QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems

Ulrich Junker
ILOG
1681, route des Dolines
06560 Valbonne
France
ujunker@ilog.fr

Abstract

Over-constrained problems can have an exponential number of conflicts, which explain the failure, and an exponential number of relaxations, which restore the consistency. A user of an interactive application, however, desires explanations and relaxations containing the most important constraints. To address this need, we define preferred explanations and relaxations based on user preferences between constraints and we compute them by a generic method which works for arbitrary CP, SAT, or DL solvers. We significantly accelerate the basic method by a divide-and-conquer strategy and thus provide the technological basis for the explanation facility of a principal industrial constraint programming tool, which is, for example, used in numerous configuration applications.

Introduction

Even experienced modelling experts may face over-constrained situations when formalizing the constraints of a combinatorial problem. In order to identify and to correct modelling errors, the expert needs to identify a subset of the constraints that explain the failure, while focusing on the most important ones. Alternatively, the expert can be interested in a subset of the constraints that have a solution, again preferring the important constraints.

In interactive applications, the careful selection of explanations and relaxations is an even more important problem. We consider a simple sales configuration problem, where not all user requirements can be satisfied:

Example 1 *A customer wants to buy a station-wagon with following options, but has a limited budget of* 3000*:*

	Option	*Requirement ρ_i*	*Costs*
1.	*roof racks*	$x_1 = 1$	$k_1 = 500$
2.	*CD-player*	$x_2 = 1$	$k_2 = 500$
3.	*one additional seat*	$x_3 = 1$	$k_3 = 800$
4.	*metal color*	$x_4 = 1$	$k_4 = 500$
5.	*special luxury version*	$x_5 = 1$	$k_5 = 2600$

where the boolean variable $x_i \in \{0,1\}$ indicates whether the i-th option is chosen and the costs $y = \sum_{i=1}^{5} k_i \cdot x_i$ are smaller than the total budget of 3000*.*

A constraint solver maintaining bound consistency will successively increase the lower bound for y if the requirements are propagated one after the other (see Table 1). When propagating the last requirement, the lower bound of 4900 exceeds 3000 and a failure is obtained. A straightforward explanation for this failure is obtained if we maintain the set of requirements explaining why $y \geq lb_i$. Unfortunately, the resulting explanations contains all requirements meaning that requirements may be removed without need.

Requirement	Deduction	Argument/Conflict
$\rho_1 : x_1 = 1$	$y \geq 500$	$\{\rho_1\}$
$\rho_2 : x_2 = 1$	$y \geq 1000$	$\{\rho_1, \rho_2\}$
$\rho_3 : x_3 = 1$	$y \geq 1800$	$\{\rho_1, \rho_2, \rho_3\}$
$\rho_4 : x_4 = 1$	$y \geq 2300$	$\{\rho_1, \rho_2, \rho_3, \rho_4\}$
$\rho_5 : x_5 = 1$	$y \geq 4900$	$\{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$
	$fail$	$\{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$

Table 1: Computing a conflict during propagation.

Requirement	Deduction	Argument/Conflict
$\rho_4 : x_4 = 1$	$y \geq 500$	$\{\rho_4\}$
$\rho_5 : x_5 = 1$	$y \geq 3100$	$\{\rho_4, \rho_5\}$
	$fail$	$\{\rho_4, \rho_5\}$

Table 2: Propagation for producing a minimal conflict.

We are therefore interested in minimal (i.e. irreducible) conflicts. Table 2 shows another sequence of propagations, which results into the minimal conflict $\{\rho_4, \rho_5\}$. If the customer prefers a special luxury version to metal color, ρ_4 will be removed, meaning that we can get another conflict, e.g. $\{\rho_3, \rho_5\}$. However, the customer prefers an additional seat to the special luxury version and now removes ρ_5, meaning that only $\{\rho_1, \rho_2, \rho_3\}$ are kept. This relaxation of the requirements is not maximal, since ρ_4 can be re-added after the removal of ρ_5. Unnecessary removals can be avoided if we directly produced the conflict $\{\rho_3, \rho_5\}$ containing the preferred requirements. If we take into account user preferences between requirements, we can directly determine preferred explanations as the one shown in Table 3 (we write $\rho_i \prec \rho_j$ iff ρ_i is preferred to ρ_j).

