Return of the JTMS: Preferences Orchestrate Conflict Learning and Solution Synthesis

Ulrich Junker and Olivier Lhomme[1]

Abstract. We use a lexicographical preference order on the problem space to combine solution synthesis with conflict learning. Given two preferred solutions of two subproblems, we can either combine them to a solution of the whole problem or learn a 'fat' conflict which cuts off a whole subtree. The approach makes conflict learning more pervasive for Constraint Programming as it well exploits efficient support finding and compact representations of Craig interpolants.

1 Introduction

A justification-based truth maintenance system (JTMS) [6] allows a problem solver to record logical dependencies between deductions. A JTMS supports the retraction of constraints and provides explanations for the recorded deductions. The JTMS also introduced a technique which is now called conflict learning [10, 20]. Conflicts permit a search procedure to backtrack to the causes of a failure, while keeping choices in independent subproblems unchanged. As a consequence, we obtain an additive complexity instead of a multiplicative complexity for independent subproblems [9]. Although independent subproblems are frequent in configuration and design problems, they are rare in popular applications of Constraint Programming (CP) such as scheduling and resource allocation. Hence, conflict learning was seldom used in CP, although recent efforts are promising [14].

However, conflict learning proved to be essential for the recent success of SAT solvers [10, 20]. It can be observed that conflict learning reduces the complexity of a problem even if subproblems are not independent, but have some variables in common. Suppose a problem solver first determines a solution S_1 of subproblem $A(x, y)$ and then solves the subproblem $B(y, z)$ under the restrictions that S_1 imposes on the common variables y. If this restricted version of B has no solution, the problem solver backtracks and seeks a new solution S_2 of the subproblem A. It then solves subproblem B under the restrictions S_2 imposes on y. Hence, the subproblem B is solved several times and this for different solutions S_i of the subproblem A. Although these solutions are all different, their projection on the common variables y can be the same, meaning that the same restricted version of the subproblem B is solved again and again. This thrashing behaviour will in particular be very cumbersome if the whole problem has no solution and the whole solution space of A needs to be traversed. Conflict learning permits the problem solver to avoid this thrashing behaviour. If a subproblem B fails for a restriction C on the common variables y, the conflict $\neg C$ is incorporated into the problem A and avoids that solutions with restriction C on variables y are generated again. If the whole problem is inconsistent, then the set of learned conflicts is a Craig interpolant [4, 16]. Conflict learning thus permits a factorization of the problem solving. We can also show that conflict learning behaves like a lazy version of bucket elimination [5] if the buckets are chosen as subproblems. Hence, conflict learning is exponential in the induced width [5, 8] in the worst-case.

JTMS thus well supports factorization of problems provided it uses a static preference ordering between variables as proposed in [17] when choosing the culprits to be retracted. Moreover, JTMS can be used to maintain a lexicographically preferred solution. As described in [6], non-monotonic justifications can be used to encode preferences between the different values of a variable, which leads to an ordered disjunction [3]. This mechanism always activates the best assignment for a variable that is not defeated by a recorded conflict. If this best assignment occurs in a new conflict, JTMS can retract this assignment and activate the next one. This procedure will finally result in a lexicographically preferred solution, which will be updated if new constraints are added (e.g. new conflicts learned from interactions with other problems). However, if variable domains are large, then this procedure may try out many value assignments. An activated value assignment $y = w$ will only be abandoned if it occurs in a conflict. When seeking a support for an assignment $x = v$ on the constraint $C(x, y)$, d such assignments $y = w_i$ may be tried out, meaning that d conflicts have to be learned.

Modern algorithms that maintain arc consistency (MAC) [19] directly seek supports (i.e. solutions of a single constraint) without needing to learn so many conflicts. Moreover, the preferences between values can be used to activate preferred supports [2]. Provided with an assignment $x = v$, the constraint $C(xy)$ can thus determine the preferred support (v, w) for $x = v$ and activate the assignment $y = w$ without trying out all better value for y. An even further idea is to synthesize the preferred supports from different constraints into a lexicographically preferred solution. PrefAC [2] has this behaviour for tree-like CSPs if parent variables are preferred to child variables. However, the activation of preferred supports gets blocked for cyclic constraint networks when different preferred supports activate different values for the same variable.

