A Cost-Based Model and Algorithms for Interleaving Solving and Elicitation of CSPs*

Nic Wilson, Diarmuid Grimes, and Eugene C. Freuder

Cork Constraint Computation Centre
Department of Computer Science
University College Cork, Ireland
n.wilson@4c.ucc.ie, d.grimes@4c.ucc.ie, e.freuder@4c.ucc.ie

Abstract. We consider Constraint Satisfaction Problems in which constraints can be initially incomplete, where it is unknown whether certain tuples satisfy the constraint or not. We assume that we can determine such an unknown tuple, i.e., find out whether this tuple is in the constraint or not, but doing so incurs a known cost, which may vary between tuples. We also assume that we know the probability of an unknown tuple satisfying a constraint. We define algorithms for this problem, based on backtracking search. Specifically, we consider a simple iterative algorithm based on a cost limit on which unknowns may be determined, and a more complex algorithm that delays determining an unknown in order to estimate better whether doing so is worthwhile. We show experimentally that the more sophisticated algorithms can greatly reduce the average cost.

1 Introduction

In Constraint Satisfaction Problems it is usually assumed that the CSP is available before the solving process begins, that is, the elicitation of the problem is completed before we attempt to solve the problem. As discussed in the work on Open Constraints and Interactive CSPs [1,2,3,4,5], there are situations where it can be advantageous and natural to interleave the elicitation and the solving. We may not need all the complete constraints to be available in order for us to find a solution. Furthermore, it may be expensive, in terms of time or other costs, to elicit some constraints or parts of the constraints, for example, in a distributed setting. Performing a constraint check in certain situations can be computationally very expensive. We may need to pay for an option to be available, or for the possibility that it may be available. Some constraints may be related to choices of other agents, which they may be reluctant to divulge because of privacy issues or convenience, and so it could cost us something to find these out. Or they may involve an uncertain parameter, such as the capacity of a resource, and it could be expensive, computationally or otherwise, to determine more certain information about this.

* This material is based upon works supported by the Science Foundation Ireland under Grant No. 05/IN/I886.

C. Bessiere (Ed.): CP 2007, LNCS 4741, pp. 666–680, 2007.

In this paper we consider approaches for solving such partially-specified CSPs which take these costs into account. Constraints may be initially incomplete: it may be unknown whether certain tuples satisfy the constraint or not. It is assumed in our model that we can determine such an unknown tuple, i.e., find out whether this tuple is in the constraint or not, but doing so incurs a known cost, which may vary between tuples. We also assume that we know the probability of an unknown tuple satisfying a constraint. An optimal algorithm for this situation is defined to be one which incurs minimal expected cost in finding a solution.

Example

To illustrate, consider a problem with two variables X and Y, where X takes values 1, 2, 3 and 4, so $D(X) = \{1, 2, 3, 4\}$, and the domain, $D(Y)$, of Y is $\{5, 6\}$. There are two incomplete constraints, the first, c_1, is a unary constraint on X, and the second, c_2, is a binary constraint on the two variables. It is (currently) unknown if $X = 1$ satisfies constraint c_1. The probability p_1 that it does so is 0.9. We can determine (i.e., find out) if $X = 1$ satisfies constraint c_1, but this test incurs a cost of $K_1 = 50$. We write $c_1(X = 1) = \upsilon_1$, where υ_1 represents an unknown boolean value. It is also unknown if values $2, 3$ and 4 satisfy c_1. We have $c_1(X = 2) = \upsilon_2$, $c_1(X = 3) = \upsilon_3$ and $c_1(X = 4) = \upsilon_4$. The cost of determining unknowns υ_2, υ_3 and υ_4 is each 70, and the probability of success of each is 0.8. Tuples $(2, 6)$, $(3, 5)$ and $(4, 6)$ all satisfy the binary constraint, whereas tuples $(2, 5)$, $(3, 6)$ and $(4, 5)$ do not. It is unknown whether tuples $(1, 5)$ and $(1, 6)$ satisfy the constraint. We have $c_2(X = 1, Y = 5) = \upsilon_5$, and $c_2(X = 1, Y = 6) = \upsilon_6$, and $c_2(X = 2, Y = 6) = c_2(X = 3, Y = 5) = c_2(X = 4, Y = 6) = 1$. The other tuples have value 0. Unknowns υ_5 and υ_6 each have cost 200 and probability of success 0.1.

Consider a standard backtracking search algorithm with variable ordering X, Y and value ordering $1, 2, 3, 4$ for X and $5, 6$ for Y. The algorithm will first incur a cost of 50 in determining υ_1. This unknown will be determined successfully with 90% chance, and if so, then $X = 1$ satisfies c_1. After that, υ_5 will be determined, costing 200, but with only 0.1 chance of success. If both υ_1 and υ_5 are successfully determined then $(X = 1, Y = 5)$ is a solution of the CSP. However, this has only chance $0.9 \times 0.1 = 0.09$ of happening, and cost $50 + 200 = 250$ is incurred.