Hence, the essential issue in explaining a failure of a constraint solver is not the capability of recording a proof, but selecting a proof among a potentially huge number that does not contain unnecessary constraints and that involves

Requirement	Deduction	Argument/Conflict
$\rho_3 : x_3 = 1$	$y \geq 800$	$\{\rho_3\}$
$\rho_5 : x_5 = 1$	$y \geq 3400$	$\{\rho_3, \rho_5\}$
	$fail$	$\{\rho_3, \rho_5\}$

Table 3: A preferred explanation for $\rho_3 \prec \rho_1 \prec \rho_2 \prec \rho_5 \prec \rho_4$.

the most preferred constraints. We address this issue by a preference-controlled algorithm that successively adds most preferred constraints until they fail. It then backtracks and removes least preferred constraints if this preserves the failure. Relaxations can be computed dually, first removing least preferred constraints from an inconsistent set until it is consistent. The number of consistency checks can drastically be reduced by a divide-and-conquer strategy that successively decomposes the overall problem. In the good case, a single consistency check can remove all the constraints of a subproblem.

We first define preferred relaxations and explanations and then develop the preference-based explanation algorithms. After that, we discuss consistency checking involving search as well as related work.

Preferred Explanations and Relaxations

Although the discussion of this paper focuses on constraint satisfaction problems (CSP), its results and algorithms apply to any satisfiability problem such as propositional satisfiability (SAT) or the satisfiability of concepts in description logic (DL). We completely abstract from the underlying constraint language and simply assume that there is a monotonic satisfiability property: *if S is a solution of a set $\mathcal{C}_1$ of constraints then it is also a solution of all subsets $\mathcal{C}_2$ of $\mathcal{C}_1$.*

If a set of constraints has no solution, some constraints must be relaxed to restore consistency. It is convenient to distinguish a background $\mathcal{B}$ containing the constraints that cannot be relaxed. Typically, unary constraints $x \in D$ between a variable x and a domain D will belong to the background. In interactive problems, only user requirements can be relaxed, leaving all other constraints in the background. We now define a relaxation of a problem $\mathcal{P} := (\mathcal{B}, \mathcal{C})$:

Definition 1 *A subset R of $\mathcal{C}$ is a* relaxation *of a problem $\mathcal{P} := (\mathcal{B}, \mathcal{C})$ iff $\mathcal{B} \cup R$ has a solution.*

A relaxation exists iff $\mathcal{B}$ is consistent. Over-constrained problems can have an exponential number of relaxations. A user typically prefers to keep the important constraints and to relax less important ones. That means that the user is at least able to compare the importance of some constraints. Thus, we will assume the existence of a strict partial order between the constraints of $\mathcal{C}$, denoted by $\prec$. We write $c_1 \prec c_2$ iff (the selection of) constraint c_1 is preferred to (the selection of) c_2. (Junker & Mailharro 2003) show how those preferences can be specified in a structured and compact way. There are different ways to define preferred relaxations on such a partial order (cf. e.g. (Junker 2002)). In this paper, we will pursue the lexicographical approach of (Brewka 1989) which assumes the existence of a unique ranking among the constraints. The partial order $\prec$ is considered an incomplete specification of this ranking. We will introduce three extensions of this partial order:

- A *linearization* $<$ of $\prec$, which is a strict total order that is a superset of $\prec$ and which describes the ranking.
- Two lexicographic extensions of $<$, denoted by $<_{lex}$ and $<_{antilex}$, which are defined over sets of constraints.

Those lexicographic orders will be defined below. For now, we keep only in mind that two relaxations can be compared by the lexicographic extension $<_{lex}$.

Similarly, an over-constrained problem may have an exponential number of conflicts that explain the inconsistency.

Definition 2 *A subset C of $\mathcal{C}$ is a* conflict *of a problem $\mathcal{P} := (\mathcal{B}, \mathcal{C})$ iff $\mathcal{B} \cup C$ has no solution.*

A conflict exists iff $\mathcal{B} \cup \mathcal{C}$ is inconsistent. Some conflicts are more relevant for the user than other conflicts. Suppose that there are two conflicts in a given constraint system:

- Conflict 1 involves only very important constraints.
- Conflict 2 involves less important constraints.

The intuition is that conflict 1 is much more significant for the user than conflict 2. Indeed, in any way, the user will have to resolve the first conflict, and thus, he will have to relax at least one important constraint. As for the second conflict, a less important constraint can be relaxed and the user will consider such a modification as more easy to do.

We now give a formalization of the above intuitions. We define *preferred relaxations* following (Brewka 1989) and then give an analogous definition for *preferred conflicts*. Firstly, we recall the definition of the lexicographic extension of a total order.

