Boosting MUS Extraction

Santiago Macho González and Pedro Meseguer

IIIA, Institut d'Investigació en Intel.ligència Artificial
CSIC, Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Catalonia, Spain.
{smacho|pedro}@iiia.csic.es

Abstract. If a CSP instance has no solution, it contains a smaller unsolvable subproblem that makes unsolvable the whole problem. When solving such instance, instead of just returning the "no solution" message, it is of interest to return an unsolvable subproblem. The detection of such unsolvable subproblems has many applications: failure explanation, error diagnosis, planning, intelligent backtracking, etc. In this paper, we give a method for extracting a Minimal Unsolvable Subproblem (MUS) from a CSP based on a Forward Checking algorithm with Dynamic Variable Ordering (FC-DVO). We propose an approach that improves existing techniques using a two steps algorithm. In the first step, we detect an unsolvable subproblem selecting a set of constraints, while in the second step we refine this unsolvable subproblem until a MUS is obtained. We provide experimental results that show how our approach improves other approaches based on MAC-DVO algorithms.

1 Introduction

Constraint Satisfaction Problems (CSPs) have been applied with great success to tasks dealing with resource allocation, scheduling, planning, configuration and others. When a CSP instance has solution, the solver returns an assignment of values to variables such that all constraints are satisfied. But when a CSP instance is unsolvable, often the solver just returns a "no solution" message. In recent years, conflict-based reasoning is gaining interest in the field of constraint satisfaction. Instead of just certify that a CSP instance is unsolvable, more interesting is explaining why this instance has no solution. These explanations are useful in many settings: interactive applications, error diagnosis, planning, intelligent backtracking, etc. In early years, several authors focused on conflict based reasoning [19, 8, 22]. More recent work [13] focuses on extracting a minimal unsolvable subset (MUS) from the unsolvable problem, where minimal means that all subproblems of the MUS are solvable and all superproblems are unsolvable.

In the SAT field (Boolean satisfiability), many methods for finding MUSes have been developed. Early work [4, 3, 18, 23] was limited to find a single unsatisfiable subformula (US) but without guaranteeing its minimality. These unsatisfiable subformula can be minimized into a MUS using the "Minimal Unsatisfiability Prover" developed in [11]. Very interesting is the work presented in [15] where authors developed a sound and complete technique for finding all MUSes of a CNF formula, based on a strong relationship between maximal satisfiability and minimal unsatisfiability [17]. This relationship also was noted by [1].

The notion of *Maximal Satisfiable Subset*(MSS) as a complement of a MUS is presented in [16]. The authors show that MUSes and MSS are implicit encoding one of the other. They have shown that a the complement of a MSS (CoMSS) is a hitting set of the set of MUSes and contains the minimal set of constraints that should be removed in order to restore consistency.

We have to mention the approach presented in [10] that computes a MUS using a two-step algorithm. Firstly they filter constraints that will not participate in the no-solution condition, obtaining an unsolvable subset of the original CSP. This subset is used in the second step, to identify the constraints that belong to a MUS. In order to have a competitive algorithm, authors use a solver that implements a MAC algorithm with dynamic variable ordering (DVO). Up to our knowledge, this combination achieves the highest efficiency among published approaches for MUS extraction.

The contribution that we present in this paper follows a similar strategy. Given a CSP instance without solution, in a first step we obtain an unsolvable subproblem by performing a forward checking search with dynamic variable ordering (FC-DVO), and computing the hitting set among the subsets of constraints involved in the no-solution condition. This process is iterated while getting unsatisfiable subproblems of lower size, with the help of a heuristic to select variables that are likely to be in a MUS. In a second step, once a MUS candidate has been selected, it is refined until obtaining a true MUS, as the second step of [10]. Experimentally, we obtain a significant improvement in performance with respect to the results of [10].

This paper is organized as follows. In Section 2, we discuss the relation of our approach with abstraction in constraint processing. In Section 3 we introduce the theoretical background needed for the paper. The detailed algorithm appears in Section 4, while the experimental results are in Section 5. There, we compare the performance of our approach against the algorithm described in [10] that is the most efficient implementation we have found to calculate a MUS. Finally in Section 6, we summarize our approach and discuss future work.

2 MUS and Abstraction

In the constraint reasoning literature, a new contribution to CSP solving is usually given at *low level*: typically, a new algorithm, heuristic, or combination of solving methods is presented in every detail, showing how the individual elements of a CSP instance (variables, values, constraints) evolve to find a solution or to show that none exists. On the other hand, contributions that see a CSP instance as a collection of subproblems that interact among them are much more scarce. We call these *high level* descriptions, where the emphasis is not on the atomic components of the instance, but on subproblems, as an intermediate entity between individual elements and the whole instance. High level descriptions provide an alternative view on CSPs, allowing for a kind of reasoning different from the one based on atomic elements. For instance, one may want to find a subproblem having a particular property. This may generate heuristics of variable ordering, original ways of constraint processing or unexpected solving strategies. Although infrequent, this approach is not new in the constraint literature. Without trying to be exhaustive, we mention as representative examples the following works [6] [21] [5].

A simple example of high level description is the analysis of unsolvable CSP instances. If an instance has no solution, it contains at least one minimal unsatisfiable subproblem (MUS). Until this MUS is not solved (modifying it by removing some constraints or enlarging domains), the whole instance will remain without solution. Therefore, once we have seen that the original instance has no solution, the identification and extraction of this MUS by efficient algorithms is a primary goal, if we want to be able to solve the original CSP instance (in fact, an instance closer to the original instance, since this one is unsolvable).