It can be shown that the expected cost incurred by this algorithm is approximately 464 and can be written as $E_1 + qE_2$, where $q = 0.1 + 0.9^3 = 0.829$, $E_1 = 50 + 0.9(200 + 0.9 \times 200) = 392$ and $E_2 = 70 + 0.2(70 + 0.2 \times 70) = 86.8$. ($E_1$ is the expected cost in the $X = 1$ branch, E_2 is the expected cost conditional on having reached the $X = 2$ constraint check, and q is the chance that the algorithm fails to find a solution with $X = 1$.) This is far from optimal, mainly because determining unknowns υ_5 and υ_6 is very expensive, and they also have only small chance of success. An optimal algorithm for this problem (i.e., one with minimal expected cost) can be shown to have expected cost $E_2 + (0.2^3 \times 389.5) \approx 90$, which can be achieved with a backtracking search algorithm which determines unknowns in the $X = 2, 3, 4$ branches before determining unknowns in the $X = 1$ branch. ■

Algorithms with low expected cost will clearly need to consider the costs and the probabilities. A backtracking algorithm should ideally not always determine any unknown it meets, but allow the possibility of delaying determining an unknown, to check whether it seems worthwhile doing so.

We define algorithms for this problem, based on backtracking search. Such algorithms can be crudely divided into three classes:

- Type 0: determining all unknowns to begin with;
- Type 1: determining unknowns as we meet them in the search;
- Type 2: making decisions about whether it's worth determining an unknown, making use of cost and probabilistic information.

The normal solving approaches for CSPs fall into Type 0, where the full CSP is elicited first and we then solve it, based on backtracking search with propagation at each node of the search tree. Algorithms for open constraints, which don't assume any cost or probability information, can be considered as being Type 1. In this paper we construct Type 2 algorithms, which make use of the cost and probabilistic information.

We consider a simple iterative algorithm based on a limit on the costs of unknowns that may be determined; for each cost limit value, we perform a backtracking search; if this fails to find a solution we increment the cost limit, and search again. With this algorithm it can easily happen that we pay a cost of determining an unknown tuple, only to find that that particular branch fails to lead to a solution for other reasons, as in the example, with unknown v_1. A natural idea is to delay determining an unknown, in order to find out if it is worth doing so. Our main algorithm, described in Section 4, usually will not immediately determine an unknown, but explore more deeply first. The experimental results in Section 5 strongly suggest that this can be worthwhile.

Related Work: The motivation for this work is related to part of that for Open Constraints [1,2,3,6], and Interactive CSPs [4,5], with a major difference being our assumption of there being cost and probabilistic information available ([2] considers costs in optimisation problems, but in a rather different way). Although these kinds of methods could be used for our problem, not taking costs and probabilities into account will, unsurprisingly, tend to generate solutions with poor expected cost, as illustrated by the example and our experimental results.

Another approach is to ignore the probabilistic information, and look for complete assignments that will incur minimal cost to check if they are solutions. Weighted constraints methods e.g., [7] can be used to search for such assignments. If all the probabilities were equal to 1 then this would solve the problem. However, it may well turn out that all the lowest cost assignments also have relatively low probability. Consider the example with K_5 (the cost of determining v_5) changed to be 10 instead of 200. The assignment which then needs minimum cost to discover if it's a solution is $(X_1 = 1, X_2 = 5)$; this again leads to a suboptimal algorithm. Alternatively, one could search for complete assignments which have highest probability of being a solution, as in Probabilistic CSPs [8].

Although this may perform satisfactorily if all the costs are equal, with varying costs it seems that the costs should be taken into account. Consider the example, but where p_5, the probability that $v_5 = 1$, is changed from 0.1 to 0.9. The assignment with greatest chance of being a solution is $(X_1 = 1, X_2 = 5)$; however the cost of finding this solution is 250, so trying this solution first is far from optimal.

The next section describes the model and problem more formally. Section 3 analyses the related problem of determining if a particular complete assignment is a solution. This analysis is important for our main algorithm, which is described in Section 4. Section 5 describes the experimental testing and results, and Section 6 discusses extensions.

2 A Formal Model for Interleaving Solving and Elicitation

Standard CSPs: Let V be a set of variables, which are interpreted as decision variables, so that we have the ability to choose values of them. Each variable $X \in V$ has an associated domain $D(X)$. For any subset W of V, let $D(W)$ be the set of assignments to W, which can be written as $\prod_{X \in W} D(X)$. Associated with each (standard) constraint c over V, is a subset V_c of V, which is called its *scope*. Define a (standard) constraint c over V to be a function from $D(V_c)$ to $\{0, 1\}$. We will sometimes refer to a set of constraints C over V as a *Constraint Satisfaction Problem (CSP) over V*. Let S be an assignment to all the variables V. S is said to satisfy constraint c if $c(S') = 1$, where S' is S restricted to V_c. S is a solution of CSP C (or, S *satisfies* C) if it satisfies each constraint in C.

The Unknowns: As well as decision variables V, we consider a disjoint set of variables $\mathcal{U}$, which we call the set of *unknowns*. These are uncertain variables, and we have no control over them. They are all boolean variables. We assume that, for any unknown $v \in \mathcal{U}$, we can determine (i.e., discover) the value of v, that is, whether $v = 1$ or $v = 0$. So we assume we have some procedure $Det(\cdot)$ that takes an unknown v as input and returns 1 or 0. We also assume that there is a certain cost $K_v \in [0, \infty)$ for executing this procedure on v, and that we have probabilistic information about the success of this procedure. In particular we assume that we know the probability p_v of success, i.e., the probability that $Det(v) = 1$.

