SAT and Hybrid models of the Car-Sequencing problem

Christian Artigues, Emmanuel Hébrard, Valentin Mayer-Eichberger,
Mohamed Siala, Toby Walsh

SAT and Hybrid models of the Car-Sequencing problem

Christian Artigues[1,2], Emmanuel Hebrard[1,2], Valentin Mayer-Eichberger[3,4], Mohamed Siala[1,5], and Toby Walsh[3,4]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[2] Univ de Toulouse, LAAS, F-31400 Toulouse, France
[3] NICTA
[4] University of New South Wales
[5] Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
{artigues, hebrard, siala}@laas.fr
{valentin.mayer-eichberger, toby.walsh}@nicta.com.au

Abstract. We improve the state of the art for solving car-sequencing problems by combining together the strengths of SAT and CP. We compare both pure SAT and hybrid CP/SAT models. Three features of these models are crucial to success. For quickly finding solutions, advanced CP heuristics are important and good propagation (either by a specialized propagator or by a sophisticated SAT encoding that simulates one) is necessary. For proving infeasibility, clause learning in the SAT solver is critical.
Our models contain a number of other novelties. In our hybrid models, we develop a novel linear time mechanism for explaining failure and pruning for the AtMostSeqCard constraint. In our SAT models, we describe powerful encodings for the same constraint. Our study demonstrates some of the potential and complementarity of SAT and hybrid methods for solving complex constraint models.

1 Introduction

In the car-sequencing problem [24], a set of vehicles has to be sequenced in an assembly line. Each class of cars requires a set of options. However, the working station handling a given option can only mount it on a fraction of the cars passing on the line. Each option j is thus associated with a fractional number u_j/q_j standing for its capacity (at most u_j cars with option j occur in any sub-sequence of length q_j). Several global constraints have been proposed in the Constraint Programming (CP) literature to model this family of constraints (i.e. capacity constraints). Most recently, The AtMostSeqCard constraint [21] or its combination with the *Global Sequencing Constraint (*Gsc*)* [19] showed outstanding results compared to other CP models. However, based on experiments, pure CP approaches suffer when the task becomes proving unsatisfiability. The motivations over this paper comes then by exploiting Boolean-Satisfiability (SAT) as it showed outstanding results in many applications.

We therefore propose several approaches combining ideas from SAT and CP for solving the car-sequencing problem. First, we try to capture CP propagation into SAT by a careful formulation of the problem into conjunctive normal form (CNF). We propose a family of pure SAT encodings for this problem and relate them to existing encoding

techniques. To the best of our knowledge, these are the first non-trival CNF encodings for the car-sequencing problem. They are based on extension of Sinz's encoding for the CARDINALITY constraint [22] and have similarities to the decomposition of the GEN-SEQUENCE constraint given in [2]. Second, we introduce a linear time procedure for computing compact explanations for the ATMOSTSEQCARD constraint. This algorithm can be used in a hybrid CP/SAT approach such as SAT Modulo Theory, CDCL Pseudo-Boolean, or lazy clauses generation solvers, where constraints that are not in clausal form need a dedicated propagator and explanation algorithm. In principle, the hybrid approach has access to all the advances from the SAT world, whilst benefiting from constraint propagation and dedicated branching heuristics from CP. However, our experiments reveal that in practice the tradeoff is more complex. Indeed, CDCL algorithms are ever evolving. Therefore, without the best data structures and the most up to date tuning of literal activity maintenance, clause deletion, nogood reduction, (amongst other parameters) the hybrid approach is significantly outperformed on hard unsatisfiable instances.

Our evaluation also provides good empirical evidence for the three following observations: First, CP heuristics can be very useful to quickly find solutions. This was expected, and in fact we were surprised about how robust the generic activity based heuristic is. However, CP heuristics dedicated to the car-sequencing problem are much faster. Second, propagation, either through finite domain propagators, or through unit propagation via a "stronger" encoding, is extremely important to reliably find solutions on the harder instances. Indeed a stronger propagation makes it less likely to enter an unsatisfiable subproblem during search. In conjunction with this, restarting ensures that these unlikely cases do not matter. Third, clause learning is clearly critical for proving unsatisfiability. In that respect, the approaches that we introduce (especially the SAT encodings) greatly improve the state of the art for the car-sequencing problem. Moreover, counter-intuitively, it does not seem that constraint propagation of the ATMOSTSEQCARD constraint nor the "strength" of the SAT encoding, has a significant impact on the ability of the solver to prove unsatisfiability.

The remainder of this paper is organized as follows. We give in Section 2 a short background on CP, SAT and their hybridization. In Section 3, we recall state-of-art CP models for this problem and show the connection with SAT. In Section 4, we show that, based on the ATMOSTSEQCARD propagator, one can build a linear time explanation for this constraint. Then, we present advanced SAT-encodings for this constraint in Section 5. Finally, we empirically evaluate, in Section 6, the approaches we introduce against pure CP and a pseudo Boolean model.

2 Background

2.1 Constraint Programming

A constraint network is defined by a triplet $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ is a mapping of variables to finite sets of values and $\mathcal{C}$ is a set of constraints that specify allowed combinations of values for subsets of variables. We assume that $\mathcal{D}(x) \subset \mathbb{Z}$ for all $x \in \mathcal{X}$. We denote $x \leftarrow v$ the assignment of the value v to the variable x, that is the restriction of its domain $\mathcal{D}(x)$ to $\{v\}$, similarly, we denote $x \nleftarrow v$

the pruning of the value v from $\mathcal{D}(x)$. A *partial instantiation* S is a set of assignments and/or pruning such that no variable is assigned more than one value and no value is pruned and assigned for the same variable. Let $\bot$ be a *failure* or a domain wipe-out, by convention equal to the set of all possible assignments and prunings. On finite domains one should consider a *closure* of partial instantiations with respect to domains. That is, if the assignment $x \leftarrow v$ belongs to S, we also assume that $x \nleftarrow v$ for all $v \in \mathcal{D}(x) \setminus v$ belong to S. Similarly, if all but one of the values are pruned, the remaining value is added as an assignment. This is similar to *expanded* solutions in [14]. However, we shall restrict ourselves to Boolean domains in this paper. We therefore have $S \subseteq S'$ iff S' is a stronger (tighter) partial instantiation than S.