High level descriptions can be seen as abstractions, where the emphasis is put on subproblem properties and particular details of atomic elements are ignored. Abstractions provide new perspectives on the problem, and allow for useful reasoning mechanisms. By no means we are advocating to consider reasoning at high level only, and forgetting the low level description. We stress the usefulness of reasoning at high level, but once it is done, you have to go down the low level and perform the work there. In some sense, reasoning at high level drives the computational activity to be performed at low level.

This paper combines reasoning at both levels. Our goal is to find an efficient way to extract a MUS, once the original instance has been proven unsolvable. Efficiency is crucial here, because after MUS extraction, the new instance has to be solved again. Without an efficient MUS extraction, the whole approach would be practically unfeasible. Reasoning at high level, we have devised an heuristic for selecting candidate variables for an hypothetical MUS. This heuristic is applied at low level, combined with a forward checking algorithm [9]. We obtain a unsolvable subproblem, which is later refined to obtain a true MUS, using an already known approach. Experimentally, we have seen that this heuristic gives quite good results, improving the efficiency of the most performant approach published up to date [10].

3 Theoretical Background

This Section provides the reader with the notions needed to follow the paper.

Definition 1 (CSP). *A CSP is defined by a tuple $\langle X, D, C \rangle$ where,*

- $X = \{x_1, x_2, ..., x_n\}$ *is a set of* n *variables.*
- $D = \{D_1, D_2, ..., D_n\}$ *is a set of* n *domains, where variable* x_k *takes values in* D_k.

– $C = \{c_1, c_2, \ldots, c_r\}$ *is a set of* r *constraints. A constraint* c *involves a sequence of variables* $var(c) = \langle x_p, \ldots, x_q \rangle$ *denominated its scope. The extension of* c *is the relation* $rel(c)$ *defined on* $var(c)$*, formed by the permitted value tuples on the constraint scope.*

A solution of the CSP is an instantiation of values to all variables such that the assigned values belong to the corresponding domains, and this instantiation satisfies all constraints in C. Sometimes, it is not possible to find such instantiation, in that case the problem is unsolvable.

Definition 2 (Subproblem). *Let* $P = \langle X, D, C \rangle$ *be a CSP. A subset of variables* $S \subset X$ *defines the subproblem* $P|_S = \langle S, D|_S, C|_S \rangle$*, where* $D|_S$ *is the subset of domains of variables in* S *and* $C|_S$ *is the subset of constraints with their scopes in* S*. The size of the subproblem is* $|S|$*.*

When a CSP is unsolvable, instead of return the message of "no solution", could be interesting to return the unsolvable subproblem that makes unsolvable the whole problem. If we refine this unsolvable subproblem, identifying the minimal subset of constraints causing that the problem has no solution, we obtain a subproblem useful in many applications: explanation, diagnosis, planning, etc.

Definition 3 (Minimal Unsolvable Subproblem). *Let* $P = \langle X, D, C \rangle$ *be a CSP without solution. A minimal unsolvable subproblem is determined by a subset of variables* $S \subset X$ *such that* $P|_S$ *is unsolvable, but for any proper subset* $S' \subsetneq S$*,* $P|_{S'}$ *is solvable.*

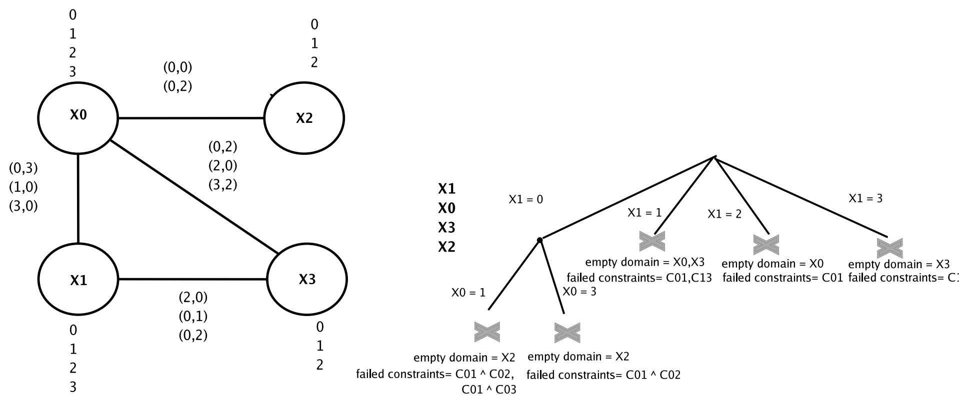
Definition 4 (Hitting Set). *Given a collection of sets* $S = \{S_1, \ldots, S_n\}$*, a hitting set of* S*,* $HST(S)$*, is a set that contains at least one element from each set* $S_1, \ldots, S_n$*, that is,* $\forall S_i \in S, HST(S) \cap S_i \neq \emptyset$*.*

Example 1. Let $S = \{S_1, S_2, S_3\}$ where $S_1 = \{c_{12}\}$ $S_2 = \{c_{03}, c_{23}, c_{13}\}$ $S_3 = \{c_{23}, c_{13}\}$. There are several hitting sets of S, i.e: $HST_1(S) = \{c_{12}, c_{23}\}$ $HST_2(S) = \{c_{12}, c_{13}\}$ $HST_3(S) = \{c_{12}, c_{13}, c_{03}\}$.