It is possible to learn a conflict in those situations. In this paper, we show how to combine solution synthesis and conflict learning. Consider again the two subproblems $A(xy)$ and $B(yz)$. We first determine a preferred solution S_1 for A. We then determine a preferred solution S_2 for B under the restriction that it must not be better than S_1 on the common variables y. If both solutions assign the same values to the common variables y, then we can synthesize them to a solution of the whole problem. Otherwise, we learn a 'fat' conflict that cuts off a whole subtree in the search space of y. All assignments between S_1 (projected to y) and S_2 (projected to y) are infeasible. Hence, less solutions of A need to be generated in turn and conflict learning becomes effective for CSPs even if domains are large. By introducing fat conflicts, we enable a return of the JTMS to CP.

[1] ILOG S.A., France, email: {ujunker, olhomme}@ilog.fr

The paper is organized as follows: Firstly, we introduce solution synthesis and conflict learning based on a lexicographical order and show how they can be used to decompose a problem into two subproblems. Secondly, we apply this decomposition recursively until we obtain buckets, i.e. constraints having a variable in common. We show how to solve the buckets by reducing lexicographical bounds.

2 Preference-based Interpolation

2.1 Constraints and Preferences

Throughout this paper, we consider combinatorial problems that are modelled by a set of constraints C formulated over a set of variables X having the domain $\mathcal{D}$. A variable x from X can be interpreted by any value v from the domain $\mathcal{D}$ and we describe such a variable-value assignment by the constraint $x = v$. An *assignment* to X is a set that contains exactly one value assignment for each variable in X. The set of all assignments to X is called the *problem space* of X. and we denote it by $\mathcal{S}(X)$. We project an assignment σ to a set Y by defining $\sigma[Y] := \{(x = v) \in \sigma \mid x \in Y\}$. If the assignment σ contains $x = v$, we denote the value v of x by $\sigma(x)$. Furthermore, we define $\sigma(x_1, \ldots, x_n)$ as the vector $(\sigma(x_1), \ldots, \sigma(x_n))$.

A constraint c is specified by an n-ary set of variables X_c, called scope, and an n-ary 'relation' R_c, which is described by a subset of $\mathcal{S}(X_c)$. The relation can be specified by a predicate, a numerical or logical expression, a set of allowed tuples, or a set of forbidden tuples. An assignment σ to X is a *solution* of the constraint c iff $\sigma[X_c] \in R_c$ holds.

The set of variables of a set of constraints C is denoted by $X_C := \bigcup_{c \in C} X_c$. An assignment σ (to X) is a *solution* of the constraints C iff it is a solution of all constraints $c \in C$. It is often convenient to replace a constraint set C by the conjunction $\bigwedge_{c \in C} c$. Similarly, we can replace a conjunction $\bigwedge_{i=1}^{k} c_i$ be the set of its conjuncts $\{c_1, \ldots, c_k\}$. All statements about the constraints C_1, C_2 can thus be applied to CSPs and vice versa. Two constraints C_1 and C_2 with same scope are equivalent iff they have the same solutions.

Throughout this paper, we assume that the user can express preferences between the possible values of a variable x_i and we consider some linearization $>_i$ of those preferences as indicated in [13]. Furthermore, we suppose that the user can express an importance ordering between the variables and we consider a linearization $>$ of this importance ordering (cf. [13]). The ordering $>$ can be used to sort a set of variables. Let $X^>$ denote the sorted vector $(x_{\pi_1}, \ldots, x_{\pi_n})$ where X is equal to $\{x_1, \ldots, x_n\}$ and π is the permutation of $1, \ldots, n$ that satisfies $x_{\pi_1} > \ldots > x_{\pi_n}$. We define the lexicographical ordering $>_{lex}$ between two vectors of values $v := (v_{\pi_1}, \ldots, v_{\pi_n})$ and $w := (w_{\pi_1}, \ldots, w_{\pi_n})$: $v >_{lex} w$ holds iff there exists an i among $1, \ldots, n$ such that $v_{\pi_j} = w_{\pi_j}$ for $j = 1, \ldots, i-1$ and $v_{\pi_i} >_{\pi_i} w_{\pi_i}$. A solution σ of C is called *preferred solution* of C iff there is no other solution σ^* s.t. $\sigma^*(X^>) >_{lex} \sigma(X^>)$. As in [13], we use an optimization operator *Lex*$(X^>)$ to describe the set of preferred solutions. The operator maps the constraint C to a new constraint *Lex*$(X^>)(C)$ that is satisfied by exactly the preferred solution of C. This notation allows us to reason about optimization results inside the constraint language.

