in which one scenario at least, includes the assignment $x_i \leftarrow v$.

Given the above definition, determining the consistency of a value involves finding all solutions (satisfying policies) to a SCSP. The definition of AC and the development of relevant filtering algorithms can hopefully help us perform pruning by local reasoning. We first give some necessary notation. Given a SCSP $A =< X, S, D, P, C, \Theta >$ we denote by A_{c_j} the SCSP in which only one constraint $c_j \in C$ is considered, i.e. the SCSP $< X, S, D, P, c_j, \theta_j >$. $\tau[x_i]$ gives the value that variable x_i takes in tuple τ. A tuple of assignments τ is *valid* if none of the values in τ has been removed from the domain of the corresponding variable. A tuple τ of a constraint c_j supports a value $v \in x_i$ iff $\tau[x_i] = v$, τ is valid, and τ is allowed by c_j.

Definition 2. A value $v \in D(xd_i)$ is *arc consistent* iff, for every constraint $c_j \in C$, v is consistent in A_{c_j}. A value $v \in D(xs_i)$ is arc consistent iff, for every constraint $c_j \in C$, there is a tuple that supports it. A SCSP is arc consistent iff all values of all variables are arc consistent.

Note that we differentiate between decision and stochastic variables. The definition of AC for values of decision variables subsumes the classical AC definition (which is used for values of stochastic variables). The above definition covers the general case where they may be multiple chance constraints. But in the problems considered here, where there is only a single global chance constraint, determining if a given SCSP is AC is a task just as hard as solving it. This is analogous to achieving AC in a classical CSP where all constraints are combined in a conjunction.

In the following we describe an algorithm that is not complete (i.e. it does not compute the AC-closure of a given SCSP) but can achieve pruning of some arc inconsistent values through local reasoning, and therefore in some cases detect arc inconsistency. In addition, the algorithm can determine failure if the maximum possible satisfaction falls below θ_l because of deletions from the domains of stochastic variables. The AC algorithm we use as basis is GAC2001/3.1 [1]. This is a coarse-grained (G)AC algorithm that does not require complicated data structures, while it achieves an optimal worst-case time complexity in its binary version. In addition to these features, GAC2001/3.1 facilitates the implementation of a specialized pruning rule that can remove arc inconsistent values from certain decision variables through local reasoning. The motivation for this rule is demonstrated in the following example.

Example 3. Consider a problem consisting of two decision variables xd_1 and xd_2 and two stochastic variables xs_3, xs_4, all with $\{0, 1\}$ domains. The probabilities of the values are shown in Figure 4 where the search tree of the problem is depicted. There is a ternary constraint c_1 with $var(c_1) = \{xd_2, xs_3, xs_4\}$ which disallows tuples $<xd_2 \leftarrow 0, xs_3 \leftarrow 0, xs_4 \leftarrow 0>$ and $<xd_2 \leftarrow 0, xs_3 \leftarrow 1, xs_3 \leftarrow 1>$. Assume that we are trying to determine if the satisfaction is at least 0.6.

It is easy to see that any policy which includes assignment $xd_2 \leftarrow 0$ cannot achieve more than 0.5 satisfaction since assigning 0 to xd_2 leaves $\{xs_3 \leftarrow 0, xs_4 \leftarrow 1\}$ and $\{xs_3 \leftarrow 1, xs_4 \leftarrow 0\}$ as the only possible sets of assignments for

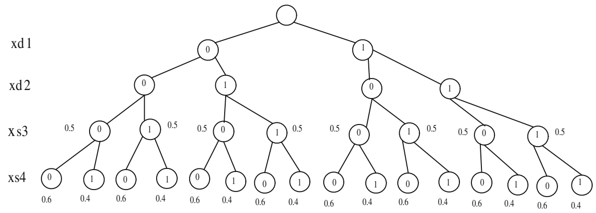

Fig. 4. Search tree of Example 3

variables xs_3 and xs_4, and these together yield 0.5 satisfaction. Therefore, we can safely prune the search tree by deleting value 0 of xd_2 prior to search.

We can generalize the idea illustrated in the example to the case where we reach a block of consecutive decision variables during search. Then if we identify certain values of these variables that, if assigned, result in policies which cannot yield satisfaction more than the current lower bound θ_l, then we know that these values are arc inconsistent and can thus prune them. We have incorporated this idea in the AC algorithm described below. Similar reasoning can be applied on decision variables further down in the variable sequence (i.e. not in the current block). However, identifying arc inconsistent values for such variables is an expensive process since it requires search.