The hitting set problem can be prove to be NP-complete by a reduction from the vertex cover problem [7].

Proposition 1. *Let* $P = \langle X, D, C \rangle$ *be a CSP without solution, explored by the forward checking (FC) algorithm. Let* $CONS \subset X$ *be the subset of variables that have been assigned by FC. Let* $EMPTY \subset X$ *be the subset of variables for which an empty domain has been detected. Calling* $S = CONS \cup EMPTY$*, the subproblem* $Q = \langle S, D|_S, C' \rangle$*, where* $C' = \{c \in C | c$ *is responsible for eliminating values of the domain of a variable that either was assigned or became empty at each branch*$\}$ *is unsolvable.*

Proof. To prove this result, it is enough to realize that FC only assigns variables in $CONS$ and only requires variables in $EMPTY$ to detect that there is no solution, using constraints of C'. Therefore, if $S = CONS \cup EMPTY$, FC will find that Q has no solution, by simply repeating the variable instantiation order used in the FC execution on P. $\square$



(a) (b)

Fig. 1. (a) CSP (b) Its search tree generated by the FC algorithm.

Thus, following proposition 1, we can obtain an equivalent unsolvable subproblem of the original unsolvable CSP using a FC Solver. It is important to remark that proposition 1 is true either for static or dynamic variable ordering.

Example 2. Let $MUS = \{x_0, x_1, x_2\}$ be a minimal unsolvable subset of the CSP shown in Figure 1(a) where constraints indicate allowed values between variables. Figure 1(b) shows the corresponding search tree generated by FC algorithm with static order x_1, x_0, x_3, x_2. Here $CONS = \{x_1, x_0\}, EMPTY = \{x_2, x_0, x_3\}$.

Example 3. Figure 2 shows the generated unsolvable subproblem following proposition 1. This subproblem is made by the union of the failed constraints of all branches. In the example, $C = \{c_{01} \wedge c_{02}, c_{01} \wedge c_{03}, c_{01} \wedge c_{02}, c_{01}, c_{13}, c_{01}, c_{13}\}$ that is equivalent to $C = \{c_{01}, c_{02}, c_{03}, c_{13}\}$. In this example, the unsolvable subproblem obtained is equal than the original CSP.

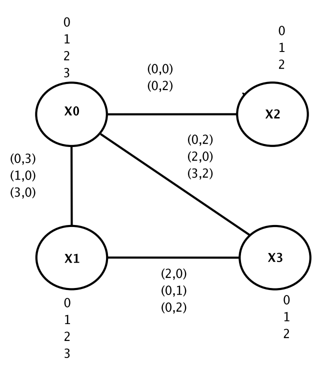


Fig. 2. An unsolvable subproblem of the CSP shown in Figure 1(a).

We notice in example 3 that the selected constraints are more than enough to guarantee the no-solution condition. If we study in more detail the search tree of the Figure 1(b), in every failed branch we have a disjunction of constraints, i.e: the leftmost branch has as failed constraints $\{c_{01} \wedge c_{02}, c_{01} \wedge c_{03}\}$. That means that in this branch, only the constraint $c_{01} \wedge c_{02}$ or the constraint $c_{01} \wedge c_{03}$ is necessary to produce an empty domain in variable x_2. The same analysis is valid for all branches of the search tree. Thus, if we select at least one constraint from each branch, the resulting set will be an unsatisfiable subset of the original problem. This is the definition of *Hitting Set*. The set of constraints of every branch, CC_i, is made by selecting the constraints C' that justify failure in each branch as explained in proposition 1. This generates the following result.

Proposition 2. *Let $\langle X, D, C\rangle$ be a CSP without solution, explored by FC. Let CC be the collection of subsets of constraints that justify failure at each branch. Then, $\langle X, D, HTS(CC)\rangle$ is unsolvable.*

Proof. $CC = \{CC_1, ..., CC_k\}$ is the collection of subsets of constraints justifying failure, one for each branch. The structure of one of these subsets is $CC_i = \{\{c_{i_1}, ..., c_{i_p}\}, ..., \{c_{i_r}, ..., c_{i_t}\}\}$, meaning that each element of CC_i is enough to justify failure in its branch. Since $HTS(CC)$ takes at least one element of each subset CC_i, FC on the subproblem $\langle X, D, HTS(CC)\rangle$ will also fail in every branch. So this subproblem has no solution. $\square$

Example 4. In the CSP of the Figure 1(a), let CC_i represents the set of the selected failed constraints by proposition 1. From the leftmost branch to the rightmost branch of the search tree 1(b), we obtain: $CC_1 = \{c_{01} \wedge c_{02}, c_{01} \wedge c_{03}\}, CC_2 = \{c_{01} \wedge c_{02}\}, CC_3 = \{c_{01}, c_{13}\}, CC_4 = \{c_{01}\}, CC_5 = \{c_{13}\}$. Let $CC = \{CC_1, CC_2, CC_3, CC_4, CC_5\}$. Every *Hitting Set* of CC produces an unsolvable subset. i.e: $HST(CC) = \{c_{01}, c_{02}, c_{13}\}$ as shown in Figure 3.