Incomplete Constraints: An *incomplete constraint* c over $(V, \mathcal{U})$ has an associated subset V_c of V called its *scope*. c is a function from $D(V_c)$ to $\{0, 1\} \cup \mathcal{U}$. Hence, to any tuple $t \in D(V_c)$, c assigns 1, 0 or some unknown. c is intended as a partial representation of some standard constraint c^* over V_c. $c(t) = 1$ is interpreted as t satisfies the constraint c^*. Also, $c(t) = 0$ is interpreted as t doesn't satisfies the constraint c^*; otherwise, if $c(t) \in \mathcal{U}$, then it is unknown if t satisfies the constraint. We will sometimes refer to a set of incomplete constraints over $(V, \mathcal{U})$ as an *incomplete CSP*. An Expected Cost-based Interactive CSP (ECI CSP) is formally defined to be a tuple $\langle V, D, \mathcal{U}, K, p, C\rangle$, for set of variables V, set of unknowns $\mathcal{U}$, functions $K : \mathcal{U} \to [0, \infty)$, $p : \mathcal{U} \to [0, 1]$, and where C is a set of incomplete constraints over $(V, \mathcal{U})$.

Associated with an incomplete constraint c are two standard constraints with the same scope. The *known constraint* $\underline{c}$ is given by $\underline{c}(t) = 1$ if and only if $c(t) = 1$ (otherwise, $\underline{c}(t) = 0$). A tuple satisfies $\underline{c}$ if and only if it is known to satisfy c^*. The *potential constraint* $\overline{c}$ is given by $\overline{c}(t) = 0$ if and only if $c(t) = 0$ (otherwise, $\overline{c}(t) = 1$). A tuple satisfies $\overline{c}$ if it could potentially satisfy c^*. For a given set of incomplete constraints C, the *Known CSP* is the set of associated known constraints: $\underline{C} = \{\underline{c} : c \in C\}$, and the *Potential CSP* $\overline{C}$ is the set of associated potential constraints: $\{\overline{c} : c \in C\}$.

Suppose that $c(t) = v$, and we determine v and find out that $v = 1$. Then we now know that t does satisfy the constraint, so we can replace $c(t) = v$ by $c(t) = 1$. Define $c[v := 1]$ to be the incomplete constraint generated from c by replacing every occurrence of v by 1. We define $c[v := 0]$ analogously. More generally, let ω be an assignment to a set $\mathcal{W} \subseteq \mathcal{U}$ of unknowns, and let c be an incomplete constraint. $c[\omega]$ is the incomplete constraint obtained by replacing each v in $\mathcal{W}$ by its value $\omega(v)$. We define $C[\omega]$ to be $\{c[\omega] : c \in C\}$. $C[\omega]$ is thus the incomplete CSP updated by the extra knowledge ω we have about unknowns.

Incomplete CSP C *is solved by assignment* S *(to variables* V*) in the context* ω if S is a solution of the associated known CSP $\underline{C[\omega]}$. In other words, if S is known to be a solution of C given ω. An incomplete CSP C is *insoluble in the context* ω if the associated potential CSP $\overline{C[\omega]}$ has no solution. In this case, even if all the other unknowns are found to be equal to 1, the CSP is still insoluble.

Policies for Solving ECI CSPs

An algorithm for solving an incomplete CSP involves sequentially determining unknowns until we can find a solution. Of course, the choice of which unknown to determine next may well depend on whether previous unknowns have been determined successfully or not. In the example in Section 1, if we determine v_1 and discover that $v_1 = 0$ then there is no point in determining unknown v_5.

What we call a *policy* is a decision making procedure that sequentially chooses unknowns to determine. The choice of unknown to determine at any stage can depend on information received from determining unknowns previously. The sequence of decisions ends either with a solution to the known part of the CSP, or with a situation in which there is no solution, even if all the undecided unknown tuples are in their respective constraints. In more abstract terms a policy can be considered as follows:-

Given an assignment ω to some (possibly empty) set $\mathcal{W}$ of unknowns, a policy does one of the following:

(a) returns a solution of the Known CSP (given ω);
(b) returns “Insoluble” (it can only do this if the Potential CSP (given ω) is insoluble);
(c) choose another undetermined unknown.

In cases (a) and (b) the policy terminates. In case (c), the chosen unknown v is determined, with value $b = 1$ or 0. ω is then extended with $v = b$, and another choice is made by the policy, given $\omega \cup [v = b]$. The sequence continues until the problem is solved or proved unsatisfiable.

Define a *scenario* to be a complete assignment to all the unknowns. Let $\Pr(\alpha)$ be the probability of scenario α occurring. In the case of the variables $\mathcal{U}$ being independent, we have $\Pr(\alpha) = \prod_{v\,:\,\alpha(v)=1} p_v \times \prod_{v\,:\,\alpha(v)=0}(1 - p_v)$. In a scenario α, a policy iteratively chooses unknowns to determine until it terminates; let $\mathcal{W}_\alpha$ be the set of unknowns determined; the policy incurs a particular cost, say, K_α, which equals $\sum_{v \in \mathcal{W}_\alpha} K_v$. The expected cost of a policy is then equal to $\sum_\alpha \Pr(\alpha) K_\alpha$, where the summation is over all scenarios α.

Evaluating Policies. We evaluate policies in terms of their expected cost. So, we aim to define algorithms that implement policies which have relatively low expected cost.

Using Dynamic Programming to Generate an Optimal Policy

Although the problem involves minimising expected cost over all policies, the structure of the decisions—dynamically choosing a sequence from a (large) set of objects—does not fit very naturally into such formalisms as Influence Diagrams [9], Markov Decision Processes [10] and Stochastic Constraint Programming [11]. We describe below a simple dynamic programming [12] algorithm for generating an optimal policy.