Our algorithm for arc consistency in SCSPs is shown in Figure 5. Before explaining the algorithm we give some necessary notation and definitions.

- We assume that the tuples in each constraint are ordered according to the lexicographic ordering. In the while loop of line 26, NIL denotes that all tuples in a constraint have been searched.
- As in GAC2001/3.1, Last$((x_i, v), c_j)$ is the most recently discovered tuple in c_j that supports value $v \in D(x_i)$, where $x_i \in var(c_j)$. Initially, each Last$((x_i, v), c_j)$ is set to 0. c_var denotes the currently instantiated variable. If the algorithm is used for preprocessing, c_var is 0.
- When we say that "variable x_i belongs to the current stage in X" we mean that c_var is a decision variable and x_i belongs to the same block of variables as c_var. In case `Stochastic_AC` is used for preprocessing, we say "variable x_i belongs to the first stage in X" meaning that the first block of variables in X is composed of decision variables and x_i belongs to this block.
- $\theta_{(x_i,v),c_j}$ holds the maximum satisfaction that can be achieved by the possible assignments of the stochastic variables after decision variable x_i in $var(c_j)$ if value v is given to x_i. This is calculated by summing the probabilities of the tuples that support $x_i \leftarrow v$ in c_j. In this context, the probability of a tuple $\tau =<\ldots, x_i \leftarrow v, \ldots>$ is the product of the probabilities of the values that the stochastic variables **after** x_i take in τ.

`Stochastic_AC` uses a queue (or stack) of variable-constraint pairs. Essentially it operates in a similar way to GAC2001/3.1 with additional fail detection and

function `Stochastic_AC`(c_var, θ_l)
  $Q \leftarrow \{x_i, c_j | c_j \in C, x_i \in var(c_j)\}$
  if $c\_var = 0$ **then**
    for each $(x_i, c_j) | x_i \in var(c_j), x_i \in Xd$ and $\exists x_m \in Xs, m > i$ and $x_m \in var(c_j)$
      for each $v \in D(x_i)$
        $\theta_{(x_i,v),c_j} \leftarrow 0$
        for τ=Last$((x_i, v), c_j)$ to last tuple in c_j
          if $\tau[x_i] = v$ **and** τ is valid **and** τ is allowed by c_j **then**
            $\theta_\tau \leftarrow \prod_{s=x_i+1}^{|var(c_j)|}$ prob$(x_s \leftarrow \tau[x_s])$
            $\theta_{(x_i,v),c_j} \leftarrow \theta_{(x_i,v),c_j} + \theta_\tau$
        if x_i belongs to the first stage in X and $\theta_{(x_i,v),c_j} < \theta_l$ **then**
          remove v from $D(x_i)$
          if $D(x_i)$ is wiped out **then return** false
  while Q not empty **do**
    select and remove a pair (x_i, c_j) from Q
    fail $\leftarrow$ false
    if `Revise`($x_i, c_j, c\_var, \theta_l$,fail)
      if fail=true **or** a domain is wiped out **then return** false
      $Q \leftarrow Q \cap \{(x_k, c_m | c_m \in C, x_i, x_k \in var(c_m), m \neq j, i \neq k\}$
  return true

function `Revise`($x_i, c_j, c\_var, \theta_l$,fail)
  DELETION $\leftarrow$ FALSE
  for each value $v \in D(x_i)$
    if Last$((x_i, v), c_j)$ is not valid **then**
      if $x_i \in Xd$ and $\exists x_m \in Xs, m > i$ and $x_m \in var(c_j)$ **then**
        $\theta_{(x_i,v),c_j} \leftarrow \theta_{(x_i,v),c_j} - \theta_{Last((x_i,v),c_j)}$
      $\tau \leftarrow$ next tuple in the lexicographic ordering
      while $\tau \neq NIL$
        if $\tau[x_i] = v$ **and** τ is allowed by c_j **then**
          if τ is valid **then break**
          else if $x_i \in Xd$ and $\exists x_m \in Xs, m > i$ and $x_m \in var(c_j)$ **then**
            $\theta_{(x_i,v),c_j} \leftarrow \theta_{(x_i,v),c_j} - \theta_\tau$
        $\tau \leftarrow$ next tuple in the lexicographic ordering
      if $x_i, c\_var \in Xd$ and x_i belongs to the current stage in X and $\theta_{(x_i,v),c_j} < \theta_l$ **then**
        remove v from $D(x_i)$
        DELETION $\leftarrow$ TRUE
      else if $\tau \neq NIL$ **then**
        Last$((x_i, v), c_j) \leftarrow \tau$
      else
        remove v from $D(x_i)$
        if $x_i \in Xs$
          $\zeta_i := \prod_{s=c\_var+1}^{n} \sum_{t=1}^{|D(x_s)|}$ prob$(x_s \leftarrow v_t)$
          if $\zeta_i < \theta_l$ **then**
            fail $\leftarrow$ true
            return true
        DELETION $\leftarrow$ TRUE
  return DELETION