Thus, we can develop an algorithm to obtain an unsolvable subset, firstly selecting the constraints that justify failure at every branch of the search tree

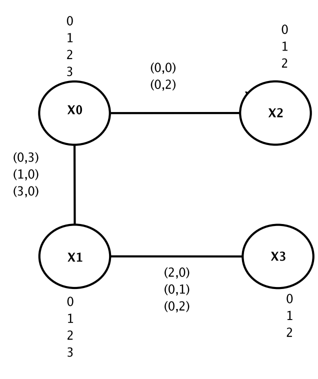


Fig. 3. An unsolvable subproblem of the CSP shown in Figure 1(a).

and afterwards calculating its HST. In addition, if the variables that are likely to be in a MUS are put first in the search tree, the subproblem size tend to be smaller. The following result helps to detect such variables.

Proposition 3. *Let a CSP without solution, explored by FC-SVO. Let $EMPTY \subset X$ be the subset of variables for which an empty domain has been detected during search. $EMPTY$ contains a variable of a MUS.*

Proof. Let us consider the deepest branch instantiated by FC-SVO, formed by the ordered set of variables $\{x_1, ..., x_k\}$. From the FC-SVO search, we know that $\{x_1, ..., x_k\} \cup EMPTY$ is unsolvable. But the subset of variables $\{x_1, ..., x_k\}$ is solvable, because FC-SVO has instantiated it. Therefore, it should exist at least one variable in $EMPTY$, the other subset of variables, that makes the whole subproblem unsolvable. To see that it must belong to a minimal unsolvable subproblem, it is enough to realize that inside any unsolvable subproblem there is a minimal unsolvable one. □

We will use this result, valid for static variable ordering only, as a heuristic when forward checking uses dynamic variable ordering. This heuristic will help trying to detect MUS variables, as we will see in the next Section.

4 The CORE-FC Algorithm

In this Section we present the algorithm **CORE-FC**, that implements our approach. It extracts a MUS from an unsolvable CSP instance, using the theoretical background previously introduced. Having an unsolvable CSP, the basic idea of **CORE-FC** (algorithm 1) is first to generate an unsolvable subset using the function **CORE-FC1** and afterwards refine this subset with function **CORE-FC2** until a minimal subset is found. We describe these algorithms in the following.

4.1 The CORE-FC1 Algorithm

As mentioned before, the goal of **CORE-FC1** (algorithm 2) is to obtain an unsolvable subproblem (not necessary minimal) of an unsolvable CSP instance. This algorithm is based on propositions 2 and 3. We decided to use a solver based on a Forward Checking algorithm with Dynamic Variable Ordering (FC-DVO), where variables are selected according to the popular minimum domain heuristic [9]. The reason for this choice is that using a solver based on a backtracking with

Algorithm 1 *The* **CORE-FC** *algorithm.*

Require: X, D, C is an unsolvable problem
1: **procedure** **CORE-FC**(X, D, C)
2: $X_{us}, D_{us}, C_{us} \leftarrow$ CORE-FC1(X, D, C) $\{X_{us}, D_{us}, C_{us}$ is an unsolvable subset of $X, D, C\}$
3: $X_{mus}, D_{mus}, C_{mus} \leftarrow$ CORE-FC2(X_{us}, D_{us}, C_{us})
4: **return** $X_{mus}, D_{mus}, C_{mus}$
Ensure: $X_{mus}, D_{mus}, C_{mus}$ is a minimal unsolvable subset of X, D, C

DVO is not competitive. A solver based on MAC-DVO is competitive, but we notice that we can select a smaller subset of constraints involved in a MUS by using a solver based on FC-DVO. Experiments show that FC-DVO tends to find smaller MUS candidates than MAC-DVO. We explain this fact as follows. MAC performs arc-consistency on every constraint. When MAC realizes that the instance has no solution, it has to include each constraint that is responsible for removing a value into the MUS candidate (this includes constraints among variables that never have been instantiated). On the other hand, FC propagation is simpler: it performs arc consistency on constraints connecting assigned and unassigned variables at any stage of search. In particular, constraints among variables that have never been instantiated are not considered. For this reason, we believe that FC focuses better than MAC on suitable MUS candidates. This intuition is confirmed in practice (see Section 5).

CORE-FC1 works as follows. It takes as input an unsolvable CSP instance. It returns as output an unsolvable subproblem of that instance. Firstly, it pass the instance to a modified FC-DVO solver. This solver takes as input a CSP instance and a variable (line 5). This variable is forced to be the first one instantiated by the solver although the DVO criteria does not select it first. The reason for this will become apparent later. As output, it returns a variable with empty domain in the deepest branch explored by FC-DVO.

In CC we collect the subsets of constraints that justify failure at each branch of FC-DVO (line 6). By proposition 2, we know that the subproblem generated by the hitting set of CC, $HTS(CC)$, has no solution. Therefore, we replace the input instance by this subproblem. Algorithm 3 computes the Hitting Set of a set of constraints. This function is used by the algorithm 2 in order to obtain an unsolvable subproblem (line 7).

To decrement the size of the unsolvable subproblem, one can repeat the above process with a different variable ordering. If variables that belong to a MUS are instantiated at the first levels of the FC search tree, the resulting unsolvable subproblem tend to be smaller. Following this idea, we consider that a variable with empty domain in the deepest branch explored by FC-DVO in the current iteration it is likely to be in a MUS. This is based on proposition 3, which expresses a result valid for SVO, while we use it here as heuristic for DVO.