Consider ECI CSP $\langle V, D, \mathcal{U}, K, p, C\rangle$. Let ω be an assignment to some set of unknowns $\mathcal{W} \subseteq \mathcal{U}$. Define $A(\omega)$ to be the minimal expected cost over all policies for solving $\langle V, D, \mathcal{U} - \mathcal{W}, K, p, C[\omega]\rangle$, the ECI CSP updated with ω. Then $A(\omega) = 0$ if either the associated Known CSP $\underline{C[\omega]}$ is soluble or the associated Potential CSP $\overline{C[\omega]}$ is insoluble. Otherwise, any policy chooses some unknown $v \in \mathcal{U} - \mathcal{W}$ to determine, incurring cost K_v and with chance p_v of finding that $v = 1$. If $v = 1$ then we have incomplete CSP $C(\omega \cup [v := 1])$ to solve, which has minimal expected cost $A(\omega \cup [v := 1])$. Therefore, $A(\omega)$ can be written as

$$\min_{v \in \mathcal{U} - \mathcal{W}} \big(K_v + p_v A(\omega \cup [v := 1]) + (1 - p_v) A(\omega \cup [v := 0])\big).$$

The minimal expected cost over all policies for solving the original ECI CSP is equal to $A[\top]$, where $\top$ is the assignment to the empty set of variables. We can thus find the minimal expected cost by using a simple dynamic programming algorithm, iteratively applying the above equation, starting with all scenarios (or from minimal assignments ω such that $\overline{C[\omega]}$ is insoluble); we can also find an optimal policy in this way, by recording, for each ω, a choice $v \in \mathcal{U} - \mathcal{W}$ which minimises the expression for $A(\omega)$. However, there are $3^{|\mathcal{U}|}$ different possible assignments ω, so this optimal algorithm will only be feasible for problems with very small $|\mathcal{U}|$, i.e., very few unknowns (whereas problem instances in our experiments in Section 5 involve more than 2,000 unknowns). More generally, it seems that we will need to use heuristic algorithms.

3 Evaluating a Complete Assignment

In this section we consider the problem of testing if a given complete assignment is a solution of an ECI CSP $\langle V, D, \mathcal{U}, K, p, C\rangle$; the key issue is the order in which we determine the associated unknowns. This analysis is relevant for our main algorithm described in Section 4.

Associated with each potential solution S (i.e., solution of the associated potential CSP $\overline{C}$) is a set of unknowns, which can be written as: $\{c(S) : c \in C\} \cap \mathcal{U}$. An unknown υ is in this set if and only if there exists some constraint c such that $c(S) = \upsilon$. Label these unknowns as $U = \{\upsilon_1, \ldots, \upsilon_m\}$; we abbreviate p_{υ_i} to p_i, and K_{υ_i} to K_i. We also define $r_i = K_i/(1 - p_i)$, where we set $r_i = \infty$ if $p_i = 1$. Assignment S is a solution of the unknown CSP if and only if each of the unknown values in U is actually a 1. In this section we assume that the unknowns are independent variables, so that the probability that S is a solution of the CSP is $p_1 p_2 \cdots p_m$, which we write as $P(U)$.

To evaluate set of unknowns $\{\upsilon_1, \ldots, \upsilon_m\}$, we determine them in some order until either we find one which fails, i.e., until $Det(\upsilon_i) = 0$, or until we have determined them all. Associated with an unknown υ_i is the cost K_i and success probability p_i. Suppose we evaluate the unknowns in the sequence $\upsilon_1, \ldots, \upsilon_m$. We start by determining υ_1, incurring cost K_1. If υ_1 is successfully determined (this event has chance p_1), we go on to determine υ_2, incurring additional cost K_2, and so on. The expected cost in evaluating these unknowns in this order is therefore $K_1 + p_1 K_2 + p_1 p_2 K_3 + \cdots + p_1 p_2 \cdots p_{m-1} K_m$. Let R_π be the expected cost incurred in evaluating unknowns $\{\upsilon_1, \ldots, \upsilon_m\}$ in the order $\pi(1), \pi(2), \ldots, \pi(m)$, i.e., with $\upsilon_{\pi(1)}$ first, and then $\upsilon_{\pi(2)}$, etc. Expected cost R_π is therefore equal to $K_{\pi(1)} + p_{\pi(1)} K_{\pi(2)} + p_{\pi(1)} p_{\pi(2)} K_{\pi(3)} + \cdots + p_{\pi(1)} p_{\pi(2)} \cdots p_{\pi(m-1)} K_{\pi(m)}$.

Proposition 1. *For a given set of unknowns $\{\upsilon_1, \ldots, \upsilon_m\}$, R_π is minimised by choosing π to order unknowns with smallest r_i $(= K_i/(1 - p_i))$ first, i.e., setting ordering $\pi(1), \pi(2), \ldots, \pi(m)$ in any way such that $r_{\pi(1)} \le r_{\pi(2)} \le \cdots \le r_{\pi(m)}$.*

For a set of unknowns $U = \{\upsilon_1, \ldots, \upsilon_m\}$ we define $R(U)$ to be R_π, where π is chosen so as the minimise the cost, by ordering the unknowns to have smallest r_i first (as shown by Proposition 1).

Imagine a situation where we are given an ECI CSP, and a complete assignment S which is a possible solution. We will consider an expected-utility-based analysis of whether it is worth determining these unknowns, to test if S is a solution, where cost is negative utility. Suppose the utility of finding that S is a solution is Q. The chance of finding that S is a solution is $P(U)$, so the expected reward is $P(U) \times Q$. If we determine all the unknowns U (based on the minimal cost order) then the expected cost is $R(U)$. Therefore the overall expected gain is $(P(U) \times Q) - R(U)$, so there is a positive expected gain if and only if $P(U) \times Q > R(U)$, i.e., if and only if $Q > R(U)/P(U)$.