A constraint C defines a relation $Rel(C)$, that is, a set of instantations, over the variables in $Scope(C)$. It is *generalized arc consistent* (GAC) iff, for every value v of every variable x in $Scope(C)$, there exists a consistent instantiation S in $Rel(C)$ such that $x \leftarrow v \in S$. Conversely, we say that a constraint is *dis-entailed* with respect to a partial instantiation S iff there is no t in $Rel(C)$ such that $S \subseteq t$.

Throughout the paper we shall associate a constraint C to a *propagator*, that is, a function mapping partial instantiations to partial instantiations or to the failure $\bot$. Given a partial instantiation S we denote $C(S)$ the partial instantiation (or failure) obtained by applying the propagator associated to C on S, and we have $S \subseteq C(S)$. We say that the partial instantiation S implies the assignment or pruning p with respect to the constraint C iff $p \notin S \ \& \ p \in C(S)$. Given an initial domain $\mathcal{D}$ and a partial instantiation S, we can derive a current domain taking into account the pruning and assignments of S in $\mathcal{D}$. There will not be ambiguities about the original domains, therefore we simply denote $S(x)$ the domain $\mathcal{D}(x)$ updated by the assignment or pruning associated to x in S. Moreover, we shall denote $\min(S(x))$ (resp. $\max(S(x))$) the minimum (resp. maximum) value in $S(x)$.

Finally, the *level* of an assignment or a pruning p is the order of appearance of the assignment (respectively pruning) in the tree search, and we denote it $lvl(p)$.

2.2 SAT-Solving

The Boolean Satisfiability problem (SAT) is a particular case of CSP where domains are Boolean and constraints are only clauses (disjunction of literals). A SAT solver is thus a program that computes a satisfying instantiation of a formula of propositional logic in conjunctive normal form (CNF) or proves that no such instantiation exists. The most widely used method to solve SAT problems is based on the DPLL algorithm ([8]), which is a depth first search with backtracking using a special propagator for clauses. Unit propagation (UP) prunes the assignment of the remaining literal in a clause when all other literals have become false. An important improvement to the DPLL algorithms goes under the name of Conflict-Driven Clause Learning (CDCL). These solvers record for each conflict an appropriate reason in form of a clause, add it to the clause database and can then potentially prune unseen parts of the search tree. Furthermore, SAT solvers are equipped with robust domain-independent branching and decision heuristics (for instance VSIDS [15]). For a comprehensive introduction to SAT solving in general and its techniques we refer to [4].

Modelling in CNF is a crucial step for the success of solving problems with SAT. A natural approach to find a good SAT model is to describe the problem with higher level constraints and then translate these constraints to CNF. In accordance with this methodology the representation of integer domains and encodings of a variety of global constraints have been proposed and analyzed [2, 11, 26]. Similarly the notion of GAC adapts to SAT. Unit propagation is said to maintain GAC on the CNF encoding of a constraint if it forces all assignments to the variables representing domain values that must be set to avoid unsatisfiability of the constraint. A quality measure for an encoding in SAT is a good compromise between its size and its level of consistency by UP. Moreover, it has to be taken into account that SAT solvers cannot distinguish between variables representing domains and auxiliary variables to compactify the translation. So when aiming for a good CNF encoding one has to ensure to relate auxiliary variables to facilitate better propagation.

2.3 Hybrid SAT/CP

The notion of nogood learning in constraint programming is not new, in fact it predates [20] similar concepts in SAT. However, CDCL learns and uses nogoods in a particular way, and recently CDCL based methods have been reintroduced into CP. For instance Katsirelos's generalized nogoods [13] [14] enable this type of algorithms for arbitrary domains. Moreover, propagation does not need to be restricted to unit propagation. As in standard CP, a given constraint may be associated to a specific propagator. However, to simulate the behavior of CDCL, it is necessary to *explain* either a failure or the pruning of a domain value. Lazy-clause generation [16] solvers add a clause whenever pruning is performed in order to provide an explanation for the pruning.

In this paper we use a solver with a slightly different architecture, where constraints are associated with a propagator *and* an explanation algorithm. However, as opposed to explanation based constraint programming [6, 7], the explanations are used exactly as in CDCL, i.e., literals are replaced by their explanation until the current nogood contains a Unique Implication Point of the current level's pruning. In this sense it is very close to the way some Pseudo-Boolean CDCL solvers, such as PBS [1], PBChaff [9] or SAT4JPseudo [3] integrate unit propagations on clauses, dedicated propagators and explanations (cutting planes) for linear equations.

We say that a partial instantiation S is an *explanation* of the pruning $x \nleftarrow v$ with respect to a constraint C if it implies $x \nleftarrow v$ (that is, $x \nleftarrow v \in C(S) \setminus S$). Moreover, S is a valid explanation iff $lvl(x \nleftarrow v) > max(\{lvl(p) \mid p \in S\})$. For instance, if C is the clause $p \vee \neg q \vee r$, the only possible explanation for $p \nleftarrow 0$ with respect to C is $\{q \leftarrow 1, r \leftarrow 0\}$. Consider now the CARDINALITY constraint $\sum_{i=1}^{n} x_i = t$ such that for all i, $D(x_i) = \{0, 1\}$. Here is the characterisation of a propagator for this constraint:

$$\textsc{Cardinality}(S) = \begin{cases} \bot & \text{if } |\{x_j | S(x_j) = \{1\}\}| > t \\ \bot & \text{if } |\{x_j | S(x_j) = \{0\}\}| > n - t \\ S \cup \{x_i \leftarrow 0 \mid S(x_i) = \{0,1\}\} & \text{if } |\{x_j | S(x_j) = \{1\}\}| = t \\ S \cup \{x_i \leftarrow 1 \mid S(x_i) = \{0,1\}\} & \text{if } |\{x_j | S(x_j) = \{0\}\}| = n - t \\ S & \text{otherwise} \end{cases} \quad (2.1)$$

Now, one can explain this constraint as follows:

- Explaining failure: if the failure was triggered by the first case, we already exceed the required demand, hence the explanation would be $\{x_j \leftarrow 1 | S(x_j) = \{1\}\}$, otherwise, it is impossible to meet the demand and the explanation would be $\{x_j \leftarrow 0 | S(x_j) = \{0\}\}$.
- Explaining pruning: if the assignment $x_k \leftarrow 0$ was triggered by the third case, we are sure that we already met the demand, hence the explanation would be $\{x_j \leftarrow 1 | S(x_j) = \{1\}\}$, otherwise, the assignment $x_k \leftarrow 1$ would be explained by $\{x_j \leftarrow 0 | S(x_j) = \{0\}\}$.