Fig. 5. An arc consistency algorithm for stochastic CSPs

pruning operations to account for the stochastic nature of the problem. Initially, all variable-constraint pairs (x_i, c_j), where $x_i \in var(c_j)$, are inserted in Q. Then, a preprocessing step, which implements the pruning rule described above, takes place (lines 2-12). For every decision variable x_i and any constraint c_j where x_i participates, such that the constraint includes stochastic variables after x_i in $vars(c_j)$ (this is tested in line 3), we iterate through the available values in $D(x_i)$. For each such value v we compute the maximum satisfaction $\theta_{(x_i,v),c_j}$ that the stochastic variables after x_i in $vars(c_j)$ can yield, under the assumption that x_i is given value v. This is computed as the sum of satisfaction for all sub-tuples that support $x_i \leftarrow v$ in c_j. The satisfaction of a sub-tuple is simply the product of probabilities for the values of the stochastic variables after $x_i \leftarrow v$ in the tuple (line 8). In case x_i belongs to the first stage in the problem and $\theta_{(x_i,v),c_j}$ is less than θ_l then we know that the assignment $x_i \leftarrow v$ cannot be part of a policy with satisfaction greater than θ_l and therefore v is removed from $D(x_i)$. If no domain wipeout is detected then the algorithm proceeds with the main propagation phase.

During this phase pairs (x_i, c_j) are removed from Q and function `Revise` is called to look for supports for the values of x_i in c_j. For each value $v \in D(x_i)$ we first check if $\text{Last}((x_i, v), c_j)$ is still valid. If it is we proceed with the next value. Otherwise we search c_j's tuples until one that supports v is found or there are no more tuples (lines 25-31). In the former case, $\text{Last}((x_i, v), c_j)$ is updated accordingly (line 36). In the latter case, v is removed from $D(x_i)$ (line 38). If x_i is a decision variable then $\theta_{(x_i,v),c_j}$ is reduced while the search for a support in c_j proceeds. This is done as follows: Whenever a tuple τ that was previously a support for $x_i \leftarrow v$ in c_j but is no longer one (because it is no longer valid) is encountered, $\theta_{(x_i,v),c_j}$ is reduced by θ_τ (lines 24,30). As in the preprocessing phase, θ_τ is computed as the product of probabilities for the values of the stochastic variables after $x_i \leftarrow v$ in τ. If $\theta_{(x_i,v),c_j}$ falls below θ_l, the current variable is a decision one and x_i belongs to the same stage as it, then v is removed from $D(x_i)$ (lines 32,33).

If a value of a stochastic variable is removed then we check if the remaining values in the domains of the future stochastic variables can contribute enough to the satisfaction of the problem so that the lower bound is met. This is done in a way similar to the improved function `check` of `FC`. That is, by comparing quantity $\prod_{s=c\_var+1}^{n} \sum_{t=1}^{|D(x_s)|} \text{prob}(x_s \leftarrow v_t)$ to θ_l. If it is lower then the algorithm returns failure as the threshold cannot be met. If this occurs during search then the currently explored policy should be abandoned and a new one should be tried.