Definition 3 *Given a total order $<$ on $\mathcal{C}$, we enumerate the elements of $\mathcal{C}$ in increasing $<$-order $c_1, \ldots, c_n$ starting with the most important constraints (i.e. $c_i < c_j$ implies $i < j$) and compare two subsets X, Y of $\mathcal{C}$ lexicographically:*

$$\begin{array}{c} X <_{lex} Y \\ \textit{iff} \\ \exists k : c_k \in X - Y \textit{ and} \\ X \cap \{c_1, \ldots, c_{k-1}\} = Y \cap \{c_1, \ldots, c_{k-1}\} \end{array} \tag{1}$$

Next, we define preferred relaxations, first for a total order over the constraints, and then for a partial order:

Definition 4 *Let $\mathcal{P} := (\mathcal{B}, \mathcal{C}, <)$ be a totally ordered problem. A relaxation R of $\mathcal{P}$ is a* preferred relaxation *of $\mathcal{P}$ iff there is no other relaxation R^* of $\mathcal{P}$ s.t. $R^* <_{lex} R$.*

Definition 5 *Let $\mathcal{P} := (\mathcal{B}, \mathcal{C}, \prec)$ be a partially ordered problem. A relaxation R of $\mathcal{P}$ is a* preferred relaxation *of $\mathcal{P}$ iff there is a linearization $<$ of $\prec$ s.t. R is a preferred relaxation of $(\mathcal{B}, \mathcal{C}, <)$.*

A preferred relaxation R is maximal (non-extensible) meaning that each proper superset of R has no solution. If no preferences are given, i.e. $\prec$ is the empty relation, then the maximal relaxations and the preferred relaxations coincide. If $\prec$ is a strict total order and $\mathcal{B}$ is consistent, then $\mathcal{P}$ has a unique preferred relaxation.

The definitions of preferred conflicts follow the same scheme as the definitions of the preferred relaxations. In order to get a conflict among the most important constraints, we prefer the retraction of least important constraints:

Definition 6 *Given a total order $<$ on $\mathcal{C}$, we enumerate the elements of $\mathcal{C}$ in increasing order $c_1, \ldots, c_n$ (i.e. $c_i < c_j$ implies $i < j$) and compare X and Y lexicographically in the reverse order:*

$$\begin{array}{c} X <_{antilex} Y \\ \textit{iff} \\ \exists k : c_k \in Y - X \textit{ and} \\ X \cap \{c_{k+1}, \ldots, c_n\} = Y \cap \{c_{k+1}, \ldots, c_n\} \end{array} \qquad (2)$$

A preferred conflict can now be defined:

Definition 7 *Let $\mathcal{P} := (\mathcal{B}, \mathcal{C}, <)$ be a totally ordered problem. A conflict C of $\mathcal{P}$ is a* preferred conflict *of $\mathcal{P}$ iff there is no other conflict C^* of $\mathcal{P}$ s.t. $C^* <_{antilex} C$.*

Definition 8 *Let $\mathcal{P} := (\mathcal{B}, \mathcal{C}, \prec)$ be a partially ordered problem. A conflict C of $\mathcal{P}$ is a* preferred conflict *of $\mathcal{P}$ iff there is a linearization $<$ of $\prec$ s.t. C is a preferred conflict of $(\mathcal{B}, \mathcal{C}, <)$*

A preferred conflict C is minimal (irreducible) meaning that each proper subset of C has a solution. If no preferences are given ($\prec$ is empty), then the minimal conflicts and the preferred conflicts coincide. If $\prec$ is a strict total order and $\mathcal{B} \cup \mathcal{C}$ is inconsistent, then $\mathcal{P}$ has a unique preferred conflict. Hence, a total order uniquely specifies or characterizes the conflict that will be detected by our algorithms. It is also interesting to note that the constraint graph consisting of the constraints of a minimal conflict is connected.

Proposition 1 *Let C be a conflict for a CSP $\mathcal{P} := (\emptyset, \mathcal{C}, \prec)$. If C is a minimal conflict of $\mathcal{P}$, then the constraint graph of C consists of a single strongly connected component.*

There is a strong duality between relaxations and conflicts with a rich mathematical structure. The relationships between $<_{antilex}$ and $<_{lex}$ can be stated as follows:

Proposition 2 *$X <_{antilex} Y$ iff $Y (<^{-1})_{lex} X$.*

Conflicts correspond to the complements of relaxations of the negated problem with inverted preferences:

Proposition 3 *Let $\neg c_j \succ' \neg c_i$ iff $c_i \prec c_j$. R is a preferred relaxation (conflict) of $(\mathcal{B}, \mathcal{C}, \prec)$ iff $\{\neg c \mid c \in \mathcal{C} - C\}$ is a preferred conflict (relaxation) of $(\neg\mathcal{B}, \{\neg c \mid c \in \mathcal{C}\}, \succ')$.*