On Boolean domains, the hybrid SAT/CP approach we use works as follows:

Propagation: The propagation is performed by a standard CP engine, except that for each pruned value we record the constraint responsible for this pruning (a simple pointer to the constraint is stored). Both original and learned clauses are handled by a dedicated propagator simulating the behavior of a clause base (i.e., using watched literals).

Learning: When a failure is raised, the CDCL standard conflict analysis algorithm is used. The constraint C responsible for the failure is asked to provide an explanation for this failure. The literals of this explanation form the base nogood Ng. Subsequently, any assignment $x \leftarrow v$ such that $lvl(x \leftarrow v) \geq lvl(d)$ where d is the last decision, is removed from Ng and replaced by its explanation by the constraint marked as responsible for it. This process continues until Ng has a Unique Implication Point.

Search: Since a CP library (Mistral[1]) was used to implement this approach, it is possible to use hand made CP heuristics as well as built-in strategies such as VSIDS, backjumping and branching, however, is done as in CDCL algorithms.

3 The *Car-Sequencing* problem

3.1 Problem description

In the car-sequencing problem, n vehicles have to be produced on an assembly line. There are c classes of vehicles and m types of options. Each class $k \in \{1, \ldots, c\}$ is associated with a demand D_k^{class}, that is, the number of occurrences of this class on the assembly line, and a set of options $\mathcal{O}_k \subseteq \{1, \ldots, m\}$. Each option is handled by a working station able to process only a fraction of the vehicles passing on the line. The capacity of an option j is defined by two integers u_j and q_j, such that no subsequence of size q_j may contain more than u_j vehicles requiring option j. A solution of the problem is then a sequence of cars satisfying both demand and capacity constraints. For convenience, we shall also define, for each option j, the corresponding set of classes of vehicles requiring this option $\mathcal{C}_j = \{k \mid j \in \mathcal{O}_k\}$, and the option's demand $D_j = \sum_{k \in \mathcal{C}_j} D_k^{class}$.

[1] https://github.com/ehebrard/Mistral-2.0

3.2 CP Modelling

As a standard CP Model, we use two sets of variables. The first set corresponds to n integer variables $\{x_1, \ldots, x_n\}$ taking values in $\{1, \ldots, c\}$ and standing for the class of vehicles in each slot of the assembly line. The second set of variables corresponds to nm Boolean variables $\{o_1^1, \ldots, o_n^m\}$, where o_i^j stands for whether the vehicle in the i^{th} slot requires option j. For the constraints, we distinguish three sets :

1. *Demand constraints* : for each class $k \in \{1..c\}$, $|\{i \mid x_i = k\}| = D_k^{class}$. This constraint is usually enforced with a Global Cardinality Constraint (GCC) [18] [17].
2. *Capacity constraints* : for each option $j \in \{1..m\}$, we post the constraint ATMOSTSEQCARD$(u_j, q_j, D_j, [o_1^j..o_n^j])$ using the propagator introduced in [21].
3. *Channelling* : Finally, we channel integer and Boolean variables : $\forall j \in \{1, ..., m\}, \forall i \in \{1, ..., n\}$, $o_i^j = 1 \Leftrightarrow x_i \in \mathcal{C}_j$

3.3 Default Pseudo-Boolean and SAT Models

The above CP Model can be easily translated into a pseudo Boolean model since the majority of the constraints are sum expressions.

Variables:

- $c_i^j : \forall i \in [1..n], \forall j \in [1..c]$, c_i^j is *true* iff the class of the ith vehicle is j.
- $o_i^j : \forall i \in [1..n], \forall j \in [1..m]$, o_i^j is *true* iff the ith vehicle requires option j.

Constraints:

- First we have to ensure that at each position i, we have only one class of vehicles: $\forall i \in [1..n], \sum_j c_i^j = 1$
- Second, we link class variables with options:
  - $\forall i \in [1..n], \forall l \in [1..c]$, we have :
    - $\forall j \in \mathcal{O}_l, \overline{c_i^l} \vee o_i^j$
    - $\forall j \notin \mathcal{O}_l, \overline{c_i^l} \vee \overline{o_i^j}$
  - For better propagation, we add the following redundant clause : $\forall i \in [1..n], j \in [1..m], \overline{o_i^j} \vee \bigvee_{l \in \mathcal{C}_j} c_i^l$
- Demand constraints : $\forall j \in [1..c], \sum_i c_i^j = D_j$
- Capacity constraints : $\sum_{l=i}^{i+q_j-1} o_l^j \leq u_j, \forall i \in \{1, \ldots, n - q_j + 1\}$

A SAT Encoding for this problem could translate each sum constraint (in this case only CARDINALITY constraints) into a CNF formula. We will show in Section 5 how such a translation can be improved.

4 Explaining the AtMostSeqCard constraint

We present here a propagation-based algorithm explaining the AtMostSeqCard constraint. For that, we need to recall the corresponding propagator. Let $[x_1, x_2..x_n]$ be a sequence of Boolean variables, u, q and d be integer variables. The AtMostSeqCard constraint is defined as follows :

Definition 1.

$$\text{AtMostSeqCard}(u, q, d, [x_1, \ldots, x_n]) \Leftrightarrow \bigwedge_{i=0}^{n-q} (\sum_{l=1}^{q} x_{i+l} \leq u) \wedge (\sum_{i=1}^{n} x_i = d)$$

In [21], the authors proposed a $O(n)$ filtering algorithm achieving AC on this constraint. We outline the main idea of the propagator.