The lexicographical ordering can be used to formulate a constraint in the constraint language. Let x be a sorted vector of variables and v be a sorted vector of values. The constraint $x >_{lex} v$ is equivalent to

$$\bigvee_{i=1}^{n} \left(\bigwedge_{j=1}^{i-1} x_{\pi_j} = v_{\pi_j} \wedge x_{\pi_i} >_{\pi_i} v_{\pi_i} \right) \tag{1}$$

We obtain similar constraints for $\geq_{lex}$, $<_{lex}$, and $\leq_{lex}$. The lexicographical constraint allows us to express a property which is fundamental for this paper. If σ^* is a solution of *Lex*$(X^>)(C)$ then C implies $X^> \leq_{lex} \sigma^*(X^>)$.

2.2 Solution Synthesis and Fat Conflicts

We now consider two sets of constraints C_1 and C_2. Each set represents a subproblem. We will show how the whole problem $C_1 \cup C_2$ can be solved by solving the two subproblems independently. The only information that we exchange between the two subproblems are supplementary constraints that are formulated on the common variables of C_1 and C_2, i.e. $X_{C_1} \cap X_{C_2}$. If the whole problem $C_1 \cup C_2$ is inconsistent, then it is well-known that there is a constraint C' that contains only the common variables, that is implied by C_2, and that is inconsistent w.r.t. C_1. The constraint C' is called a *Craig interpolant* [4, 16]. In this section, we will show that we can either synthesize two solutions of the subproblems into a solution of the whole problem or compute a Craig interpolant by generating a sequence of conflicts. We use the lexicographical preference ordering to determine which solutions and conflicts may be generated in each step.

We can only synthesize two solutions of the subproblems if the importance ordering $>$ on the variables ensures that the variables added by C_2 are less important than the variables in C_1. We first lift the ordering on variables to sets of variables: $X_1 \succ X_2$ iff $x_i > x_j$ for all $x_i \in X_1, x_j \in X_2$. We then extend it to constraints:

Definition 1 *A constraint C_1 is more important than a constraint C_2 in the ordering $>$, written $C_1 \succ C_2$, iff $X_{C_1} \succ X_{C_2} - X_{C_1}$.*

An example is given be the two constraints in table 1. The domain is $\mathcal{D} := \{1, 2, 3\}$. Greater values are preferred for all variables and the variables are ordered as follows: $x_1 > x_2 > x_3 > x_4$.

From now on, we suppose that C_1 is more important than C_2. In section 3.1, we will show that we can always find subproblems that meet this condition. In the sequel, we write X_1 for the ordered tuple of variables $X_{C_1}^>$ of C_1, X_2 for the ordered tuple of variables $X_{C_2}^>$ of C_2, Δ for the ordered tuple of variables $(X_{C_1} \cap X_{C_2})^>$ occurring in both subproblems, and Ω for the ordered tuple of variables $(X_{C_1} \cup X_{C_2})^>$ occurring in the whole problem.

We start the whole solving process by seeking a preferred solution for the first subproblem C_1. If this problem is inconsistent, then $C_1 \cup C_2$ is inconsistent as well. Hence, the failure of the first subproblem leads to a global failure and we can stop in this case.

If the first subproblem has the preferred solution σ_1, then we project it on the common variables by taking $\sigma_1(\Delta)$. If we want to find a solution σ_2 of the second subproblem that is compatible with σ_1, then this other solution cannot be lexicographically better than σ_1 on the common variables Δ. Indeed, if $\sigma_2(\Delta) >_{lex} \sigma_1(\Delta)$ holds, then the two solutions cannot be synthesized. We therefore temporarily add a lexicographical constraint $\Delta \leq_{lex} \sigma_1(\Delta)$ to C_2 when determining a preferred solution of the second subproblem. Please note that this constraint involves only variables that are shared by both subproblems. If the enforced version of the second subproblem has no solution, then the added constraint cannot be true and we can add its negation to the first subproblem. We thus obtain a first part of an interpolant:

Proposition 1 *Let C_1, C_2 be two constraint sets and X_1, Δ as defined above. Let σ_1 be a solution of Lex$(X_1)(C_1)$. If $C_2 \cup \{\Delta \leq_{lex} \sigma_1(\Delta)\}$ is inconsistent, then*

$$C_1^* := C_1 \cup \{\Delta >_{lex} \sigma_1(\Delta)\}$$

Table 1. Two constraints $C_1(x_1, x_2, x_3)$ and $C_2(x_2, x_3, x_4)$

x_1	x_2	x_3		x_2	x_3	x_4	
3	2	2	(1. σ_1)	3	3	2	
3	1	3		3	1	3	(4. σ_2')
1	3	2	(3. σ_1')	2	3	3	
1	2	2		2	3	1	
1	2	1		1	2	3	(2. σ_2)
1	1	3		1	1	2	

is a constraint that is not satisfied by σ_1 and $C_1^* \cup C_2$ is equivalent to $C_1 \cup C_2$. Furthermore, $C_1^* \succ C_2$ if $C_1 \succ C_2$.