Algorithm 2 *The* **CORE-FC1** *algorithm.*

Require: X, D, C is an unsolvable problem
1: **function** **CORE-FC1**(X, D, C)
2: $us_{cur} \leftarrow C$
3: **repeat**
4: $\quad us_{prev} \leftarrow us_{cur}$
5: $\quad x_k \leftarrow$ solveCSP-FC-DVO(X, D, C, x_1) { x_k is candidate to belong to the MUS}
6: $\quad CC \leftarrow$ collection of subsets of constraints that justify failure at each branch of FC-DVO
7: $\quad us_{cur} \leftarrow HST(CC)$
8: $\quad$ reorder variables so that x_k becomes x_1
9: **until** $|us_{prev}| \leq |us_{cur}|$
10: $< X_{us}, D_{us}, C_{us} > \leftarrow generateCSP(us_{prev})$
11: **return** $< X_{us}, D_{us}, C_{us} >$

Ensure: $< X_{us}, D_{us}, C_{us} >$ is an unsolvable subset of $< X, D, C >$

Algorithm 3 *The* **Hitting Set** *algorithm.*

procedure **HST**(CC)
$hittingSet \leftarrow \emptyset$
$CC \leftarrow initialPurge(CC)$
while $\neg isHittingSet(hittingSet)$ **do**
    $c \leftarrow chooseConstraint(CC)$
    $hittingSet \leftarrow hittingSet \cup \{c\}$
    $CC \leftarrow purge(CC)$
end while
return $hittingSet$

Then, we take that variable to be instantiated first at the next iteration (line 8). This variable is returned by the FC-DVO solver. The whole process iterates until the unsolvable subproblem does no longer decrement its size (line 9).

We are interested in a hitting set algorithm that minimizes the number of variables that are added to the HST when a new constraint is included. Unfortunately [14] shows that calculating the minimal hitting set is an NP-hard problem. We implement a strategy to minimize the number of variables that are included in the hitting set using the function *chooseConstraint*. This function selects the constraint that minimizes the number of variables that we include in the HST. Note that this heuristic does not guarantee that the resulting hitting set has a minimum number of variables, but empirically it works well and provides good results. The *purge* function remove sets that are superset of others.

4.2 The CORE-FC2 Algorithm

CORE-FC2 (algorithm 4) refines the unsolvable instance that takes as input, and returns a MUS of that instance. It is based on the *dcMUC* function shown in [10]. The algorithm enters in a loop (lines 4 - 12) where new variables that belongs to the MUS are discovered using the *DichotomicSearch* function (line 5). When a new variable x_k is discovered, it is included in the MUS candidate (line 6). If this candidate has no solution (line 9), then it is confirmed as MUS (line 10) and the loop ends. Otherwise, the original instance is searched again for more variables in the MUS, looking into the subproblems that at least contain $\{x_0, \ldots, x_{k+1}\}$ (line 12).

In order to identify a variable that belongs to a MUS, the next procedure can be used. Starting from the first variable in the given order, add at every step one more variable until the CSP becomes unsatisfiable. When that occurs the last added variable belongs to a MUS. If we want to speed up this algorithm, we can use a dichotomic search. The *DichotomicSearch* function (algorithm 5) starts a dichotomic search in order to find a variable that belongs to the MUS (similar to function *dcTransition* described in [10]). It searches this variable in the set $\{x_{min}, \ldots, x_{max}\}$. Initially, index min takes value k, while max takes the total number of variables. Parameter k is the limit between the discovered variables of the MUS and the undiscovered ones. The algorithm enters in a loop between lines 3-12 until a variable of the MUS is discovered. It takes the set $\{x_0, \ldots, x_{center}\}$ and checks if it is solvable or not. If it is solvable, it searches

Algorithm 4 *The* **CORE-FC2** *algorithm.*

Require: X, D, C is a superset of a MUS
function **CORE-FC2**(X,D,C)
$X_{MUS} \leftarrow \emptyset, D_{MUS} \leftarrow \emptyset, C_{MUS} \leftarrow \emptyset$
$MUS \leftarrow$ **false**, $k \leftarrow 0$
while $\neg MUS$ **do**
  $x_k \leftarrow DichotomicSearch(X, D, C, k)$
  $X_{MUS} \leftarrow X_{MUS} \cup \{x_k\}$ {we include this var to the MUS}
  $C_{MUS} \leftarrow C_{MUS} \cup \{c_{ik}\}$ {c_{ik} is a set of constraints involving variable x_k}
  $D_{MUS} \leftarrow D_{MUS} \cup \{d_k\}$
  if solveCSP-MAC-DVO($X_{MUS}, D_{MUS}, C_{MUS}$) $= UNSAT$ **then**
    $MUS \leftarrow$ **true**
  end if
  $k \leftarrow k + 1$
end while
return $X_{MUS}, D_{MUS}, C_{MUS}$ {minimal unsolvable subproblem}
Ensure: $X_{MUS}, D_{MUS}, C_{MUS}$ is a MUS

Algorithm 5 *The* **Dichotomic Search** *algorithm.*

1: **procedure** **DichotomicSearch**(X,D,C,k)
2: $min \leftarrow k, max \leftarrow |X|$
3: **while** $min \neq max$ **do**
4:   $center \leftarrow (min + max)/2$
5:   $X_{DIC} \leftarrow \{x_0, \ldots, x_{center}\}$
6:   $C_{DIC} \leftarrow$ set of constraints involving vars $\{x_0, \ldots, x_{center}\}$
7:   $D_{DIC} \leftarrow \{d_0, \ldots, d_{center}\}$
8:   **if** solveCSP-MAC-DVO($X_{DIC}, D_{DIC}, C_{DIC}$) $= SAT$ **then**
9:     $min \leftarrow center + 1$
10:   **else**
11:     $max \leftarrow center$
12:   **end if**
13: **end while**
14: **return** x_{min}

for the variable of the MUS among the set $\{x_{center+1}, \ldots, x_{max}\}$. If the problem is unsolvable then the variable belongs to the set $\{x_{min}, \ldots, x_{center}\}$. Doing this procedure, the variable that belongs to the MUS is obtained when $x_{max} = x_{min}$ (line 3).