Let $\mathcal{X} = [x_1..x_n]$ be a sequence of variables, and S a partial instantiation over these variables. The procedure `leftmost` returns an instantiation $\overrightarrow{w}_S \supseteq S$ of maximum cardinality by greedily assigning the value 1 from left to right while respecting the AtMost constraints. Let $\overrightarrow{w}^i_S$ denote the partial instantiation $\overrightarrow{w}_S$ at the beginning of iteration i, and let $\overrightarrow{w}^1_S = S$. The value $\max_S(i)$ denotes the maximum minimum cardinality, with respect to the current domain $\overrightarrow{w}^i_S$, of the q subsequences involving x_i. It is computed alongside $\overrightarrow{w}_S$ and will be useful to explain the subsequent pruning. It is formally defined as follows (where $\min(\overrightarrow{w}^i_S(x_k)) = 0$ if $k < 1$ or $k > n$):

$$\max_S(i) = \max_{j \in [1..q]} (\sum_{k=i-q+j}^{i+j-1} \min(\overrightarrow{w}^i_S(x_k)))$$

Definition 2. *The outcome of the procedure* `leftmost` *can be recursively defined using* $\max_S$*: at each step* i*,* `leftmost` *adds the assignment* $x_i \leftarrow 1$ *iff this assignment is consistent with* $\overrightarrow{w}^i_S$ *and* $\max_S(i) < u$*, it adds the assignment* $x_i \leftarrow 0$ *otherwise.*

Example 1. For instance, consider the execution of the procedure `leftmost` on the constraint AtMostSeqCard$(2, 4, 6, [x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}])$. We suppose that we start from the partial instantiation $\{x_2 \leftarrow 0, x_6 \leftarrow 1, x_8 \leftarrow 0\}$. Initially, we have the following structures, for each i representing an iteration (and also the index of a variable):

	1	2	3	4	5	6	7	8	9	10
$\overrightarrow{w}^1_S(x_i)$	$\{0,1\}$	0	$\{0,1\}$	$\{0,1\}$	$\{0,1\}$	1	$\{0,1\}$	0	$\{0,1\}$	$\{0,1\}$
$\overrightarrow{w}^2_S(x_i)$	1	0	$\{0,1\}$	$\{0,1\}$	$\{0,1\}$	1	$\{0,1\}$	0	$\{0,1\}$	$\{0,1\}$
$\overrightarrow{w}^3_S(x_i)$	1	0	$\{0,1\}$	$\{0,1\}$	$\{0,1\}$	1	$\{0,1\}$	0	$\{0,1\}$	$\{0,1\}$
$\overrightarrow{w}^4_S(x_i)$	1	0	1	$\{0,1\}$	$\{0,1\}$	1	$\{0,1\}$	0	$\{0,1\}$	$\{0,1\}$
...										
$\overrightarrow{w}^{11}_S(x_i)$	1	0	1	0	0	1	1	0	0	1
$\max_S(i)$	0	1	1	2	2	1	1	2	2	1

The partial solution $\overrightarrow{w}^1_S$ is equal to S. Then at each step i, `leftmost` adds the assignment $x_i \leftarrow 1$ or $x_i \leftarrow 0$ according to Definition 2. For instance, at the begining of

step 4, the subsequences to consider are $[x_1, x_2, x_3, x_4]$, $[x_2, x_3, x_4, x_5]$, $[x_3, x_4, x_5, x_6]$ and $[x_4, x_5, x_6, x_7]$, of cardinality 2, 1, 2 and 1, respectively, with respect to the instantiation $\overrightarrow{w}^4_S(x_i)$. The value of $\max_S(4)$ is therefore 2.

To detect failure, we simply need to run this procedure and check that the final cardinality of $\overrightarrow{w}_S$ is greater than or equal to the demand d, and we shall see that we can explain pruning by using essentially the same procedure.

However, in order to express declaratively the full propagator, we need the following further steps: The same procedure is applied on variables in reverse order $[x_n..x_1]$, yielding the instantiation $\overleftarrow{w}_S$. Observe that the returned instantiations $\overrightarrow{w}_S$ and $\overleftarrow{w}_S$ assign every variable in the sequence to either 0 or 1. We denote respectively $L_S(i)$ and $R_S(i)$ the sum of the values given by $\overrightarrow{w}_S$ (resp. $\overleftarrow{w}_S$) to the i first variables (resp. $n - i + 1$ last variables). That is:

$$L_S(i) = \sum_{k=1}^{i} \min(\overrightarrow{w}_S(x_k)) \quad , \quad R_S(i) = \sum_{k=i}^{n} \min(\overleftarrow{w}_S(x_k))$$

Now we have all the tools to define the propagator associated to the constraint AtMostSeqCard described in [21], and which is a conjunction of GAC on the AtMost constraints on each subsequence, of Cardinality constraint $\sum_{i=1}^{n} x_i = d$, and of the following:

$$\text{AtMostSeqCard}(S) = \begin{cases} S, & \text{if } L_S(n) > d \\ \bot, & \text{if } L_S(n) < d \\ S & \cup \{x_i \leftarrow 0 \mid S(x_i) = \{0,1\} \\ & \quad \& \, L_S(i) + R_S(i) \leq d\} \\ & \cup \{x_i \leftarrow 1 \mid S(x_i) = \{0,1\} \\ & \quad \& \, L_S(i-1) + R_S(i+1) < d\} \text{ otherwise} \end{cases} \tag{4.1}$$

If a failure/pruning is detected by the Cardinality or an AtMost constraint, then it is easy to give an explanation similarly to Section 2. However, if a failure or a pruning is due to the propagator defined in equation 4.1, then we need to specify how to generate a relevant explanation.

We start by giving an algorithm explaining a failure. We show after that how to use this algorithm to explain pruning.

4.1 Explaining Failure

Suppose that the propagator detects a failure at a given level l. The original instantiation S would be a possible naive explanation expressing this failure. We propose in the following a procedure generating more compact explanations.

In example 2, the instantiation $S = \{x_1 \leftarrow 1, x_3 \leftarrow 0, x_6 \leftarrow 0\}$ is subject to $\text{AtMostSeqCard}(2, 5, 3, [x_1..x_6])$. S is unsatisfiable since $L_S(6) < d$. Consider now the sequence $S^* = \{x_6 \leftarrow 0\}$. The result of `leftmost` on S and S^* is the identical. Therefore, S^* and S are both valid explanations for this failure, however

S^* is shorter. The idea behind our algorithm for computing shorter explanations is to characterise which assignments will have no impact on the behavior of the propagator, and thus are not necessary in the explanation.