The definition of preferred relaxations and preferred conflicts can be made constructive, thus providing the basis for the explanation and relaxation algorithms. Consider a totally ordered problem $\mathcal{P} := (\mathcal{B}, \mathcal{C}, <)$ s.t. $\mathcal{B}$ is consistent, but not $\mathcal{B} \cup \mathcal{C}$. We enumerate the elements of $\mathcal{C}$ in increasing $<$-order $c_1, \ldots, c_n$. We construct the preferred relaxation of $\mathcal{P}$ by $R_0 := \emptyset$ and

$$R_i := \begin{cases} R_{i-1} \cup \{c_i\} & \text{if } \mathcal{B} \cup R_{i-1} \cup \{c_i\} \text{ has a solution} \\ R_{i-1} & \text{otherwise} \end{cases}$$

The preferred conflict of $\mathcal{P}$ is constructed in the reverse order. Let $C_n := \mathcal{C}$ and

$$C_i := \begin{cases} C_{i+1} - \{c_i\} & \text{if } \mathcal{B} \cup C_{i+1} - \{c_i\} \text{ has no solution} \\ C_{i+1} & \text{otherwise} \end{cases}$$

Adding a constraint to a relaxation thus corresponds to the retraction of a constraint from a conflict. As a consequence of this duality, algorithms for computing relaxations can be reformulated for computing conflicts and vice versa.

Preferred conflicts explain why best elements cannot be added to preferred relaxations. In fact, the $<$-minimal element that is not contained in the preferred relaxation R of a problem $\mathcal{P} := (\mathcal{B}, \mathcal{C}, <)$ is equal to the $<$-maximal element of the preferred conflict C of $\mathcal{P}$:

Proposition 4 *If C is a preferred conflict of $\mathcal{P} := (\mathcal{B}, \mathcal{C}, <)$ and R is a preferred relaxation of $\mathcal{P}$, then the $<$-minimal element of $\mathcal{C} - R$ is equal to the $<$-maximal element of C.*

Preferred conflicts permit an incremental construction of preferred relaxations while avoiding unnecessary commitments. For example, consider $\alpha \prec \beta \prec \gamma$ and the background constraints $\neg\beta \vee \neg\delta$, $\neg\gamma \vee \neg\delta$. Then $\{\beta, \delta\}$ is a preferred conflict for the order $\alpha < \beta < \gamma < \delta$. Since γ is neither an element of the conflict $\{\beta, \delta\}$, nor $\prec$-preferred to any of its elements, we can move it behind δ, thus getting a new linearization $\alpha <' \beta <' \delta <' \gamma$. The linearizations $<$ and $<'$ have the same preferred conflict and the same preferred relaxation. This observation shows that we can construct the head (or start) of a preferred relaxation from a preferred conflict C of $\prec$. We identify a worst element for C, precede it by the other elements of C and all constraints $Pred(C)$ that are preferred to an element of C. We then reduce the problem to

$$(\mathcal{B} \cup Pred(C) \cup C - \{\alpha\}, \mathcal{C} - Pred(C) - C, \prec)$$

Please note that non-preferred conflicts such as $\{\gamma, \delta\}$ include irrelevant constraints such as γ and do not allow this reduction of the problem. Given different preferred conflicts, we can construct different preferred relaxations. This is interesting in an interactive setting where the user wants to control the selection of a relaxation.

Computing Preferred Explanations

We compute preferred conflicts and relaxations by following the constructive definitions. The basic algorithm will (arbitrarily) choose one linearization $<$ of the preferences $\prec$, thus fixing the resulting conflict or relaxation. It then inspects one constraint after the other and determines whether it belongs to the preferred conflict or relaxation of $<$. It thus applies a consistency checker *isConsistent*(C) to a sequence of subproblems. In this section, we assume that the consistency checker is complete and returns true if C has a solution. Otherwise, it returns false. For a CSP, complete consistency checking can be achieved as follows:

- arc consistency AC is sufficient for tree-like CSPs.
- systematic tree search maintaining AC is needed for arbitrary CSPs.

Incomplete checkers can provide non-minimal conflicts, as will be discussed in the next section.