The Pruning Rule for Binary Constraints. Pruning of decision variables that belong to the current decision stage can be made stronger when dealing with binary constraints. For each binary constraint c_j, where $var(c_j) = \{xd_i, xs_l\}$, and each value $v \in D(xd_i)$, we can calculate the maximum possible satisfaction of assignment $xd_i \leftarrow v$ on constraint c_j as $\theta_{(xd_i,v),xs_l} = \sum_{t=Last((xd_i,v),xs_l)}^{|D(xs_l)|}$, s.t. t and v are compatible. In this case $\text{Last}((xd_i, v), xs_l)$ is the most recently discovered value in $D(xs_l)$ that supports v. Therefore, the maximum satisfaction of assignment $xd_i \leftarrow v$ is the product of $\theta_{(xd_i,v),xs_l}$ for all constraints c_j, where $var(c_j) = \{xd_i, xs_l\}$ and xs_l is after xd_i in the variable sequence. By comparing this quantity with θ_l (lines 10 and 32), we can exploit the probabilities of future stochastic variables in a more "global" way, as in the enhancement of FC described in Section 3, and thus stronger pruning can be achieved.

Note that a similar, but more involved, enhancement is possible for non-binary constraints but in that case we have to be careful about future stochastic variables that appear in multiple constraints involving xd_i. When calculating the maximum possible satisfaction we have to make sure that the probabilities of the values of each such stochastic variable are taken into account only once. When dealing with binary constraints no such issue arises, assuming that each stochastic variable can participate in at most one constraint with xd_i.

We now analyze the time complexity of algorithm `Stochastic_AC`. We assume that the maximum domain size is D and the maximum constraint arity is k.

Proposition 1. The worst-case time complexity of `Stochastic_AC` is $O(enk^2D^{k+1})$.

Proof. The preprocessing phase of lines 2-12 is executed for decision variables. For every constraint c_j that includes a decision variable x_i and at least one later stochastic variable, we go through all values in $D(x_i)$. For each such value v, we iterate through the, at most, D^{k-1} tuples that include assignment $x_i \leftarrow v$. Assuming that the calculation of the product of probabilities requires $O(k)$ operations, the complexity of the preprocessing phase is $O(eDkD^{k-1}k)$=$O(ek^2D^k)$.

In the main propagation phase there are at most kD calls to `Revise` for any constraint c_j, one for every deletion of a value from the k variables in $var(c_j)$. In the body of `Revise` (called for $x_i \in var(c_j)$) there is a cost of $O(kD^{k-1})$ to search for supporting tuples for the values of x_i (see [1] for details). The complexity of the pruning rule for decision variables is constant. The failure detection process of lines 35-39 costs $O(nD)$ in the worst case. Therefore, the asymptotic cost of one call to `Revise` is $O(kD^{k-1}nD)$=$O(nkD^k)$. Since there can be kD calls to `Revise` for each of the e constraints, and the use of Last$((x_i, v), c_j)$ ensures that in all calls the search for support for $v \in D(x_i)$ on c_j will never check a tuple more than once, the complexity of `Stochastic_AC` is $O(enk^2D^{k+1})$ □

Since the preprocessing phase alone costs $O(ek^2D^k)$, we may want to be selective in the constraints on which the pruning rule is applied, based on properties such as arity and domain size of the variables involved.

The space complexity of the algorithm is determined by the data structures required to store Last$((x_i, v), c_j)$ and $\theta_{(x_i,v),c_j}$. Both need $O(ekD)$ space, so this is the space complexity of `Stochastic_AC`. However, this may rise to $O(enkD)$ when `Stochastic_AC` is maintained during search and no advanced mechanism is used to restore the Last$((x_i, v), c_j)$ and $\theta_{(x_i,v),c_j}$ structures upon failed instantiations and backtracks. This may be too expensive in large problems but it is always possible to reduce the memory requirements by dropping structures Last$((x_i, v), c_j)$ and $\theta_{(x_i,v),c_j}$ and reverting to a (G)AC-3-type of processing.

5 Experiments

We ran some preliminary experiments on randomly generated binary SCSPs. The best algorithm was the improved version of FC coupled with AC preprocessing. AC appears to be advantageous when used for preprocessing, but MAC is slower than FC on these problems. To generate random SCSPs we used a model with four parameters: the number of variables n, the uniform domain size d, the constraint density p (as a fraction of the total possible constraints), and the constraint tightness q (as a fraction of the total possible allowed tuples). The probabilities of the values for the stochastic variables were randomly distributed.