Example 2.

S	1	.	0	.	.	0
$\overrightarrow{w}(S)$	1	1	0	0	0	0
$max(S)$	1	1	2	2	2	1
$L(S)$	1	2	2	2	2	2
					$d = 3$	
				$L(6) = 2$		
				$\rightarrow$ Failure		

S^*	.	.	.	.	.	0
$\overrightarrow{w}(S^*)$	1	1	0	0	0	0
$max(S^*)$	1	1	2	2	2	1
$L(S^*)$	1	2	2	2	2	2
					$d^* = 3$	
				$L^*(6) = 2$		
				$\rightarrow$ Failure		

Let $I = [x_{k+1}..x_{k+q}]$ be a (sub)sequence of variables of size q and S be a partial instantiation. We denote $card(I, S)$ the minimum cardinality of I under the instantiation S, that is: $card(I, S) = \sum_{x_i \in I} \min(S(x_i))$.

Lemma 1. *If* $S^* = S \setminus (\{x_i \leftarrow 0 \mid max_S(i) = u\} \cup \{x_i \leftarrow 1 \mid max_S(i) \neq u\})$ *then* $\overrightarrow{w}_S = \overrightarrow{w}_{S^*}$.

Proof. Suppose that there exists an index $i \in [1..n]$ s.t. $\overrightarrow{w}_S(x_i) \neq \overrightarrow{w}_{S^*}(x_i)$ and let k be the smallest index verifying this property. Since the instantiation S^* is a subset of S (i.e., S^* is weaker than S) and since `leftmost` is a greedy procedure assigning the value 1 whenever possible from left to right, it follows that $\overrightarrow{w}_S(x_k) = 0$ and $\overrightarrow{w}_{S^*}(x_k) = 1$. Moreover, it follows that $\max_S(k) = u$ and $\max_{S^*}(k) < u$. In other words, there exists a subsequence I containing x_k s.t the cardinality of I in $\overrightarrow{w}^k_S$ (i.e. $card(I, \overrightarrow{w}^k_S)$) is equal to u, and the cardinality of I in $\overrightarrow{w}^k_{S^*}$ ($card(I, \overrightarrow{w}^k_{S^*})$) is less than u. From this we deduce that there exists a variable $x_j \in I$ such that $\min(\overrightarrow{w}^k_S(x_j)) = 1$ and $\min(\overrightarrow{w}^k_{S^*}(x_j)) = 0$.

First, we cannot have $j < k$. Otherwise, both instantiations $\overrightarrow{w}^k_S(x_j)$ and $\overrightarrow{w}^k_{S^*}(x_j)$ contain an assignment for x_j, and therefore we have $\overrightarrow{w}^k_S(x_j) = \{1\}$ and $\overrightarrow{w}^k_{S^*}(x_j) = \{0\}$, which contradicts our hypothesis that k is the smallest index of a discrepancy.

Second, suppose now that $j > k$. Since we have $card(I, \overrightarrow{w}^k_S) = u$, we can deduce that $card(I, \overrightarrow{w}^j_S) = u$. Indeed, when going from iteration k to iteration j, `leftmost` only adds assignments, and therefore $card(I, \overrightarrow{w}^j_S) \geq card(I, \overrightarrow{w}^k_S)$. It follows that $\max_S(j) = u$, and by construction of S^*, we cannot have $x_j \leftarrow 1 \in S \setminus S^*$. However, it contradicts the fact that $\min(\overrightarrow{w}^k_S(x_j)) = 1$ and $\min(\overrightarrow{w}^k_{S^*}(x_j)) = 0$.

□

Theorem 1. *If* S *is a valid explanation for a failure and* $S^* = S \setminus (\{x_i \leftarrow 0 \mid max_S(i) = u\} \cup \{x_i \leftarrow 1 \mid max_S(i) \neq u\})$*, then* S^* *is also a valid explanation.*

Proof. By Lemma 1, we know that the instantiations $\overrightarrow{w}_S$ and $\overrightarrow{w}_{S^*}$, computed from, respectively the instantiations S and S^* are equal. In particular, we have $L_S(n) = L_{S^*}(n)$ and therefore AtMostSeqCard$(S) = \bot$ iff AtMostSeqCard$(S^*) = \bot$.

□

Theorem 1 gives us a linear time procedure to explain failure. In fact, all the values $\max_S(i)$ can be generated using one call of `leftmost`. Example 3 illustrates the explanation procedure.

Example 3.

S	1	0	1	0	0	.	.	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1
$max_S(i)$	2	**2**	2	**2**	**2**	1	2	**2**	**2**	**2**	2	2	**2**	**2**	**2**	1	**1**	1	1	1	1	**1**
$\overrightarrow{w}_S(x_i)$	1	0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1
$L_S(i)$	1	1	2	2	2	3	3	3	3	3	4	5	5	5	5	5	6	6	6	6	6	7
S^*	1	.	1	.	.	.	.	.	.	.	1	1	.	.	.	0	.	0	0	0	0	.
$max_{S^*}(i)$	2	2	2	2	2	1	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1
$\overrightarrow{w}_{S^*}(x_i)$	1	0	1	0	0	1	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	1

We illustrate here the explanation of a failure on ATMOSTSEQCARD$(2, 5, 8, [x_1..x_{22}])$. The propagator returns a failure since $L_S(22) = 7 < d = 8$. The default explanation corresponds to the set of all the assignments in this sequence, whereas our procedure shall generate a more compact explanation by considering only the assignments in S^*. Bold face values in the $max_S(i)$ line represent the variables that will not be included in S^*. As a result, we reduce the size of the explanation from 20 to 9 assignments. Note, however, that S^* is not optimal (w.r.t the size) since `leftmost` returns exactly the same result on S^* and $S^* \setminus x_1 \leftarrow 1$ (hence the same failure).