Iterative Addition and Retraction

The basic algorithm successively maps a problem to a simpler problem. Initially, it checks whether the background is

```
Algorithm QUICKXPLAIN($\mathcal{B}, \mathcal{C}, \prec$)
 1.  if isConsistent($\mathcal{B} \cup \mathcal{C}$) return 'no conflict';
 2.  else if $\mathcal{C} = \emptyset$ then return $\emptyset$;
 3.  else return QUICKXPLAIN'($\mathcal{B}, \mathcal{B}, \mathcal{C}, \prec$);
Algorithm QUICKXPLAIN'($\mathcal{B}, \Delta, \mathcal{C}, \prec$)
 4.  if $\Delta \neq \emptyset$ and not isConsistent($\mathcal{B}$) then return $\emptyset$;
 5.  if $\mathcal{C} = \{\alpha\}$ then return $\{\alpha\}$;
 6.  let $\alpha_1, \ldots, \alpha_n$ be an enumeration of $\mathcal{C}$ that respects $\prec$;
 7.  let $k$ be split($n$) where $1 \leq k < n$;
 8.  $\mathcal{C}_1 := \{\alpha_1, \ldots, \alpha_k\}$ and $\mathcal{C}_2 := \{\alpha_{k+1}, \ldots, \alpha_n\}$;
 9.  $\Delta_2 :=$ QUICKXPLAIN'($\mathcal{B} \cup \mathcal{C}_1, \mathcal{C}_1, \mathcal{C}_2, \prec$);
10.  $\Delta_1 :=$ QUICKXPLAIN'($\mathcal{B} \cup \Delta_2, \Delta_2, \mathcal{C}_1, \prec$);
11.  return $\Delta_1 \cup \Delta_2$;
```

Figure 1: Divide-and-Conquer for Explanations.

inconsistent. If $\mathcal{C}$ is empty, then the problem can immediately be solved:

Proposition 5 *Let $\mathcal{P} := (\mathcal{B}, \mathcal{C}, \prec)$. If $\mathcal{B}$ is inconsistent then the empty set is the only preferred conflict of $\mathcal{P}$ and $\mathcal{P}$ has no relaxation. If $\mathcal{B} \cup \mathcal{C}$ is consistent then $\mathcal{C}$ is the only preferred relaxation of $\mathcal{P}$ and $\mathcal{P}$ has no conflict.*

If $\mathcal{C}$ is not empty, then the algorithm follows the constructive definition of a preferred relaxation. In each step, it chooses a $\prec$-minimal element α and removes it from $\mathcal{C}$. If $\mathcal{B} \cup R \cup \{\alpha\}$ is consistent, α is added to R. A preferred relaxation can be computed by iterating these steps.

The constructive definition of a preferred conflict starts by checking the consistency of the complete set $\mathcal{C} \cup \mathcal{B}$ and then removes one constraint after the other. Whereas the addition of a constraint is an incremental operation for a consistency checker, the removal is a non-incremental operations. Therefore, the computation of a conflict starts with the process of constructing a relaxation R_i. When the first inconsistency is obtained, then we have detected the best element α_{k+1} that is removed from the preferred relaxation. According to Proposition 4, α_{k+1} is the worst element of the preferred conflict. Hence, the preferred conflict is a subset of $R_k \cup \{\alpha_{k+1}\}$ and C_{n-k} is equal to $\{\alpha_{k+1}\}$. We now switch over to the constructive definition of preferred conflicts and use it to find the elements that are still missing.

Example 2 *As a simple benchmark problem, we consider n boolean variables, a background constraint $\sum_{i=1}^{n} k_i \cdot x_i < 3n$ (with $k_i = n$ for $i = 9, 10, 12$ and $k_i = 1$ otherwise) and n constraints $x_i = 1$. The algorithm introduces the constraints for $i = 1, \ldots, 12$, then switches over to a removal phase.*

Divide-and-Conquer for Explanations

We can significantly accelerate the basic algorithms if conflicts are small compared to the number of constraints. In this case, we can reduce the number of consistency checks if we remove whole blocks of constraints. We thus split $\mathcal{C}$ into subsets $\mathcal{C}_1$ and $\mathcal{C}_2$. If the remaining problem $\mathcal{C}_1$ is inconsistent, then we can eliminate all constraints in $\mathcal{C}_2$ while needing a single check. Otherwise, we have to re-add some of the constraints of $\mathcal{C}_2$. The following property explains how the conflicts of the two subproblems can be assembled.

Proposition 6 *Suppose $\mathcal{C}_1$ and $\mathcal{C}_2$ are disjoint and that no constraint of $\mathcal{C}_2$ is preferred to a constraint of $\mathcal{C}_1$:*