If the enforced version of the second subproblem has the preferred solution σ_2, then we need to check whether σ_1 and σ_2 coincide on the common variables Δ. If they do, then we can synthesize the two solutions. Moreover, we also know that the result is a preferred solution as C_2 only adds less important variables. This property is important if we use the whole procedure recursively. However, if the solutions do not coincide on the common variables, then we can remove all assignments to Δ which are between $\sigma_1(\Delta)$ and $\sigma_2(\Delta)$ with respect to the lexicographical ordering. Whereas standard conflict learning only removes a branch in a subproblem, we remove a whole subtree. Hence, we learn 'fat' conflicts:

Proposition 2 *Let C_1, C_2 be two constraint sets and X_1, X_2, Δ, Ω as defined above. Let σ_1 be a solution of Lex$(X_1)(C_1)$ and σ_2 be a solution of Lex$(X_2)(C_2 \cup \{\Delta \leq_{lex} \sigma_1(\Delta)\})$.*

1. *If $\sigma_1(\Delta) = \sigma_2(\Delta)$ and $C_1 \succ C_2$ then $\sigma_1 \cup \sigma_2$ is a solution of Lex$(\Omega)(C_1 \cup C_2)$.*
2. *If $\sigma_1(\Delta) \neq \sigma_2(\Delta)$ then*

$$C_1^* := C_1 \cup \{\Delta \leq_{lex} \sigma_2(\Delta) \vee \Delta >_{lex} \sigma_1(\Delta)\}$$

is a constraint that is not satisfied by σ_1 and $C_1^* \cup C_2$ is equivalent to $C_1 \cup C_2$. Furthermore, $C_1^* \succ C_2$ if $C_1 \succ C_2$.

This proposition summarizes the approach of this paper. The first case corresponds to solution synthesis. The second case corresponds to conflict learning. Both are coordinated by the global preference ordering.

Compared to MAC [19], the constraints do not communicate conjunctions of domain constraints (such as $x_i \neq v_i$) to each other, but lexicographical constraints between a subset of variables and a vector of values. The result of this communication is either solution synthesis or a 'fat' conflict. As an effect, the number of iterations for eliminating inconsistent values can be reduced. In the example of table 1, the inconsistency of the problem is proved in the third iteration.

2.3 Lex-based Interpolation

We now develop an interpolation algorithm based on the principles developed in the last section. It basically iterates those principles until a preferred solution of the whole problem has been found or a Craig interpolant has been added to the first subproblem, thus making it inconsistent.

The algorithm LEXINTERPOLATE is shown in figure 2. In each loop iteration, it first solves the first subproblem which consists of C_1 and of an initially empty set I containing the learned conflicts. If the first subproblem has no solution, then the whole problem is inconsistent and the algorithm stops. Otherwise, it takes the preferred

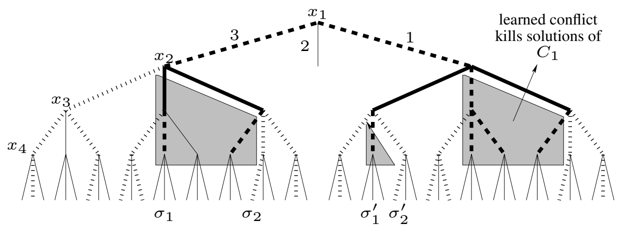

Figure 1. Effect of conflict learning on the search tree of C_1.

solution σ_1 of the first subproblem and formulates a lexicographical upper bound constraint on the common variables Δ and adds it temporarily to the second subproblem. If the second subproblem produces a preferred solution σ_2 that matches σ_1 then both solutions are combined into a preferred solution of the whole problem and the algorithm stops. In all other cases, a new conflict is learned and added to the interpolant. This conflict invalidates the previous solution σ_1 of the first subproblem, meaning that this solution will be revised in the next iteration. Hence, each iteration eliminates at least one assignment in the problem space of Δ meaning that the algorithm terminates for finite domains.