4.2 Explaining Pruning

Suppose that a pruning $x_i \nleftarrow v$ was triggered by the propagator in equation 4.1 at a given level l on S (i.e. propagating ATMOSTSEQCARD(S) implies $x_i \nleftarrow v$). Consider the partial instantiation $S_{x_i \leftarrow v}$ identical to S on all assignments at level l except for $x_i \leftarrow v$ instead of $x_i \nleftarrow v$. By construction $S_{x_i \leftarrow v}$ is unsatisfiable. Let S^* be the explanation expressing this failure using the previous mechanism. We have then $S^* \setminus x_i \leftarrow v$ as a valid explanation for the pruning $x_i \nleftarrow v$.

5 SAT-Encoding for the ATMOSTSEQCARD constraint

In this section we present SAT-encodings for the ATMOSTSEQCARD constraint and relate them to existing encoding techniques. First we describe a translation of Boolean cardinality constraints by a variant of the sequential counter encoding [22]. This encoding can be used to translate the decomposition of ATMOSTSEQCARD into cardinality and ATMOST. Then we introduce an encoding taking advantage of the globality of ATMOSTSEQCARD by reusing the auxiliary variables for the cardinality constraint and integrating the sequence of ATMOST constraints. This technique is similar to the encoding of GEN-SEQUENCE in [2] and the decomposition of SEQUENCE into cumulative sums in [5]. Finally, for our last encoding we add redundant clauses for each capacity constraint in order to increase propagation.

5.1 Sequential Counter

We describe the translation of $\sum_{i \in [1..n]} x_i = d$ to CNF by a sequential counter encoding (SC). For technical reasons we use an additional variable x_0 s.t. $D(x_0) = \{0\}$.

- Variables:
  - $s_{i,j}$: $\forall i \in [0..n]$, $\forall j \in [0..d+1]$, $s_{i,j}$ is *true* iff for the positions $[0..i]$ x_i is at least j times true.

– Encoding: $\forall i \in [1..n]$
  - Clauses for restrictions on the same level: $\forall j \in [0..d+1]$
    1. $\neg s_{i-1,j} \vee s_{i,j}$
    2. $x_i \vee \neg s_{i,j} \vee s_{i-1,j}$
  - Clauses for increasing the counter, $\forall j \in [1..d+1]$
    3. $\neg s_{i,j} \vee s_{i-1,j-1}$
    4. $\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j}$
  - Initial values for the bounds of the counter:
    5. $s_{0,0} \wedge \neg s_{0,1} \wedge s_{n,d} \wedge \neg s_{n,d+1}$

We refer to the clauses by numbers 1 to 5. The variables $s_{i,j}$ represent the bounds for cumulative sums of the sequence $x_1 \dots x_i$. The encoding is best explained by visualising $s_{i,j}$ as a two dimensional grid with positions (horizontal) and cumulative sums (vertical). The binary clauses 1 and 3 ensures that the counter (i.e. the variables representing the cumulative sums) is monotonically increasing. Clauses 2 and 4 control the interaction with the variables x_i. If x_i is true, then the counter has to increase at position i whereas if x_i is false an increase is prevented at position i. The conjunction 5 sets the initial values for the counter to start counting at 0 and ensures that the partial sum at position n is equal to d.

Example 4. We illustrate the auxiliary variables of SC. Given a sequence of 8 variables and $d = 2$. To the left the initial condition of the variables, followed assigning x_2 to true and then to the right x_7 to true.

3	0	0	0	0	0	0	0	0	0
2	0	0	.	.	.	.	.	.	1
1	0	.	.	.	.	.	.	1	1
0	1	1	1	1	1	1	1	1	1
$s_{i,j}$	0	1	2	3	4	5	6	7	8
x_i		.	.	.	.	.	.	.	.

3	0	0	0	0	0	0	0	0	0
2	0	0	.	.	.	.	.	.	1
1	0	.	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1
$s_{i,j}$	0	1	2	3	4	5	6	7	8
x_i		1	.	.	.	.	.	.	.

3	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	1	1
1	0	0	1	1	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1
$s_{i,j}$	0	1	2	3	4	5	6	7	8
x_i		0	1	0	0	0	0	1	0

The SC encoding has the following property regarding propagation:

Proposition 1. *Unit Propagation on the SC encoding enforces* GAC *on the cardinality constraint* $\sum_{i \in [1 \dots n]} x_i = d$.

The encoding introduces $n \cdot (d+2)$ auxiliary variables. However, preprocessing prunes the lower and upper row $j = 0$ and $j = d+1$ and further d^2 variables by clause 2. The number of unassigned variables is thus $(n-d) \cdot d$ and the number of clauses is bounded by $4 \cdot n \cdot d$.

By a variant of conjunction 5 we can encode an ATMOST constraint in the same way. If $s_{n,d}$ is removed from 5 the counter allows all assignments where at most d variables of x_i, $i \in [1..n]$ are true. We refer to this encoding as SCA, Sequential Counter for ATMOST constraints.

The encoding in [22] translates inequalities (as SCA) but uses only half the number of clauses and also enforces GAC on ATMOST. However, their encoding does not force all auxiliary variables when all x_i are assigned and this effectively increases the model count which can lead to unnecessary search. Furthermore, the auxiliary variables used

here are tightened by the redundant clauses 1 and 3 and branching on these variables is facilitated. The encoding in [2] of the more general AMONG constraint has similarities to a counter encoding. Our encoding consists only of binary and ternary clauses whereas their encoding due to the more general constraint they translate, introduces long clauses up to the size of d literals.

5.2 Extension to ATMOSTSEQCARD

For the ATMOSTSEQCARD we add the following clauses by reusing the auxiliary variables $s_{i,j}$. $\forall i \in [q..n], \forall j \in [u..d+1]$:

6. $\quad \neg s_{i,j} \vee s_{i-q,j-u}$

We refer to this encoding using clauses 1 to 6 as SCS, Sequential Counter with Sequence. The number of additional clauses is bounded by $n \cdot d$, so the complete encoding consists of $5 \cdot n \cdot d$ clauses.