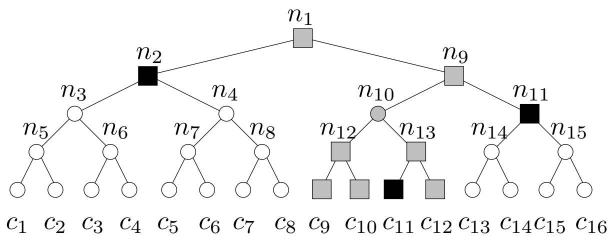
1. *If Δ_1 is a preferred relaxation of $(\mathcal{B}, \mathcal{C}_1, \prec)$ and Δ_2 is a preferred relaxation of $(\mathcal{B} \cup \Delta_1, \mathcal{C}_2, \prec)$, then $\Delta_1 \cup \Delta_2$ is a preferred relaxation of $(\mathcal{B}, \mathcal{C}_1 \cup \mathcal{C}_2, \prec)$.*
2. *If Δ_2 is a preferred conflict of $(\mathcal{B} \cup \mathcal{C}_1, \mathcal{C}_2, \prec)$ and Δ_1 is a preferred conflict of $(\mathcal{B} \cup \Delta_2, \mathcal{C}_1, \prec)$, then $\Delta_1 \cup \Delta_2$ is a preferred conflict of $(\mathcal{B}, \mathcal{C}_1 \cup \mathcal{C}_2, \prec)$.*

We divide an inconsistent problem in this way until we obtain subproblems of the form $\mathcal{P}' := (\mathcal{B}, \{\alpha\}, \prec)$, where all but one constraint are in the background. We then know that $\mathcal{B} \cup \{\alpha\}$ is inconsistent. According to Proposition 5, it is sufficient to check whether $\mathcal{B}$ is consistent in order to determine whether $\{\alpha\}$ is a preferred conflict of $\mathcal{B}$. Algorithm QUICKXPLAIN (cf. Figure 1) exploits propositions 5 and 6. It is parameterized by a split-function that chooses the subproblems for a chosen linearization of $\prec$ (see line 6):

Theorem 1 *The algorithm* QUICKXPLAIN*($\mathcal{B}, \mathcal{C}, \prec$) always terminates. If $\mathcal{B} \cup \mathcal{C}$ has a solution then it returns 'no conflict'. Otherwise, it returns a preferred conflict of $(\mathcal{B}, \mathcal{C}, \prec)$.*

QUICKXPLAIN spends most of its time in the consistency checks. A subprocedure QUICKXPLAIN' is only called if $\mathcal{C}$ is a non-empty conflict and if a part of the background, namely $\mathcal{B} - \Delta$ has a solution. Figure 2 shows the call graph of QUICKXPLAIN' for example 2. If no pruning (line 4) occurs, then the call graph is a binary tree containing a leaf for each of the n constraints. This tree has $2n - 1$ nodes. The square nodes correspond to calls of QUICKXPLAIN' that test the consistency of the background (line 4). Successful tests are depicted by grey squares, whereas failing tests are represented by black squares. For example, the test fails for node n_{11}, meaning that n_{11} is pruned and that its subtree is not explored (indicated by white circles). The left sibling n_{10} of the pruned node n_{11} does not need a consistency check (line 4) and is depicted by a grey circle. If a test succeeds for a leaf, then its constraint belongs to the conflict (line 5) and will be added to the background.

If we choose *split*$(n) := \frac{n}{2}$ then subproblems are divided into smaller subproblems of same size and a path from the root to a leaf contains $\log n$ nodes. If the preferred conflict has k elements, then the non-pruned tree is formed of the k paths from the root node to the k leaves of those elements. In the best case, all k elements belong to a single subproblem that has $2k - 1$ nodes and there is a common path for all elements from the root to the root of this subproblem. This path has the length $log\ n - log\ k = log\frac{n}{k}$. In the worst case, the paths join in the top in a subtree of depth $log\ k$. Then we have k paths of length $log\frac{n}{k}$ from the leaves of this subtree to the leaves of the conflict. All other cases fall in between these extremes. For problems with one million of constraints, QUICKXPLAIN thus needs between 33 and 270 checks if the conflict contains 8 elements. Table 4 gives the complexities of different split-functions. For lines 2 and 3, the shortest path has length 1, but the longest one has length n.



n_1
n_2
n_9
n_3
n_4
n_{10}
n_{11}
n_5
n_6
n_7
n_8
n_{12}
n_{13}
n_{14}
n_{15}
c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8 c_9 c_{10} c_{11} c_{12} c_{13} c_{14} c_{15} c_{16}

Figure 2: Call graph for QUICKXPLAIN.

Method	Split	Best Case	Worst Case
1.	$split(n) = n/2$	$log\frac{n}{k} + 2k$	$2k \cdot log\frac{n}{k} + 2k$
2.	$split(n) = n - 1$	$2k$	$2n$
3.	$split(n) = 1$	k	$n + k$

Table 4: Number of Consistency Checks.

If a problem is decomposable and the preferred conflict is completely localized in one of the subproblems, say P^*, then the size of the conflict is bounded by the size of P^*. QUICKXPLAIN will prune all subtrees in the call graph that do not contain an element of P^* and thus discovers irrelevant subproblems dynamically. Similar to (Mauss & Tatar 2002), it thus profits from the properties of decomposable problems, but additionally takes preferences into account.