Theorem 1 *Suppose that $C_1 \succ C_2$. The algorithm* LEXINTERPOLATE*(C_1, C_2) terminates after at most $O(d^{|\Delta|})$ steps. If $C_1 \cup C_2$ is consistent, then it returns a solution of Lex$(\Omega)(C_1 \cup C_2)$. Otherwise, it returns an inconsistency and the set I of the learned conflicts is a Craig interpolant for C_2, C_1.*

The Craig interpolant in the example of table 1 is as follows:

$$\{((x_2 \leq 1) \vee (x_2 = 1 \wedge x_3 \leq 2)) \vee ((x_2 > 2) \vee (x_2 = 2 \wedge x_3 > 2)),$$
$$((x_2 \leq 3) \vee (x_2 = 3 \wedge x_3 \leq 1)) \vee ((x_2 > 3) \vee (x_2 = 3 \wedge x_3 > 2))\}$$

Figure 1 shows how these conflicts are learned. The upper three levels of the figure show the search tree for C_1. The lower three levels show two copies of the search tree of C_2 which are superposed with the one of C_1. Solutions of C_1 are displayed by dashed branches and solutions of C_2 are displayed by dotted branches (straight lines are obtained when both overlap). The shadowed areas show the conflicts. The first conflict is learned in the first copy of C_2's search tree. As a consequence, it also appears in the right copy. As it has been added to C_1, it now kills several solutions of C_1.

There are several ways to improve the algorithm. For example, we can add a constraint $X_1 \leq_{lex} \sigma_1(X_1)$ to C_1 recording the fact that there is no better solution than σ_1. When the first problem is resolved again in later iterations, then the search can immediately start from σ_1 and the part of the search space that is lexicographically greater than σ_1 need not be searched again. Furthermore, the first subproblem may consist of several independent parts and the addition of the conflict may concern only one of those parts. We need to keep a lexicographical upper bound constraints on each independent part in order to ensure that solving complexity is additive for the independent parts and not multiplicative [9].

Another issue are functional constraints such as $x_1 + x_2 = x_3$, in particular if this constraint belongs to the second subproblem and all variables belong to Δ. When we learn a conflict for Δ, the value of x_3 depends on those of x_1 and x_2, meaning that we can simplify the conflict by suppressing x_3. It is therefore worth to keep track of functional dependencies between variables and to detect a subset of Δ by a conflict analysis similar to [6, 10, 20]. The elaboration of this algorithm is a topic of future work.

3 Solution Synthesis by Support Activation

3.1 The Activation Process

We now apply the algorithm LEXINTERPOLATE to an arbitrary constraint network C and to an arbitrary total ordering $>$ of the variables of C. The idea is to recursively decompose the constraints C into subproblems. We observe that decomposition only makes sense if the second subproblem has a variable that is not contained in the first subproblem. As a consequence, we stop decomposition if all the constraints in a subproblem share the last variable of the subproblem. In [5], this is called a *bucket*. There are different ways to decompose a problem into buckets. In this paper, we visit the variables of C in the ordering $X^{>} = (x_{\pi_1}, \ldots, x_{\pi_n})$ that is produced by $>$. For each variable x_{π_i}, we define the two subproblems $C_1^{(i)}$ and $C_2^{(i)}$:

1. $C_1^{(i)}$ is the set of all constraints from C that contain only variables from $x_{\pi_1}, \ldots, x_{\pi_{i-1}}$
2. $C_2^{(i)}$ is the set of all constraints from C that contain only variables from $x_{\pi_1}, \ldots, x_{\pi_i}$ and that contain the variable x_{π_i}.

The two sets are disjoint. The first subproblem for the first variable $C_1^{(1)}$ is equal to the empty set. We consider the whole problem of a variable x_{π_i}, which is the union $C_1^{(i)} \cup C_2^{(i)}$ of the two subproblems. This whole problem is equal to $C_1^{(i+1)}$ if $i < n$. The whole problem of the last variable is equal to C. The second subproblem $C_2^{(i)}$ is the bucket for the variable x_{π_i} as all of its constraints contain x_{π_i} as their last variable. The first subproblem of a variable is more important than its second subproblem, i.e. $C_1^{(i)} \succ C_2^{(i)}$ for $i = 1, \ldots, n$.