Proposition 2. *Unit Propagation on the SCS encoding representing the* ATMOSTSEQCARD *constraint detects dis-entailment on any partial assignment.*

Proof. We omit the proof for this proposition due to space limits. It follows a similar structure as the proof for Theorem 3 in [2] since the encoding SCS resembles a special case of the encoding of the GEN-SEQUENCE constraint. A difference to their encoding lies in the auxiliary variables that encode the equality $\sum_{l=1}^{i} x_l = j$ and the resulting change for the clauses. □

Observe that SCS will have improved propagation on the auxiliary variables due to the binary clauses 1, 3 and 6 that restrict the counter to be in a consistent state and branching on these variables is promoted independently of the concrete assignment to x_i. Note also that SCS is checking dis-entailment and pruning values in many cases, however, it does not fully enforce GAC on ATMOSTSEQCARD. See the following example.

Example 5. Let $u = 1$,$q = 2$,$d = 2$,$n = 5$ and let x_3 be true, then UP does not enforce x_2 nor x_4 to false. Setting them to true will lead to a conflict by UP through clauses 4 and 6 on positions 2, 3 and 4.

3	0 0 0 0 0 0
2	0 0 0 . . 1
1	0 . . 1 1 1
0	1 1 1 1 1 1
$s_{i,j}$	0 1 2 3 4 5
x_i	. . 1 . .

This example motivates us to define an encoding on ATMOSTSEQCARD extending SCS by re-encoding each ATMOST constraints to enforce this missing propagation. Each ATMOST constraint is separately translated to CNF by an SCA encoding in addition to clauses 6. We expect this complete encoding to have the advantages of both previous encodings.

6 Experimental results

We tested the different approaches on the three data sets available on the CSPLib [12]. The first set of "historical" instances contains 5 unsatisfiable and 4 satisfiable instances of relatively small size (100 cars). The second set contains 70 instances generated with varying usage rate. All instances in this set are satisfiable and involve 200 cars. These two sets were used in most of the CP literature on this problem. The third set, proposed by Gagné, features larger instances divided into three sets of ten each, involving respectively 200, 300 and 400 cars. Seven of these instances were solved using local search algorithms. To the best of our knowledge the remaining 23 instances have never been proved unsatisfiable.

We grouped the instances into three categories to help us better outline the differences between the methods we tested.

- In the first category (`sat[easy]`), we consider the 70 satisfiable instances of the second set as well as the 4 satisfiable instances of the first set. All these instances are extremely easy for all the methods we introduce in this paper.
- In the second category (`sat[hard]`), we consider the 7 known satisfiable instances of the second set. These instances are challenging and were often out of reach of previous systematic approaches.
- In the third category (`unsat/unknown`), we consider the remaining 5 unsatisfiable instances of the first set as well as the 23 unknown instances form the third set. Those instances are challenging and indeed open for 23 of them.

All experiments ran on Intel Xeon CPUs 2.67GHz under Linux. For each instance, we launched 5 randomized runs with a 20 minutes time cutoff. We ran the following methods:

- Minisat (version 2.2.0) with default parameter settings on three variants of the SAT encoding. Links between classes and options as well as the constraint for exactly one class of vehicle per position are translated as in the basic model. For each option and for each class we encode one AtMostSeqCard. The capacity constraint for options is given by the problem specification whereas for each class we choose the strictest capacity constraints among all its options. The three models differ only in how the AtMostSeqCard constraint is translated (w.r.t Section 5):
  1. *SAT (1)* encodes the basic model by using SC for the global demand and SCA for each window of the capacity constraint.
  2. *SAT (2)* encodes each AtMostSeqCard by the SCS.
  3. *SAT (3)* combines *SAT (1)* and *SAT (2)*.
- Mistral as a hybrid CP/SAT solver (Section 2) using the proposed explanation for the AtMostSeqCard constraint. We tested four branching heuristics :
  1. *hybrid (VSIDS)* uses VSIDS;
  2. *hybrid (Slot)* uses the following CP heuristic (denoted by $Slot$) : we branch on $option$ variables from the middle of the sequence and towards the extremities following the first unassigned $Slot$. The options are firstly evaluated by their dynamic usage rate[23] then lexicographically compared.

3. *hybrid (Slot → VSIDS)* first uses the CP heuristic, then switches after 100 non-improving restarts to VSIDS.
4. *hybrid (VSIDS → Slot)* uses VSIDS and switches after 100 non-improving restarts to the CP heuristic.

We also used two "control" approaches ran in the same setting:

- *pseudo Boolean*: MiniSat+ [10] on a straightforward pseudo-Boolean encoding, similar to that described in Section 3 except that the AtMostSeqCard constraint is decomposed into Cardinality and AtMost constraints.
- *CP*: Mistral without clause learning on the model described in Section 3 using the $Slot$ branching.

For each considered data set, we report the total number of successful runs (*#suc*).[2] Then, we report the number of fail nodes (*fails*) and the CPU time (*time*) in seconds both averaged over all successful random runs on every instance. We emphasize the statistics of the best method (w.r.t. *#suc*, ties broken by CPU time) for each data set using bold face fonts.

Table 1: Evaluation of the models

Method	`sat[easy]` (74 × 5)			`sat[hard]` (7 × 5)			`unsat/unknown` (28 × 5)		
	#suc	avg fails	time	#suc	avg fails	time	#suc	avg fails	time
SAT (1)	370	2073	1.71	28	337194	282.35	**85**	**249301**	**105.07**
SAT (2)	370	1077	1.18	30	42790	33.02	67	217103	182.23
SAT (3)	370	667	1.30	35	50233	66.23	74	137639	70.47
hybrid (VSIDS)	370	903	0.23	16	207211	286.32	35	177806	224.78
hybrid (VSIDS → Slot)	370	739	0.23	35	76256	64.52	37	204858	248.24
hybrid (Slot → VSIDS)	370	132	0.04	34	4568	2.50	37	234800	287.61
hybrid (Slot)	370	132	0.04	**35**	**6304**	**3.75**	23	174097	299.24
CP	**370**	**43.06**	**0.03**	35	57966	16.25	0	-	-
pseudo Boolean	277	538743	236.94	0	-	-	43	175990	106.92

Overall efficiency We first observe that most of the approaches we introduce in this paper significantly improve the state of the art, at least for systematic methods. For instance, in the experiments reported in [21] several instances of the set `sat[hard]` were not solved within a 20 minutes cutoff. Moreover we are not aware of other systematic approaches being able to solve these instances.