5 Experimental Results

In this section we have performed several experiments in order to compare the performance of our approach against the **PCORE+WCORE** algorithm described in [10], that at the present seems to be the most performant published algorithm. We use non competitive FC and MAC solvers based on the JCL library [12, 20].

The **PCORE+WCORE** described in [10] works as follows. This is a 2-step algorithm, the PCORE step and the WCORE step. In the PCORE step, a MAC-DVO solver is used to return an unsolvable subproblem made by all the constraints that during the search removed at least one value in the domain of a variable. A dom/wdeg heuristic is used to choose the order in which variables will be instantiated. Calls to the MAC-DVO solver are done (updating the heuristic at every call) until the size of the obtained subproblem does not

BENCHMARK			PCORE + WCORE					
Name	VARS	CONS	SIZE US PCORE	CHECKS PCORE	SIZE MUS WCORE	TIME	CHECKS	VIS NODES
randomB-25-58	25	58	19	4181	8	9.3s	60731	569
randomB-16-90	16	90	16	2648	7	4.2s	25072	364
randomB-26-6	26	63	17	7591	6	3.7s	17534	228
randomB-31-347	31	347	20	2550	8	9.8s	55033	582
randomB-43-176	43	176	35	34859	15	50.7s	381684	2012
randomB-36-470	36	470	21	1116	7	5.8s	10719	314
randomB-48-220	48	220	29	39116	16	47.1s	410452	1992
randomB-45-739	45	739	19	1818	7	8.3s	13725	426
pigeons5	5	10	5	1400	5	**0.23s**	3903	250
pigeons6	6	15	6	8250	6	**0.76s**	19683	883
pigeons7	7	21	7	52092	7	**3.73s**	102378	4219
pigeons8	8	28	8	369446	8	**25.22s**	618219	26825
pigeons9	9	36	9	2963760	9	**219.54s**	4597144	209178
pigeons10	10	55	10	26686962	10	**2458s**	40353999	1872587
pigeons11	11	55	11	266889620	11	**26324s**	400961870	18708727
dual-ehi-85-297-0	297	4094	60	120167	25	742.98s	1489447	10001
dual-ehi-85-297-1	297	4112	83	143760	19	738.69s	988227	7136
dual-ehi-85-297-7	297	4111	82	152272	18	627.45s	1167588	6030
dual-ehi-85-297-9	297	4118	64	55215	20	599.06s	911398	6322
dual-ehi-85-297-18	297	4120	69	144520	18	683.49s	927860	6577
dual-ehi-85-297-20	297	4106	72	85655	20	576.08s	1033272	6166
dual-ehi-85-297-24	297	4105	70	89563	19	694.70s	1037183	6703
dual-ehi-85-297-26	297	4102	19	115150	19	282.94s	1032841	6876
dual-ehi-85-297-27	297	4120	57	80516	20	575.42s	939922	6399
dual-ehi-85-297-44	297	4130	70	96148	16	474.16s	639734	4677
dual-ehi-85-297-49	297	4124	61	118023	22	495.73s	1255883	7603
dual-ehi-85-297-65	297	4116	74	147270	21	539.22s	975204	7997
dual-ehi-85-297-83	297	4099	90	169078	20	802.28s	1789755	8064
dual-ehi-85-297-88	297	4119	69	114815	18	603.50s	758366	6306
dual-ehi-85-297-92	297	4106	65	142880	20	595.57s	947950	7133
dual-ehi-85-297-99	297	4115	117	124209	19	710.32s	1319022	7371

Table 1. Results for the PCORE+WCORE algorithm.

longer decrease. Once the PCORE step returns an unsolvable subproblem (not minimal), the WCORE step extract a MUS from this unsolvable subproblem, using a dichotomic search.

We ran three different set of benchmarks. Firstly, we have generated unsolvable subproblems using a modified *random model B generator*, where we forced the random generator to return unsolvable CSPs. The second set of constraints is the well known problem of the *pigeons*, where we have to put n pigeons into n-1 boxes, one pigeon per box. These pigeons problems are interesting, because the whole problem is a MUS. Finally we ran experiments on the *dual-ehi* benchmarks that are 3-SAT instances converted to binary CSP instances using the dual method. The pigeons and the dual-ehi benchmarks can be found in [2].

Table 1 shows the performance of the algorithm proposed in [10], while table 2 shows the performance of our approach described in the previous section. The columns indicates: the name of the benchmark, the number of variables and constraints the benchmark has, the size of the US (not minimal) and the number of checks after the first step, the size of the obtained MUS, the execution time in seconds, the total number of checks and the number of visited nodes.

We study separately the results for the three different types of benchmarks.

5.1 Random Benchmarks

We generated several unsolvable random problems using the model B. We have modified our generator in order to force it to return unsolvable instances. The generated problems have between 15 to 48 variables and from 58 to 739 constraints. It is important to point that we cannot control if the generated problems have more than one unsolvable subproblem, thus, it is possible that the algorithms find different MUSes. Comparing the benchmarks where both algorithms returns the same MUS, *randomB-36-470* and *randomB-45-739*, we notice that our first step returns smaller unsolvable subproblems than the first step of the PCORE+WCORE algorithm. The second step are equivalent for both algorithms, but our approach has the advantage that the unsolvable subproblem that is the input of the second step is smaller than the unsolvable subproblem of the PCORE+WCORE algorithm. Experiments show that our approach reduces the execution time, the number of checks and the number of visited nodes.