A natural approach, therefore, for solving an Expected Cost-based Interactive CSP is to search for solutions whose associated set of unknowns U has relatively low value of $R(U)/P(U)$. In particular, we can perform iterative searches based on an upper bound on $R(U)/P(U)$, where this upper bound is increased with each search. This is the basis of our main algorithm, described in the next section.

The monotonicity property, shown by the following proposition, is important since it allows the possibility of subtrees being pruned: if a partial assignment S has associated set of unknowns U, and we find that $R(U)/P(U)$ is more than our cost bound Q, then we can backtrack, since the set of unknowns U' associated with any complete assignment extending S will also have $R(U')/P(U') > Q$. Since $\mathcal{U}$ is finite, it is sufficient to show the result for the case when U' contains a single extra unknown (as we can then repeatedly add extra unknowns one-by-one to prove the proposition). The result for this case follows quite easily by expanding $R(U)$.

Proposition 2. *Let U and U' be any sets of unknowns with $U \subseteq U' \subseteq \mathcal{U}$. Then $R(U)/P(U) \leq R(U')/P(U')$.*

4 Iterative Expected Cost-Bound Algorithm

In this section we define our main algorithm for solving a given Expected Cost-based CSP $\langle V, D, \mathcal{U}, K, p, C \rangle$. The key idea behind this algorithm is to allow the possibility of delaying determining an unknown associated with a constraint check, until it has explored further down the search tree; this is in order to see if it is worth paying the cost of determining that unknown. The algorithm performs a series of depth-first searches; each search is generated by the procedure *TreeSearch*. The structure of each search is very similar to that of a standard backtracking CSP algorithm.

The behaviour in each search (i.e., in each call of *TreeSearch*) depends on the value of a global variable Q, which is involved in a backtracking condition, and is increased with each tree search. For example, in the experiments described in Section 5, we set $Q_{initial} = 20$ and define $Next(Q)$ to be $Q \times 1.5$, so that the first search has Q set to 20, the second search has $Q = 30$, and then $Q = 45$, and so on. The value of Q can be roughly interpreted as the cost that the algorithm is currently prepared to incur to solve the problem.[1]

The procedure *TopLevel* first initialises the cost incurred (`GlobalCost`) to zero. It then performs repeated tree searches until a solution is found (see procedure *ProcessNode*$(\cdot)$ below) or until all unknowns have been determined; in the latter case, it then performs one further tree search (which is then an ordinary CSP backtracking algorithm).

[1] Our experimental results for the main algorithm (without the size limit modification) tally very well with this interpretation, with the average Q for the last iteration being close to the average overall cost incurred (within 25% of the average cost for each of the four distributions used).

```
Procedure TopLevel
    GlobalCost := 0;    $Q := Q_{initial}$
    repeat
        TreeSearch
        $Q := Next(Q)$
        until all unknowns have been determined
    TreeSearch
```

```
Procedure TreeSearch
    $\texttt{Unknowns}_{root} := \emptyset$;
    Construct child $N$ of root node
    ProcessNode($N$)
```

The core part of the algorithm is the procedure *ProcessNode*(N). Let N be the current node, and let $Pa(N)$ be its parent, i.e., the node above it in the search tree. Associated with node N is the set $\texttt{Unknowns}_N$ of *current unknowns*. At a node, if any of the current unknowns evaluates to 0 then there is no solution below this node. Conversely, if all of the current unknowns at a node evaluate to 1 then the current partial assignment is consistent with all constraints that have been checked so far. If all the current unknowns at a leaf node evaluate to 1 then the current assignment is a solution.

At a search node, we perform, as usual, a constraint check for each constraint c whose scope V_c has just been fully instantiated (i.e., such that (i) the last variable instantiated is in the scope, and (ii) the set of variables instantiated contains the scope). A constraint check returns either 1, 0 or some unknown. The algorithm first determines if any constraint check fails, i.e., if it returns 0. If so, we backtrack to the parent node, in the usual way, assigning an untried value of the associated variable, when possible, and otherwise backtracking to *its* parent node. Propagation can be used in the usual way to eliminate elements of a domain which cannot be part of any solution extending the current assignment.

The set $\texttt{DirectUnknowns}_N$, of unknowns directly associated with the node N, is defined to be the set of unknowns which are generated by the constraint checks at the node. The set of *current unknowns* at the node, $\texttt{Unknowns}_N$, is then initialised to be the union of $\texttt{DirectUnknowns}_N$ and the current unknowns of the parent node.

The algorithm then tests to see if it is worth continuing, or if it is expected to be too expensive to be worth determining the current set of unknowns. The backtracking condition is based on the analysis in Section 3. We view Q as representing (our current estimate of) the value of finding a solution. Then the expected gain, if we determine all the unknowns in $\texttt{Unknowns}_N$, is $P(\texttt{Unknowns}_N) \times Q$ where $P(\texttt{Unknowns}_N)$ is the chance that all the current unknowns evaluate to 1. The expected cost of determining these unknowns sequentially is $R(\texttt{Unknowns}_N)$, as defined in Section 3, since we evaluate unknowns with smallest r_i first. So, determining unknowns $\texttt{Unknowns}_N$ is not worthwhile if the expected gain is less than the expected cost: $P(\texttt{Unknowns}_N) \times Q < R(\texttt{Unknowns}_N)$. Therefore we backtrack if $R(\texttt{Unknowns}_N)/P(\texttt{Unknowns}_N) > Q$.