Each decomposition thus meets the requirements of algorithm LEXINTERPOLATE and can be solved by it. Hence, we apply LEXINTERPOLATE in a sequence, activating one variable after the other. When we activate a variable x_{π_i} for $i > 1$, we already have a solution for its first subproblem as it is equal to the whole problem of the previous variable. We just need to find a solution for the bucket of the variable while taking into account the lexicographical constraint $\Delta \leq_{lex} \sigma_1(\Delta)$ that is formulated on all variables of the bucket except for x_{π_i}. If the second subproblem alone (i.e. without lexicographical constraint) is inconsistent, then this inconsistency will only be detected after the first subproblem is solved. If the second subproblem is smaller in size than the first subproblem, we will check the consistency of the second subproblem before solving the first one.

The whole process has some similarities to bucket elimination (BE). BE processes the variables from the last one to the first one. It eliminates a variable x_{π_i} from its bucket by deriving a new constraint between the remaining variables of the bucket. This constraint is a Craig interpolant for $C_1^{(i)}$ and $C_2^{(i)}$ if the problem is inconsistent and thus corresponds to the set of conflicts learned by LEXINTERPOLATE. Otherwise, LEXINTERPOLATE learns only a part of those constraints and thus behaves like a lazy form of bucket elimination.

3.2 Heavy Supports

We now discuss how to determine a solution for the bucket $B := C_2^{(i)}$ of variable $y := x_{\pi_i}$. When searching this solution, we have already determined a preferred solution σ_1 for the problem $C_1^{(i)}$. This solution assigns a value to each variable of the bucket except for y. Let Δ be the ordered vector of all those variables, i.e. $\Delta := (X_B - \{y\})^{>}$. As explained in section 2.2, we add a lexicographical upper bound constraint $\Delta \leq_{lex} \sigma_1(\Delta)$ when seeking a preferred solution of the bucket. This constraint can easily be extended to an equivalent constraint that covers the variable y as well. We simply add to σ the assignment of the variable y to its best value v^*, thus obtaining $\sigma := \sigma_1 \cup \{y = v^*\}$. The best value of y is simply the value in the domain of y that is maximal w.r.t. the ordering $>_{\pi_i}$. Let Y be the ordered vector of the variables in the bucket, i.e. $Y := X_B^{>}$. Our problem then consists in finding a preferred solution of the bucket B and the lexicographical constraint $Y \leq_{lex} \sigma(Y)$.

If the bucket contained only a single constraint, then the problem would reduce to that of finding a support of the constraint that is worse than or equal to $\sigma(Y)$. We first test whether $\sigma(Y)$ satisfies the constraint. If not, we seek the next best tuple according to the lexicographical order. If the bucket contains multiple constraints, all defined on the same variables Y, then we can iterate this approach. Each time, we invoke a constraint with a given lexicographical upper bound σ, it will reduce this to a new upper bound. We iterate this approach until the given upper bound satisfies all the constraints or until we encounter a constraint that cannot be satisfied under the reduced bound. The solution obtained in the first case combines the supports of all the constraints of variable y. We therefore call it a *heavy support*. Whereas classic supports indicate that a single variable-value assignment is consistent w.r.t. a constraint, a heavy support indicates that a bucket has a solution under a given lexicographical bound.

A bucket can contain multiple constraints which have some variables in common, but not all. An example is

$$c_1 := (x_1 = 2 \wedge x_3 = 2 \wedge y = 1) \vee (x_1 = 2 \wedge x_3 = 1 \wedge y = 2)$$
$$c_2 := (x_1 = 2 \wedge x_2 = 2 \wedge y = 2) \vee (x_1 = 1 \wedge x_2 = 1 \wedge y = 1)$$

Suppose that higher values are preferred for all variables and that the variables are ordered as follows: $x_1 > x_2 > x_3 > y$. If the initial bound is $\{x_1 = 2, x_2 = 2, x_3 = 2, y = 2\}$, we first invoke c_1, which reduces this bound to $\{x_1 = 2, x_2 = 2, x_3 = 2, y = 1\}$. If the constraint c_2 ignores the variable x_3, then it will return $\{x_1 = 1, x_2 = 1, y = 1\}$ as best support under $\{x_1 = 2, x_2 = 2, y = 1\}$. However, if x_3 is taken into account, then new bound should be $\{x_1 = 2, x_2 = 2, x_3 = 1, y = 2\}$. It is obtained by reducing the value for x_3 and by choosing the tuple $\{x_1 = 2, x_2 = 2, y = 2\}$ of c_2. Hence, a constraint can modify the bound on a variable even if it does not contain it. We capture this by the following definition:

Definition 2 *Let σ be an assignment to X and c be a constraint s.t. $X_c \subseteq X$. An assignment σ' to X is a* support *for c under σ iff (1) $\sigma' \leq_{lex} \sigma$ and (2) σ' satisfies c, i.e. $\sigma'[X_c] \in R_c$. An assignment σ' is a* preferred support *for c under σ iff σ' is a support for c under σ and there is no other support σ^* for c under σ s.t. $\sigma' <_{lex} \sigma^*$.*

A preferred support for c under σ can be found by a generic algorithm. Customized algorithms for certain types of constraints (tables with forbidden tuples; lex constraints; numerical constraints) can speed up support finding and are a topic of future work.

Algorithm LEXINTERPOLATE(C_1, C_2)

```
1.  $I := \emptyset$;
2.  while true do
3.      $\sigma_1 := Lex(X_1)(C_1 \cup I)$;
4.      if $\sigma_1$ is an inconsistency then return $\bot$;
5.      $\sigma_2 := Lex(X_2)(C_2 \cup \{\Delta \leq_{lex} \sigma_1(\Delta)\})$;
6.      if $\sigma_2$ is an inconsistency then
7.          $I := I \cup \{\Delta >_{lex} \sigma_1(\Delta)\}$;
8.      else if $\sigma_1(\Delta) = \sigma_2(\Delta)$ then
9.          return $\sigma_1 \cup \sigma_2$;
10.     else $I := I \cup \{\Delta \leq_{lex} \sigma_2(\Delta) \vee \Delta >_{lex} \sigma_1(\Delta)\}$;
```

Figure 2. Algorithm for lexicographic interpolation.

Algorithm HEAVYSUPPORT(B, σ)

```
1.  $Q := \{c \in B \mid \sigma[X_c] \notin R_c\}$;
2.  while $Q \neq \emptyset$ do
3.      choose best $c$ from $Q$;
4.      if $c$ has a support under $\sigma$ then
5.          let $\sigma'$ be the preferred support of $c$ under $\sigma$;
6.          set $\sigma$ to $\sigma'$;
7.      else return $\perp$.
8.      $Q := \{c \in B \mid \sigma[X_c] \notin R_c\}$;
9.  return $\sigma$.
```

Figure 3. Algorithm for finding heavy supports.

An algorithm for finding a heavy support is described in figure 3. In each step, the algorithm determines the set of constraints that are not satisfied by the lexicographical bound. It selects such a constraint c, determines a preferred support for c, and uses it to reduce the lexicographical bound. Constraints having more important variables should be processed first as they can result in higher decreases of the bound. When reducing the bound, some variables are changing their value and all constraints containing those variables need to be rechecked. If they are not satisfied by the new bound, they are added to the queue Q. The process is repeated until the current bound satisfies all the constraints or there is a constraint that has no support under the given bound.

Theorem 2 *The algorithm* HEAVYSUPPORT*(B, σ) terminates after at most $O(d^{|Y|})$ iterations. If $B \cup \{Y \leq_{lex} \sigma(Y)\}$ is consistent, then it returns a solution of Lex$(Y)(B \cup \{Y \leq_{lex} \sigma(Y)\})$. Otherwise, it returns an inconsistency.*

It is important to understand the 'backtracking' behaviour of the algorithm. When seeking for a preferred support, the algorithm will first reduce the bound on the last variable. If this fails, it backtracks to the previous variable and will reduce its bound. By doing this, it can hit a 'hole', i.e. a variable that does not belong to the constraint. After reducing the bound on the 'hole', the algorithm can choose any value for the succeeding variables, return a support, and hand over control to a constraint having a variable covering the hole. However, it can also happen that there is no support given the values of the variables preceding the hole. In that case, the algorithm backjumps over the hole and changes a value of a variable that precedes the hole.

4 Conclusion

In [15], David McAllester has predicted that MAC-based solvers will outperform constraint engines based on truth maintenance. For more than one decade, this had been true. The recent success of conflict analysis and conflict learning in SAT solvers has changed the picture again. It became an intensive research topic to extend SAT by specific theories such as linear constraints, uninterpreted functions, and others, and to make conflict learning work for those problems. In this paper, we showed that conflict learning can profit from the structure of general CSPs as well. It reacts to the failure of synthesizing supports and the learned conflicts cut off whole subtrees and not just partial assignments. We thus obtain a smooth adaption of JTMS ideas from [6] to CSPs, while generalizing the form of conflicts and allowing compact representations of Craig interpolants in form of lexicographical constraints. The approach makes conflict learning more pervasive for CP and permits a return of the JTMS to CP. However, the approach may also be interesting for other applications of Craig interpolants [1, 16].