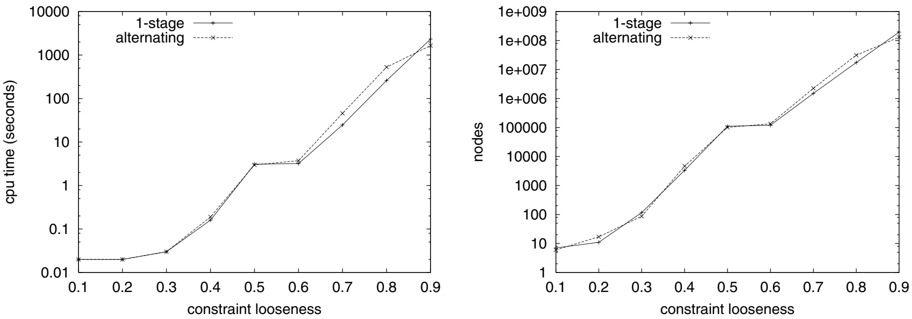

Fig. 6. AC+FC on random problems

Figure 6 demonstrates average results (over 50 instances) from SCSPs where $n = 20$, $d = 3$, $p = 0.1$, and q is varying from 0.1 to 0.9 in steps of 0.1. We show the cpu time (in seconds) and node visits required by FC with AC preprocessing to find the maximum satisfaction. The curve entitled “1-stage” corresponds to one-stage problems where 10 decision variables are followed by 10 stochastic ones, while the curve entitled “alternating” corresponds to problems where there is an alternation of decision and stochastic variables in the sequence. As we can see, both types of problems give similar results. When there are few allowed combinations of values per constraint, problems are easy as the algorithm quickly determines that most policies are not satisfying. When there are many allowed combinations of values per constraint, problems are much harder since there are many satisfying policies, and as a result, a larger part of the search tree must be searched to find the maximum satisfaction.

6 Conclusion and Future Work

We developed algorithms for SCSPs based on the exploration of the policy space. We first identified and corrected a flaw in the FC procedure proposed by Walsh. We also extended FC to better take advantage of probabilities and thus achieve stronger pruning. Then we defined AC for SCSPs and introduced an AC algorithm that can handle constraints of any arity. We ran some preliminary experiments, but further experimentation is necessary to evaluate the practical value for the proposed algorithms.

In the future we intend to further enhance the backtracking algorithms presented here, both in terms of efficiency (e.g. by adding capabilities such as backjumping), and in terms of applicability (e.g. by extending them to deal with multiple chance constraints, joint probabilities for stochastic variables and optimization problems). Also we plan to investigate alternative approaches to solving stochastic CSPs. In particular, techniques adapted from stochastic programming [8] and on-line optimization [5]. Some techniques of this kind have been already successfully developed in [7].

Acknowledgements

This work has been supported by GR Pythagoras grant number 1349 under the Operational Program for Education and Initial Training.

References

1. C. Bessière, J.C. Régin, R. Yap, and Y. Zhang. An Optimal Coarse-grained Arc Consistency Algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.
2. L. Bordeaux, M. Cadoli, and T. Mancini. CSP Properties for Quantified Constraints: Definitions and Complexity. In *Proceedings of AAAI-2005*, pages 360–365, 2005.
3. Lucas Bordeaux. Boolean and interval propagation for quantified constraints. In *Proceedings of the CP-05 Workshop on Quantification in Constraint Programming*, pages 16–30, 2005.
4. D. Dubois, H. Fargier, and H. Prade. Possibility Theory in Constraint Satisfaction Problems: Handling Priority, Preference and Uncertainty. *Applied Intelligence*, 6(4):287–309, 1996.
5. A. Fiat and G.J. Woeginger. *Online Algorithms*, volume 1442. LNCS, 1997.
6. M.L. Littman, S.M. Majercik, and T. Pitassi. Stochastic Boolean Satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, 2000.
7. S. Manandhar, A. Tarim, and T. Walsh. Scenario-based Stochastic Constraint Programming. In *Proceedings of IJCAI-03*, pages 257–262, 2003.
8. A. Ruszczynski and A. Shapiro. *Stochastic Programming, Handbooks in OR/MS*, volume 10. Elsevier Science, 2003.
9. G. Verfaillie and N. Jussien. Constraint Solving in Uncertain and Dynamic Environments: A Survey. *Constraints*, 10(3):253–281, 2005.
10. T. Walsh. Stochastic Constraint Programming. In *Proceedings of ECAI-2002*, pages 111–115, 2002.