We then construct a child node in the usual way, by choosing the next variable Y to instantiate, choosing a value y of the variable, and extending the current assignment with $Y = y$. If Y is the last variable to be instantiated then we use the *ProcessLeafNode*$(\cdot)$ procedure on the new node; otherwise we use the *ProcessNode*$(\cdot)$ procedure on the new node.

The *ProcessLeafNode*$(\cdot)$ procedure is similar to *ProcessNode*$(\cdot)$, except that we can no longer delay determining unknowns, so we determine each current unknown until we fail, or until all have been determined successfully. We determine an unknown with smallest $r_i = K_i/(1-p_i)$ first (based on Proposition 1). If an unknown v_i is determined unsuccessfully, then there is no solution beneath this node. In fact, if N' is the furthest ancestor node of N which v_i is directly associated with (i.e. such that $\texttt{DirectUnknowns}_{N'} \ni v_i$), then there is no solution beneath N'. Therefore, we jump back to N' and backtrack to its parent node $Pa(N')$. If all the unknowns $\texttt{Unknowns}_N$ have been successfully determined then the current assignment, which assigns a value to all the variables V, has been shown to be a solution of the CSP, so the algorithm has succeeded, and we terminate the algorithm.

```
Procedure ProcessNode(N)
  if any constraint check returns 0 then backtrack
  $\texttt{Unknowns}_N := \texttt{Unknowns}_{Pa(N)} \cup \texttt{DirectUnknowns}_N$
  if $R(\texttt{Unknowns}_N)/P(\texttt{Unknowns}_N) > Q$
      then backtrack to parent node
  Construct child node $N'$ of $N$
  if $N'$ is a leaf node (all variables are instantiated)
      then ProcessLeafNode($N'$) else ProcessNode($N'$)
```

```
Procedure ProcessLeafNode(N)
  if any constraint check returns 0 then backtrack
  $\texttt{Unknowns}_N := \texttt{Unknowns}_{Pa(N)} \cup \texttt{DirectUnknowns}_N$
  if $R(\texttt{Unknowns}_N)/P(\texttt{Unknowns}_N) > Q$ then backtrack to parent node
  while Unknowns non-empty do:
      Let $v_i$ be unknown in $\texttt{Unknowns}_N$ with minimal $r_i$
      Determine $v_i$;
      $\texttt{GlobalCost} := \texttt{GlobalCost} + K_i$
      if $v_i$ determined unsuccessfully
          then Jump Back to furthest ancestor node associated with $v_i$
              and backtrack to its parent node
      $\texttt{Unknowns}_N := \texttt{Unknowns}_N - \{v_i\}$
      end (while )
  Return current (complete) assignment as a solution and STOP
```

If we apply this algorithm to the example in Section 1 (again using variable ordering X, Y, and numerical value ordering), no unknown will be determined until we reach an iteration where Q is set to being at least 87.5. Then the first leaf node N that the algorithm will reach is that associated with the assignment $(X =$

$1,\ Y = 5)$. $\texttt{Unknowns}_N$ is equal to $\{v_1, v_5\}$. $r_1 = K_1/(1-p_1) = 50/0.1 = 500 > r_5 = 200/0.9$, so $R(\{v_1, v_5\}) = K_5 + p_5 K_1 = 205$, and $R(\{v_1, v_5\})/P(\{v_1, v_5\}) = 205/(0.9 \times 0.1) \approx 2278$, so the algorithm will backtrack; similarly for the leaf node associated with assignment $(X = 1,\ Y = 6)$. The leaf node corresponding to $(X = 2,\ Y = 6)$ has current set of unknowns $\{v_2\}$. Since $R(\{v_2\})/P(\{v_2\}) = 70/0.8 = 87.5 \geq Q$, unknown v_2 will be determined. If it evaluates to 1 then $(X = 2,\ Y = 6)$ is a solution, and the algorithm terminates. Otherwise, v_3 and v_4 will be next to be determined. In fact, for this example, the algorithm generates an optimal policy, with expected cost of around 90.

One approach to improving the search efficiency of the algorithm is to set a limit $\texttt{SizeLimit}$ on the size of the current unknown set $\texttt{Unknowns}_N$ associated with a node. If $|\texttt{Unknowns}_N|$ becomes larger than $\texttt{SizeLimit}$ then we repeatedly determine unknowns, in increasing order of r_i, and remove the unknown from the current set until $|\texttt{Unknowns}_N| = \texttt{SizeLimit}$. It is natural then to change the backtracking condition to take this into account. In particular, we can change the test to be $(\texttt{CostDetSucc}_N + R(\texttt{Unknowns}_N))/P(\texttt{Unknowns}_N) > Q$, where $\texttt{CostDetSucc}_N$ is the cost incurred in (successfully) determining unknowns in ancestors of the current node, which can be considered as the cost that has already been spent in consistency checking of the current assignment.

In the algorithm whenever we determine an unknown in $\texttt{Unknowns}_N$ we choose an unknown v_i with minimum r_i. This is in order to minimise the expected cost of determining the set of current unknowns, because of Proposition 1. Alternatively, one could bias the ordering towards determining more informative unknowns. For example, suppose $\texttt{Unknowns}_N$ includes two unknowns v_i and v_j, where v_i is associated with a unary constraint, and v_j is associated with a constraint of larger arity. Even if r_i is slightly more than r_j, it may sometimes be better to determine v_i before v_j since v_i may well be directly associated with many other nodes in the search tree.