More importantly, we are able to close 13 out of the 23 large open instances proposed by Gagné. The set of open instances is now reduced to `pb_200_02`, `pb_200_06`, `pb_200_08`, `pb_300_02`, `pb_300_06`, `pb_300_09`, `pb_400_01`, `pb_400_02`, `pb_400_07`, `pb_400_08`.

[2] They all correspond to solutions found for the two first categories, and unsatisfiability proofs for the last.

Finding solutions quickly: Second, on satisfiable instances, we observe that pure CP approaches are difficult to outperform. It must be noticed that the results reported for *CP* are significantly better than those previously reported for similar approaches. For instance, the best methods introduced in [25] take several seconds on most instances of the first category and were not able to solve two of them within a one hour time cutoff. Moreover in [21], the same solver on the same model had a similar behavior on the first category (`sat[easy]`), however was only able to solve 2 instances of the second category (`sat[hard]`). The only difference with the method we ran in this paper is that restarts according to the Luby sequence were used.

However, overall, the best method on satisfiable instances is the hybrid solver using a pure CP heuristic. Moreover, we can see that even with a "blind" heuristic, MiniSat on the strongest encodings has extremely good results (all satisfiable instances could be solved with a larger time cutoff).

This study shows that propagation is very important to find solutions quickly when they exist, by keeping the search "on track" and avoiding exploring large unsatisfiable subtrees. There are several evidences for this: First, the pure pseudo Boolean model (*pseudo Boolean*) features no non-trivial propagation and is indeed very poor on `sat[easy]` and `sat[hard]`. Second, the best SAT models for those two categories are those providing the tightest propagation. Last, previous CP approaches that did not enforce GAC on the AtMostSeqCard constraint are all dominated by *CP*.

Proving unsatisfiability: Third, for proving unsatisfiability, our results clearly show that clause learning is by far the most critical factor. Surprisingly, a stronger propagation is not always beneficial when building a proof using clause learning, as shown by the results of the different encodings. One could even argue for a negative correlation, since the "lightest" encodings are able to build more proofs than stronger ones. Similarly, the pure pseudo Boolean model performs much better comparatively to the satisfiable case. The hybrid models are slightly worse than pseudo Boolean but far better than pure CP approach that was not able to prove any case of unsatisfiability. To mitigate this observation, however, notice that other CP models with strong filtering, using the Global Sequencing Constraint [19], or a conjunction of this constraint and AtMostSeqCard [21, 25] were able to build proofs for some of the 5 unsatisfiable instances of the CSPLib. However, these models were not sufficient to solve any of the 23 larger unsatisfiable instances.

7 Conclusion

We proposed and compared hybrid CP/SAT models for the car-sequencing problem against several SAT-encodings. Both approaches exploit the AtMostSeqCard constraint. In particular, we proposed a linear time procedure for explaining failure and pruning as well as advanced SAT-encodings for this constraint. Experimental results emphasize the importance of advanced propagation for searching feasible solutions and of clause learning for building unsatisfiability proofs.

References

1. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In *Proceedings of ICCAD*, pages 450–457, 2002.
2. Fahiem Bacchus. GAC Via Unit Propagation. In *Proceedings of CP*, pages 133–147, 2007.
3. Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
4. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
5. Sebastian Brand, Nina Narodytska, Claude-Guy Quimper, Peter J. Stuckey, and Toby Walsh. Encodings of the Sequence Constraint. In *Proceedings of CP*, pages 210–224, 2007.
6. Hadrien Cambazard. *Résolution de problmes combinatoires par des approches fondées sur la notion dexplication*. PhD thesis, Ecole des mines de Nantes, 2006.
7. Hadrien Cambazard and Narendra Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. *Constraints*, 11(4):295–313, 2006.
8. Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.
9. Heidi Dixon. *Automating Pseudo-Boolean Inference within a DPLL Framework*. PhD thesis, University of Oregon, 2004.
10. Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
11. Ian P. Gent. Arc Consistency in SAT. In *Proceedings of ECAI*, pages 121–125, 2002.
12. Ian P. Gent and Toby Walsh. CSPlib: a benchmark library for constraints, 1999.
13. George Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, 2008.
14. George Katsirelos and Fahiem Bacchus. Generalized NoGoods in CSPs. In *Proceedings of AAAI*, pages 390–396, 2005.
15. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC*, pages 530–535, 2001.
16. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via Lazy Clause Generation. *Constraints*, 14(3):357–391, 2009.
17. Claude-Guy Quimper, Alexander Golynski, Alejandro López-Ortiz, and Peter van Beek. An Efficient Bounds Consistency Algorithm for the Global Cardinality Constraint. *Constraints*, 10(2):115–135, 2005.
18. Jean Charles Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of AAAI*, volume 2, pages 209–215, 1996.
19. Jean-Charles Régin and Jean-François Puget. A Filtering Algorithm for Global Sequencing Constraints. In *Proceedings of CP*, pages 32–46, 1997.
20. Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic CSP. In *Proceeding of ICTAI*, pages 48–55, 1993.
21. Mohamed Siala, Emmanuel Hebrard, and Marie-Jose Huguet. An Optimal Arc Consistency Algorithm for a Chain of Atmost Constraints with Cardinality. In *Proceedings of CP*, pages 55–69, 2012.
22. Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In *Proceedings of CP*, pages 827–831, 2005.
23. Barbara M. Smith. Succeed-first or Fail-first: A Case Study in Variable and Value Ordering, 1996.
24. Christine Solnon, Van Dat Cung, Alain Nguyen, and Christian Artigues. The car sequencing problem: Overview of state-of-the-art methods and industrial case-study of the ROADEF'2005 challenge problem. *European Journal of Operational Research*, 191:912–927, 2008.

25. Willem J. van Hoeve, Gilles Pesant, Louis-Martin Rousseau, and Ashish Sabharwal. New Filtering Algorithms for Combinations of Among Constraints. *Constraints*, 14(2):273–292, 2009.
26. Toby Walsh. SAT v CSP. In *Proceedings of CP*, pages 441–456, 2000.