Further improvements of QUICKXPLAIN are possible if knowledge of the constraint graph is exploited. Once an element c of the conflict has been determined, all non-connected elements can be removed. If two elements c_1, c_2 have been detected and all paths from c_1 and c_2 go through a constraint from X, then at least one element of X belongs to the conflict. Hence, graph algorithms for strongly connected components and cut detection make QUICKXPLAIN more informed and enable deductions on conflicts.

Multiple Preferred Explanations

We use preference-based search (Junker 2002) to determine multiple preferred relaxations. It sets up a choice point each time a constraint c_i is consistent w.r.t. a partial relaxation R_{i-1}. The left branch adds c_i to R_{i-1} and determines preferred relaxations containing c_i. The right branch adds other constraints to R_{i-1} that entail $\neg c_i$. We can adapt PBS to the constructive definition of preferred conflicts. We set up a choice point when c_i is removed from C_{i+1}. The left branch removes c_i from C_{i+1} and determines preferred conflicts not containing c_i. The right branch removes other constraints from C_{i+1} such the removal of c_i leads to a solution.

Consistency Checking with Search

If search fails, but not constraint propagation, then the consistency checking of QUICKXPLAIN requires multiple searches through similar search spaces.

We consider a variant of example 1, where the type of each option needs to be chosen from a product catalogue in order to determine its precise price. Furthermore, we suppose that there are several compatibility constraint between those types. A solution consists of a set of options and their types such that the budget and the compatibility constraints are met. Propagation is insufficient to detect the infeasibility of an option if the constraint network contains one or several cycles.

Hence, the consistency checker will search for a solution to prove the consistency of a set X of constraints. If successful, QUICKXPLAIN adds further constraints Δ and checks $X \cup \Delta$. For example, X may contain a requirement for a CD-player and Δ may refine it by requiring a CD-player of type A if one is selected. In order to prove the consistency of X, the checker must be able to produce a solution S of X. If S contains a CD-player of type A, then it satisfies Δ and it is not necessary to start a new search. Or we may repair S by just changing the type of the CD-player. We therefore keep the solution S as witness for the consistency of X. This *witness of success* can guide the search for a solution of $X \cup \Delta$ by preferring the variable values in S. It can also avoid a re-exploration of the search tree for X if the new search starts from the search path that produced S.

If a consistency check fails for X, then QUICKXPLAIN removes some constraints Δ from X and checks $X - \Delta$. For example, X may contain requirements for all options, including that for a CD-player of type A. Suppose that the inconsistency of X can be proved by trying out all different metal colors. Now we remove the requirement for a CD-player of type A from X. If the CD-player type was not critical for the failure, then it is still sufficient to instantiate the metal color in order to fail. Otherwise, we additionally instantiate the type of the CD-player. Since these critical variables suffice to provoke a failure of search, we can keep them as *witness of failure* and instantiate them first when checking $X - \Delta$. Decomposition methods (Dechter & Pearl 1989) such as cycle cutset give good hints for identifying a witness of failure.

This analysis shows that QUICKXPLAIN does not need to start a search from scratch for each consistency check, but can profit from witnesses for failure and success. The witness of success guides a least-commitment strategy that tries to prove consistency, whereas a first-fail strategy is guided by a witness of failure and tries to prove inconsistency.

If problems are more difficult, but search of the complete problem fails in a specified time, then approximation techniques can be used. Firstly, QUICKXPLAIN can be stopped when it has found the k worst elements of a preferred conflict, which is sometimes sufficient. Secondly, a correct, but incomplete method can be used for consistency checking. An arc consistency based solver has these properties. Another example is tree search that is interrupted after a limited amount of time. If such a method reports false, QUICKXPLAIN knows that there is a failure and proceeds as usual. Otherwise, QUICKXPLAIN has no precise information about the consistency of the problem and does not remove constraints. As a consequence, it always returns a conflict, but not necessarily a minimal one. Hence, there is a trade-off between optimality of the results and the response time.

Related Work

Conflicts and relaxations are studied and used in many areas of automated reasoning such as truth maintenance systems (TMS), nonmononotonic reasoning, model-based diagnosis, intelligent search, and recently explanations for over-constrained CSPs. Whereas the notion of preferred relaxations found a lot of interest, e.g. in the form of extensions of prioritized default theories (Brewka 1989), the concept of a preferred explanation appears to be new. It is motivated by recent work on interactive configuration, where explanations should contain the most important user requirements.