5 Experimental Testing

The problem instances used in the experiments were generated as follows: A soluble random binary extensional CSP is generated with parameters $\langle n, d, m, t \rangle$, where n is the number of variables, d the uniform domain size, m the graph density and t the constraint tightness, with the parameters chosen so that each instance is likely to be fairly easily soluble. For each constraint in this CSP we randomly select a number of the allowed tuples and randomly select the same number of the disallowed tuples to be assigned unknown. Each such tuple is assigned a different unknown v_i, which is assigned a probability p_i chosen independently from a uniform distribution taking values between 0 and 1. Each v_i is allocated its true value (which the algorithms only have access to when they determine u_i): this is assigned 1 with probability p_i, otherwise it is assigned 0 (where $v_i = 1$ means that the associated tuple satisfies the constraint).

We use four different distributions for cost, where costs are integers in our experiments. Each distribution has minimum value 1 and has median around 50. For $k = 1, 2, 3, 4$ using the k^{th} distribution, each cost K_v is an independent sample of the random variable: $50 \times (2 \times \textbf{rand})^k$ rounded up to be an integer, where **rand** is a random number taking values between 0 and 1 with a uniform distribution. Therefore $k = 1$ has a linear distribution, $k = 2$ is a scaled (and truncated) square root distribution, and so on.

The parameters for $\langle n, d, m, t\rangle$ were $\langle 20, 10, 0.163, 0.4\rangle$ and $\langle 20, 10, 0.474, 0.3\rangle$. Each problem set contained 100 problems. No problem was soluble without determining at least one unknown. For $\langle 20, 10, 0.163, 0.4\rangle$, the first problem parameters, we generated four problem sets by using four different cost distributions; for the other problem set we used the linear ($k = 1$) cost distribution.

The following five algorithms were tested (where, according to the terminology in Section 1, the first is Type 1, and the others are Type 2):

Basic Algorithm: The basic algorithm works like a normal CSP depth-first search algorithm (maintaining arc consistency, and with min domain variable ordering) except that it determines each unknown as soon as it is encountered. As usual, a constraint check is performed as soon as all the variables in the constraint's scope are instantiated. When a constraint check returns an unknown v_i, we immediately determine v_i, incurring cost K_i. If v_i is determined successfully, i.e., v_i is found to be 1, then the constraint check is successful.

Basic Iterative (Cost-Bound Algorithm): This algorithm performs iterative searches, parameterised by increasing cost bound q; each search is similar to the basic algorithm, except that all unknowns with cost greater than q are removed from search for the current iteration (that is, they are set to 0, which allows for improved pruning through propagation). If a solution is found, search terminates; otherwise, the search is complete, after which search restarts with q incremented by a constant, q_{inc}. The process continues until either a solution has been found or all unknowns have been determined and the algorithm has proven insolubility. For the experiments reported below, q starts off with value 0, and q_{inc} is 5.

Main Iterative (Expected Cost-Bound Algorithm): This is the main algorithm described in Section 4. For this and the next two algorithms, $Q_{initial}$ is set to 20, with a multiplicative increment of 1.5.

Main Algorithm with Size Limit: This is the adapted version of the last algorithm discussed at the end of Section 4, which incorporates a limit `SizeLimit` on the cardinality of the set `Unknowns` of current unknowns, so that if $|$`Unknowns`$| >$ `SizeLimit` then elements of `Unknowns` are determined until either one is found to be 0 or $|$`Unknowns`$| =$ `SizeLimit`. In the experiments reported below, we set `SizeLimit` to 5.

Mixed Algorithm: This algorithm modifies the main Expected Cost-Bound algorithm by having, for each iteration, a cost bound q on each unknown being considered, in order to improve the search efficiency (because of additional propagation), whilst maintaining cost effectiveness. It therefore is a kind of hybrid of the main algorithm and the basic iterative cost-bound algorithm described above. Let `maxK` be the maximum cost of any unknown in the current problem

instance. On the first search all unknowns with cost greater than 30%(`maxK`) are removed from search, so that q starts at 30%(`maxK`); on each iteration this bound q is incremented by $q_{inc} = 5\%$(`maxK`).
We also implemented a cost-based value ordering heuristic. The heuristic chooses the value which has minimum total cost over the constraints between the variable and its instantiated neighbors. The improvements for the iterative algorithms were minimal so we only present the results for the basic algorithm with the value ordering ("Basic Value").

Table 1. Results For Different Cost Distributions. Costs are Averaged Over 100 Problems. Bottom row gives mean search nodes for linear problem set.

	Basic	Basic Value	Basic Iterative	Main Iterative	Size Limit	Mixed
Linear ($k = 1$)	3272	2625	1711	152	495	178
Square ($k = 2$)	4574	3414	900	105	346	113
Cube ($k = 3$)	6823	4997	566	79	231	77
Fourth ($k = 4$)	11123	8401	344	50	180	52
Linear Search Nodes	57	55	652	2.1×10^6	1.2×10^6	1.2×10^5

Notes: Problem parameters $\langle 20, 10, 0.163, 0.4\rangle$.

Discussion: Table 1 gives the results for the average costs on four different problem sets. These all had the same parameters $\langle n, d, m, t\rangle$ but costs were generated for unknowns using the distributions described above. The "main iterative" algorithm performs best in terms of average cost: the basic iterative algorithm incurs between around 7 to 11 times more cost for these instances, and the basic algorithm has average cost one or two orders of magnitude worse than the main iterative algorithm. (Naturally, all the algorithms do much better than determining all the unknowns prior to search, at a cost of more than 100,000.)