5.2 Pigeons Benchmarks

The pigeons benchmarks are problems where we have to put n pigeons into n-1 boxes, one pigeon per box. These problems have the characteristic that any proper subproblem is solvable (the whole problem is a MUS). Tables 1 and 2 show that with these benchmarks our algorithm has a worse performance in time than the PCORE+WCORE approach. We notice that the first step makes the difference; while our algorithm uses a FC-DVO solver, the PCORE+WCORE approach uses a MAC-DVO solver. In both approaches all constraints will be selected (the whole problem is unsolvable). The MAC-DVO solver is faster than the FC-DVO solver, thus there is an important gain in time at the end of the first step for the PCORE+WCORE algorithm over our approach. Our approach has a better number of checks because a FC solver does less checks that a MAC solver. In the opposite, the number of visited nodes is greater for the FC than the MAC due to the MAC propagation.

5.3 Dual-ehi Benchmarks

The dual-ehi benchmarks are problems where benchmarks that are 3-SAT instances are converted to binary CSP instances using the dual method. We ran several experiments with 297 variables and between 4094 to 4130 constraints. Tables 1 and 2 show that our algorithm has a better performance. In these benchmarks, we decrease the execution time by a factor of 3, also decreasing the number of checks and the number of visited nodes. It is interesting to point that very often the unsatisfiable subset obtained at the first step is a MUS. Therefore our second step is faster than the PCORE+WCORE approach, where the output of the first step is a bigger unsolvable subproblem.

BENCHMARK			CORE-FC1 + CORE-FC2					
Name	VARS	CONS	SIZE US CORE-FC1	CHECKS CORE-FC1	SIZE MUS CORE-FC2	TIME	CHECKS	VIS NODES
randomB-25-58	25	58	11	4452	7	**1.32s**	10931	391
randomB-16-90	16	90	8	2602	8	**1.05s**	12355	359
randomB-26-63	26	63	9	1176	8	**1.19s**	15498	271
randomB-31-347	31	347	8	5803	7	**1.53s**	9056	248
randomB-43-176	43	176	12	7705	10	**4.40s**	41842	974
randomB-36-470	36	470	7	2592	7	**1.54s**	3987	112
randomB-48-220	48	220	6	2954	5	**1.78s**	6110	230
randomB-45-739	45	739	7	3645	7	**2.58s**	5418	138
pigeons5	5	10	5	584	5	0.24s	3041	298
pigeons6	6	15	6	3170	6	0.79s	14489	1123
pigeons7	7	21	7	19452	7	4.45s	69506	5659
pigeons8	8	28	8	136850	8	36.75s	385201	36905
pigeons9	9	36	9	1095824	9	381.78s	2728478	289818
pigeons10	10	55	10	9863874	10	4161.96s	23529833	2598347
pigeons11	11	55	11	98640740	11	47335.87s	232711461	25966327
dual-ehi-85-297-0	297	4094	25	42881	25	**252.39s**	555210	7461
dual-ehi-85-297-1	297	4112	19	31790	19	**227.96s**	224513	4164
dual-ehi-85-297-7	297	4111	18	22259	18	**236.66s**	163585	3061
dual-ehi-85-297-9	297	4118	21	25756	20	**187.65s**	186456	3985
dual-ehi-85-297-18	297	4120	18	20782	18	**187.22s**	207200	3165
dual-ehi-85-297-20	297	4106	20	22432	20	**187.95s**	218315	3969
dual-ehi-85-297-24	297	4105	19	50346	19	**248.93s**	186793	5327
dual-ehi-85-297-26	297	4102	19	32803	19	**241.15s**	180389	4005
dual-ehi-85-297-27	297	4120	20	44817	20	**198.32s**	202829	5585
dual-ehi-85-297-44	297	4130	16	20296	16	**232.46s**	119877	1930
dual-ehi-85-297-49	297	4124	22	20971	22	**248.95s**	312244	4909
dual-ehi-85-297-65	297	4116	21	22402	21	**245.80s**	304262	4291
dual-ehi-85-297-83	297	4099	20	15077	20	**189.53s**	310092	3778
dual-ehi-85-297-88	297	4119	18	25994	18	**186.60s**	174942	3390
dual-ehi-85-297-92	297	4106	20	29802	20	**189.48s**	217860	4103
dual-ehi-85-297-99	297	4115	19	11820	19	**185.97s**	242962	3178

Table 2. Results for the CORE-FC1+CORE-FC2 algorithm.

6 Conclusions

We have developed a new approach for extracting a MUS from an unsolvable CSP instance. It is based on a two-step algorithm. In the first step, an unsolvable subproblem is obtained, using a FC-DVO solver combined with the computation of a hitting set on the constraints responsible for the no solution condition. To remove variables which do not belong to the minimal version of this subproblem, this process is iterated with the help of a heuristic to identify variables that are likely to be in a MUS. The iteration ends when the computed unsolvable subproblem does no longer decrement its size. The second step refines this unsolvable subproblem using a dichotomic search until a true MUS is found.