Our work has the same relation to BE as PrefAC [2] has to AC as it focuses effort to preferred values. We get the behaviour of PrefAC for problems with an induced width of 1. As LEXINTERPOLATE keeps results of subproblems that are solved again and again, it promises a better behaviour for proving inconsistencies as required by overconstrained configuration problems [11] and by certain software verification problems (such as the detection of tests that are always false). It also appears that we exchanged the roles of search and inference. Each constraint has a customized support finding algorithm, which is indeed a local backjumping [18] search algorithm. And LEXINTERPOLATE provides a generic algorithm that infers 'fat' conflicts based on the result of the local searches. It is important to note that this is only possible if the different subproblems share the same preference orders. We thus identified another case where preferences play an important role for problem solving [7].

Experimental evaluation for specific software verification problems is in progress. Future work will be devoted to incorporate the best-fail principle from [12] to deal with infinite domains as they occur in software verification problems.

REFERENCES

[1] Eyal Amir and Sheila A. McIlraith, 'Partition-based logical reasoning.', in *KR 2000*, pp. 389–400, (2000).

[2] Christian Bessière, Anaïs Fabre, and Ulrich Junker, 'Propagate the right thing: how preferences can speed-up constraint solving', in *IJCAI-03*, pp. 191–196, Acapulco, (2003).

[3] Gerhard Brewka, Ilkka Niemelä, and Tommi Syrjänen, 'Logic programs with ordered disjunction', *Computational Intelligence*, **20**, 335–357, (2004).

[4] William Craig, 'Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory', *Journal of Symbolic Logic*, **22**, 269–285, (1957).

[5] Rina Dechter, 'Bucket elimination: A unifying framework for reasoning.', *Artifi cial Intelligence*, **113**(1-2), 41–85, (1999).

[6] Jon Doyle, 'A truth maintenance system', *Artifi cial Intelligence*, **12**, 231–272, (1979).

[7] Jon Doyle, 'Prospects for preferences', *Computational Intelligence*, **20**, 11–136, (2004).

[8] Eugene C. Freuder, 'Complexity of k-tree structured constraint satisfaction problems.', in *AAAI-90*, pp. 4–9, (1990).

[9] Matthew L. Ginsberg, 'Dynamic backtracking', *Journal of Artifi cial Intelligence Research*, **1**, 25–46, (1993).

[10] Roberto J. Bayardo Jr. and Robert Schrag, 'Using CSP look-back techniques to solve real-world SAT instances.', in *AAAI-97*, pp. 203–208, (1997).

[11] Ulrich Junker, 'QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems', in *AAAI-04*, pp. 167–172, (2004).

[12] Ulrich Junker, 'Preference-based inconsistency proving: When the failure of the best is sufficient', in *IJCAI-05 Multidisciplinary Workshop on Advances in Preference Handling*, pp. 106–111, (2005).

[13] Ulrich Junker, 'Preference-based problem solving for constraint programming', in *Preferences: Specifi cation, Inference, Applications*, Dagstuhl Seminar Proceedings 04271, (2006).

[14] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault, 'Maintaining arc-consistency within dynamic backtracking', in *CP 2000*, pp. 249–261, (2000).

[15] David A. McAllester, 'Truth maintenance.', in *AAAI-90*, pp. 1109–1116, (1990).

[16] Kenneth L. McMillan, 'Applications of Craig interpolants in model checking.', in *TACAS 2005*, pp. 1–12, (2005).

[17] Charles Petrie, 'Revised dependency-directed backtracking for default reasoning', in *AAAI-87*, pp. 167–172, (1987).

[18] Patrick Prosser, 'Hybrid algorithms for the constraint satisfaction problem', *Computational Intelligence*, **9**, 268–299, (1993).

[19] Daniel Sabin and Eugene C. Freuder, 'Contradicting conventional wisdom in constraint satisfaction.', in *ECAI-94*, pp. 125–129, (1994).

[20] João P. Marques Silva and Karem A. Sakallah, 'GRASP: A search algorithm for propositional satisfiability.', *IEEE Trans. Computers*, **48**(5), 506–521, (1999).