Unsurprisingly, the main iterative algorithm is vastly slower than the basic algorithms, generating on average around two million total search tree nodes for each problem instance, whereas the basic iterative generates only a few hundred. The last two algorithms both aim to improve the efficiency somewhat. The "Size Limit" algorithm cuts search tree nodes by more than 50% compared to the main algorithm, but incurs roughly three or more times as much average cost. The mixed algorithm trades off cost and search efficiency much more effectively for these instances, with only slightly worse average costs, but generating only a fraction of the search tree nodes, less than 6% for the linear distribution, and less than 30% for the other distributions.

The other problem set showed similar behaviour, though even more extreme: in the $\langle 20, 10, 0.0823, 0.75\rangle$ problem set with the linear distribution, the mixed algorithm average cost was 222, compared to 5022 for the basic iterative. The size limit algorithm average cost was 1320, and generated more than nine times as many nodes as the mixed algorithm.

6 Extensions and Summary

Extending the algorithms: Our main algorithm can be considered as searching for complete assignments with small values of $R(U)/P(U)$, where U is the set of unknowns associated with the assignment, $P(U)$ is the probability that all of U are successfully determined (and hence that the assignment is a solution), and $R(U)$ is the expected cost of checking this. There are other ways of searching for assignments with small values of $R(U)/P(U)$, in particular, one could use local search algorithms or branch-and-bound algorithms. Such algorithms can be used to generate promising assignments, which we can sequentially test to see if they are solutions or not. If not, then we move on to the next potential solution (possibly updating the problem to take into account determined unknowns).

The efficiency of our main algorithm and a branch-and-bound algorithm would probably be greatly increased if one could design an efficient propagation mechanism of an upper bound constraint on $R(U)/P(U)$. Failing that, one might use a propagation method for weighted constraints [7] to prune subtrees of assignments with total associated cost above a threshold, or with probabilities below a threshold (the latter using a separate propagation, making use of the log/exponential transformation between weighted constraints and probabilistic constraints).

Extensions of the model: Our model of interleaving solving and elicitation is a fairly simple one. There are a number of natural ways of extending it to cover a wider range of situations. In particular, the framework and algorithms can be easily adapted to situations where there is a cost incurred for determining a *set* of unknowns (rather than a single unknown); for example, there may be a single cost incurred for determining all the unknowns in a particular constraint. The paper has focused especially on the case of the probabilities being independent; however, the model and algorithms can be applied in non-independent cases as well. Our model and algorithms also apply to the case where determining an unknown may leave it still unknown; unsuccessfully determining an unknown then needs to be reinterpreted as meaning that we are unable to find out if the associated tuple satisfies the constraint or not. Our current model allows a single unknown to be assigned to several tuples; although this can allow some representation of intensional constraints, we may also wish to allow non-boolean unknowns, for example, for a constraint $X - Y \geq \lambda$ where λ is an unknown constant.

In many situations, it could be hard to reliably estimate the success probabilities and costs, in particular, if a cost represents the time needed to find the associated information. However, since the experimental results indicate that taking costs and probabilities into account can make a very big difference to the expected cost, it could be very worthwhile making use of even very crude estimates of costs and probabilities.

Summary

The paper defines a particular model for when solving and elicitation are interleaved, which takes costs and success probabilities into account. A formal representation of such a problem is defined. A dynamic programming algorithm can be used to solve the problem optimally, i.e., with minimum expected cost; however this is only computationally feasible for situations in which there are few unknowns, i.e., very little unknown information. We define and experimentally test a number of algorithms based on backtracking search, with the most successful (though computationally expensive) ones being based on delaying determining an unknown until more information has been received.

References

1. Faltings, B., Macho-Gonzalez, S.: Open constraint satisfaction. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 356–370. Springer, Heidelberg (2002)
2. Faltings, B., Macho-Gonzalez, S.: Open constraint optimization. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 303–317. Springer, Heidelberg (2003)
3. Faltings, B., Macho-Gonzalez, S.: Open constraint programming. Artificial Intelligence 161(1–2), 181–208 (2005)
4. Cucchiara, R., Lamma, E., Mello, P., Milano, M.: An interactive constraint-based system for selective attention in visual search. In: International Syposium on Methodologies for Intelligent Systems, pp. 431–440 (1997)
5. Lamma, E., Mello, P., Milano, M., Cucchiara, R., Gavanelli, M., Piccardi, M.: Constraint propagation and value acquisition: why we should do it interactively. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), pp. 468–477 (1999)
6. Lallouet, A., Legtchenko, A.: Consistencies for partially defined constraints. In: Proc. International Conference on Tools with Artificial Intelligence (ICTAI'05) (2005)
7. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc consistency. Artificial Intelligence 159(1–2), 1–26 (2004)
8. Fargier, H., Lang, J.: Uncertainty in Constraint Satisfaction Problems: a probabilistic approach. In: Moral, S., Kruse, R., Clarke, E. (eds.) ECSQARU 1993. LNCS, vol. 747, pp. 97–104. Springer, Heidelberg (1993)
9. Howard, R., Matheson, J.: Influence diagrams. In: Readings on the Principles and Applications of Decision Analysis, pp. 721–762 (1984)
10. Puterman, M.: Markov Decision Processes, Discrete Stochastic Dynamic Programming. John Wiley & Sons, Chichester (1994)
11. Tarim, S.A., Manadhar, A., Walsh, T.: Stochastic constraint programming: A scenario-based approach. Constraints 11(1), 53–80 (2006)
12. Bellman, R.: Dynamic Programming. Princeton University Press (1957)