We compared our approach with the best approach we have found so far called PCORE + WCORE [10] which is also a two-step algorithm. The main difference between these two approaches occurs in the first step. While PCORE + WCORE iterates using a MAC-DVO solver, our approach iterates using a FC-DVO solver combined with the hitting set computation and the heuristic to select likely MUS variables. As result, our approach is able to compute MUS candidates of smaller size (first step of **CORE-FC**), with less computational effort. As consequence, the effort required in the second step is also smaller. Experimental results show that our approach is beneficial in most benchmarks, although in some benchmarks (pigeons) our approach is not competitive. It is worth realizing that each pigeon instance is itself a MUS, so we hypothesize that when the original unsolvable instance is already minimal, our approach is not competitive (in that case, the hitting set computation and the heuristic do not bring any benefit, they add overhead only). But we believe that this is not the general case. Usually, unsolvable instances contain smaller MUSes, for which we believe that our approach is adequate. Our intuition behind the claim that a FC algorithm is better that a MAC algorithm for finding MUSes is that whilst a FC just considers constraints between past and future constraints a MAC algorithm tends to maintain the consistency of the CSP. Thus a MAC algorithm will select more candidate constraints than a FC algorithm. This is explain why our algorithm finds a smaller and better unsolvable candidate.

The capacity of reasoning at subproblem level has been crucial to develop this approach. The hitting set idea considers the different subsets of constraints that are responsible for the no solution condition of the whole subproblem. The heuristic for variables likely to be in a MUS is inspired in a property of the complete subproblem. This view abstracts atomic CSP components, focusing on subproblems. We believe that this perspective offers new and interesting ways of reasoning in constraint solving, able to improve existing techniques or to develop new ones.

Acknowledgements

Authors thank anonymous reviewers for their constructive criticisms.

References

1. J. Bailey and P. J. Stuckey: Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization, 2005. pp174-186. In Proc. PADL05, volume 3350
2. Benchmark problems. http://cpai.ucc.ie/05/Benchmarks.html
3. R. Bruni: Approximating minimal unsatisfiable subformulae by means of adaptive core search. Discrete App.. Math. Journal Vol. 130,(2), 85 – 100, 2003. Elsevier Science Publishers B. V.
4. R. Bruni and A. Sassano: Restoring Satisfiability or Maintaining Unsatisfiability by Finding Small Unsatisfiable Subformulae. (LICS) 2001 Workshop on Theory and Applications of Satisfiability Testing (SAT 2001) Boston (Massachusetts, USA), June 14-15, 2001, Proceedings. Elsevier Science Pub (2001).
5. B. Faltings and S. Macho-Gonzalez: Open Constraint Programming. Artificial Intelligence, vol 161, pp 181–208, 2005.
6. E. Freuder and P. Hubbe: Extracting Constraint Satisfaction Subproblems. In Proc. of the 14th International Joint Conference on Artificial Intelligence, 1995, pp 548-555.

7. M. R. Garey and D. S. Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness (1979). Publisher W. H. Freeman & Co.
8. M.L. Ginsberg: Dynamic backtracking. Journal of Artificial Intelligence Research, 1:25-46, 1993.
9. R. Haralick and G. Elliot: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence, vol 14, pp 263–313, 1980.
10. F. Hemery, C. Lecoutre, L. Sais and F. Boussemart: Extracting MUCs from constraint networks. In Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), 2006.
11. J. Huang: MUP: a minimal unsatisfiability prover. ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation, 2005, 432–437. Shanghai, China. ACM Press.
12. Java Constraint Library (JCL). http://liawww.epfl.ch/JCL/
13. U. Junker: QUICKXPLAIN: Conflict Detection for Arbitrary Constraint Propagation Algorithms. IJCAI'01 Workshop on Modelling and Solving problems with constraints (CONS-1), 2001.
14. R. M. Kar: Reducibility among combinatorial problems. Complexity of Computer Computations, 1972. pp 85-103
15. M. H. Liffiton, Z. S. Andraus, and Karem A. Sakallah: From Max-SAT to Min-UNSAT: Insights and Applications. Technical Report CSE-TR-506-05, February 2005.
16. M. H. Liffiton, M. D. Moffitt, M. E. Pollack, and K. A. Sakallah: Identifying Conflicts in Overconstrained Temporal Problems, in Proc. IJCAI-05, pp. 205–211, Edinburgh, Scotland, 2005.
17. M. H. Liffiton and K. A. Sakallah: On Finding All Minimally Unsatisfiable Subformulas. in Proc. 8th International Conference on Theory and Applications of Satisfiability Testing (SAT-2005), pp. 173-186, June 2005.
18. Y. Oh, M. N. Mneimneh and Zaher S. Andraus and Karem A. Sakallah and Igor L. Markov: AMUSE: a minimally-unsatisfiable subformula extractor. DAC '04: Proceedings of the 41st annual conference on Design automation (2004),518–523,ACM Press.
19. P. Prosser: Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence, 9: 268-299, 1993.
20. M. Torrens, R. Weigel and B. Faltings: Java constraint library: Bringing constraints technology on the Internet using the java language. (1997). In Constraints and Agents: Papers from the 1997 AAAI Workshop, 21–25. Menlo Park, California.
21. G. Verfaillie, M. Lemaitre and T. Schiex: Russian Doll Search. In Proc. of the 13th National Conference on Artificial Intelligence, 1996, pp 181–187.
22. M. Yokoo: Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. CP'95: 88-102, 1995
23. L. Zhang and S. Malik: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03), 2003.