Conflicts can be computed by recording and analyzing proofs or by testing the consistency of subsets. Truth maintenance systems elaborate the first approach and record the proof made by an inference system. Conflicts are computed from the proof on a by-need basis (Doyle 1979) or by propagating conflicts (de Kleer 1986) over the recorded proof. There have been numerous applications of TMS-techniques to CSPs, mainly to achieve more intelligent search behaviour, cf. e.g. (Ginsberg & McAllester 1994; Prosser 1993; Jussien, Debruyne, & Boizumault 2000). More recently, TMS-methods have been embedded in CSPs to compute explanations for CSPs (Sqalli & Freuder 1996).

The computation of minimal and preferred conflicts, however, requires the selection of a suitable proof, which can be achieved by selecting the appropriate subset controlled by preferences. Iterative approaches successively remove elements (Bakker *et al.* 1993) or add elements (de Siqueira N. & Puget 1988) and test conflict membership. QUICKXPLAIN unifies and improves these two methods by successively decomposing the complete explanation problem into subproblems of the same size. (Mauss & Tatar 2002) follow a similar approach, but do not take preferences into account. (de la Banda, Stuckey, & Wazny 2003) determine all conflicts by exploring a conflict-set tree. These checking-based methods for computing explanations work for any solver and do not require that the solver identifies its precise inferences. This task is indeed difficult for global n-ary constraints that encapsulate algorithms from graph theory. Moreover, subset checking can also be used to find explanations for linear programming as shown in (Chinneck 1997).

Conclusion

We have developed algorithms that compute preferred conflicts and relaxations of over-constrained problems and thus help developers and users of Constraint Programming to identify causes of an inconsistency, while focusing on the most important constraints. Since the algorithms just suppose the existence of a consistency checker, they can be applied to all kind of satisfiability problems, including CSPs, SAT, or different combinatorial problems such as graph coloring. A divide-and-conquer strategy significantly accelerates the basic methods, ensures a good scalability w.r.t. problem size, and provides the technological basis for the explanation facility of a principal industrial constraint programming tool (ILOG 2003b) and a CP-based configurator (ILOG 2003a), which is used in various B2B and B2C configuration applications.

QUICKXPLAIN has a polynomial response time for polynomial CSPs. For other problems, multiple searches through similar search spaces are needed. Search overhead can be avoided by maintaining witnesses for the success and failure of previous consistency checks. If response time is limited, the QUICKXPLAIN algorithm can compute an approximation of a minimal conflict by using an incomplete checker.

Acknowledgements

I thank my colleagues and the anonymous reviewers for helpful comments. Olivier Lhomme made significant contributions to the clarity of the paper.

References

Bakker, R. R.; Dikker, F.; Tempelman, F.; and Wognum, P. M. 1993. Diagnosing and solving over-determined constraint satisfaction problems. In *IJCAI-93*, 276–281.

Brewka, G. 1989. Preferred subtheories: An extended logical framework for default reasoning. In *IJCAI-89*, 1043–1048.

Chinneck, J. W. 1997. Finding a useful subset of constraints for analysis in an infeasible linear porgram. *INFORMS Journal on Computing* 9:164–174.

de Kleer, J. 1986. An assumption–based truth maintenance system. *Artificial Intelligence* 28:127–162.

de la Banda, M. G.; Stuckey, P. J.; and Wazny, J. 2003. Finding all minimal unsatisfiable subsets. In *PPDP 2003*, 32–43.

de Siqueira N., J. L., and Puget, J.-F. 1988. Explanation-based generalisation of failures. In *ECAI-88*, 339–344.

Dechter, R., and Pearl, J. 1989. Tree clustering for constraint networks. *Artificial Intelligence* 38:353–366.

Doyle, J. 1979. A truth maintenance system. *Artificial Intelligence* 12:231–272.

Ginsberg, M., and McAllester, D. 1994. GSAT and dynamic backtracking. In *KR'94*, 226–237.

ILOG. 2003a. ILOG JConfigurator V2.1. Engine programming guide, ILOG S.A., Gentilly, France.

ILOG. 2003b. ILOG Solver 6.0. User manual, ILOG S.A., Gentilly, France.

Junker, U., and Mailharro, D. 2003. Preference programming: Advanced problem solving for configuration. *AI-EDAM* 17(1):13–29.

Junker, U. 2002. Preference-based search and multi-criteria optimization. In *AAAI-02*, 34–40.

Jussien, N.; Debruyne, R.; and Boizumault, P. 2000. Maintaining arc-consistency within dynamic backtracking. In *CP'2000*, 249–261.

Mauss, J., and Tatar, M. 2002. Computing minimal conflicts for rich constraint languages. In *ECAI-02*, 151–155.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9:268–299.

Sqalli, M. H., and Freuder, E. C. 1996. Inference-based constraint satisfaction supports explanation. In *AAAI-96*, 318–325.