# Efficient Reasoning for Nogoods in Constraint Solvers with BDDs

Sathiamoorthy Subbarayan

Computational Logic and Algorithms Group
IT University of Copenhagen, Denmark
sathi@itu.dk

**Abstract.** When BDDs are used for propagation in a constraint solver with nogood recording, it is necessary to find a small subset of a given set of variable assignments that is enough for a BDD to imply a new variable assignment. We show that the task of finding such a minimum subset is NP-complete by reduction from the hitting set problem. We present a new algorithm for finding such a minimal subset, which runs in time linear in the size of the BDD representation. In our experiments, the new method is up to ten times faster than the previous method, thereby reducing the solution time by even more than 80%. Due to linear time complexity the new method is able to scale well.

## 1 Introduction

Many useful functions have compact *Binary decision diagram* (BDD) [1] representations. Hence, the BDDs has attracted attention as a constraint representation [2–8]. The BDDs have been used in many applications, including: verification, configuration and fault-tree analysis.

The *nogood recording* [9, 10] is a technique in constraint solvers to find a subset of the variable assignments made upto a dead-end in a search tree, such that the found subset could independently lead to dead-ends. By recording such subsets called *nogoods* and by preventing similar assignment patterns, the search effort can be drastically reduced.

For a given set of variable assignments $X$, if the propagation of $X$ in a constraint $c$ implies a variable assignment $(v := a)$, denoted $X \wedge c \Rightarrow (v := a)$, then a *reason* $R$ is a subset of $X$, such that $R \wedge c \Rightarrow (v := a)$. Finding small reasons is essential for nogood recording. The nogood recording plays a major part in the successful SAT solvers. The adoption of the nogood recording in general constraint solvers requires efficient methods for finding small reasons in every important constraint representation, including BDDs.

This paper focuses on finding small reasons in BDDs. We show that the task of finding a minimum-sized reason in BDDs is NP-complete by reduction from the hitting set problem. We also present a new algorithm for finding minimal-sized reasons, which runs in time linear in the size of the BDD representation. We then empirically demonstrate the usefulness of the new algorithm over a previous method. In our experiments, the new method scales better, and is upto 10 times faster than the previous one.

## 2 Definitions

### 2.1 The Constraint Satisfaction Problem

A *constraint satisfaction problem* (CSP) instance is a triple $(V, D, C)$, where $V$ is a set of variables, $D$ is a set of finite domains, one domain $d_i \in D$ for each variable $v_i \in V$, and $C$ is a set of constraints. Each constraint $c_i \in C$ is a pair of the form $(s_i, r_i)$, where $s_i \subseteq V$ and $r_i$ is a *relation* over the variables in $s_i$.

An *assignment* $X$ is a set like $\{v_{x_1} := a_{x_1}, v_{x_2} := a_{x_2}, \ldots, v_{x_{|X|}} := a_{x_{|X|}}\}$. The variable assignment $(v_{x_i} := a_{x_i})$ fixes the value of $v_{x_i}$ to $a_{x_i}$, where $a_{x_i} \in d_{x_i}$. An assignment $X$ is *full* if $|X| = |V|$, *partial* otherwise. A *solution* to a CSP is a full assignment $S$, such that for any constraint $(s_i, r_i) \in C$, the assignment $S$ restricted to $s_i$ belongs to the relation $r_i$, i.e., $S_{|s_i} \in r_i$. For a given assignment $X$, a constraint $c_i$ *implies* a variable assignment $(v := a)$, denoted $X \wedge c_i \Rightarrow (v := a)$, if every tuple in the relation $r_i$ containing $X_{|s_i}$ also contains $(v := a)$.

### 2.2 The Binary Decision Diagrams

A *reduced ordered binary decision diagram* (BDD) [1] is a *directed acyclic graph* with two terminal nodes, one marked with 1 (true) and the other with 0 (false). The Figure 2 (a) and Figure 3 (a) show two example BDDs.

Each non-terminal node $n$ is associated with a Boolean variable $var(n)$. Each node $n$ has two outgoing edges, one *dashed* and another *solid*. The occurrence of variables in any path has to obey a *linear order*. Also, isomorphic subgraphs will be merged together, and a node $n$ with both its outgoing edges reaching the same node $n_c$ will be removed with all the incoming edges of $n$ made to reach $n_c$ directly. A BDD will be represented by its root node. The size of a BDD $b$, $|b|$, is the number of non-terminal nodes. For a given BDD, the term $solid(n_1)$ evaluates to $n_2$ iff $(n_1, n_2)$ is a solid edge in the BDD. Similarly, $dashed(n_1)$ evaluates to $n_2$ iff $(n_1, n_2)$ is a dashed edge.

The variable assignment *corresponding to an edge* $(n_1, n_2)$ is $(var(n_1) := a)$, where $a = true$ iff $n_2 = solid(n_1)$. Consider a path $p =< n_1, n_2, \ldots, n_l >$ in a BDD with $n_l = 1$, from a node $n_1$ to the 1-terminal. The assignment $X_p$ *corresponding to the path* $p$ is $X_p = \{(var(n_i) := a) \mid 1 \leq i \leq (l-1),\ (n_{i+1} = solid(n_i)) \Leftrightarrow (a = true)\}$. The $X_p$ is the set of the variable assignments corresponding to each edge in the path. The path $p$ is a *solution path* if $n_1 = b$ and $n_l = 1$, i.,e, starts from the root node.

A BDD $b$ represents a Boolean function $f$ iff for any solution $S$ to $f$, there exists a solution path $p$ in $b$, such that $X_p \subseteq S$. We may denote the function represented by a BDD $b$ by $b$ itself. If $S$ is a solution of $f$, we may specify $S \in f$. The set of solutions $S_p$ corresponding to a solution path $p$ is $S_p = \{S \mid X_p \subseteq S,\ S \in b\}$. We denote $(v := a) \in p$ to specify that there exists a $S \in S_p$ such that $(v := a) \in S$. Similarly, we denote $X \in p$ if there exists a $S \in S_p$ such that $X \subseteq S$. Note, $(v := a) \in p$ mentions that either there occurs an edge $(n_i, n_{i+1})$ in $p$ whose corresponding assignment is $(v := a)$, or there is no node $n_i$ in the path $p$ such that $var(n_i) = v$.

Although a BDD representing a Boolean function could be exponential in the number of variables in the function, for several practically useful functions the equivalent BDDs are of small size. Hence, the BDDs have found widespread usage in several applications.

### 2.3 Representing Constraints Using BDDs

To simplify the presentation, we assume that all the variables in a given CSP have Boolean domain. Given a general CSP, we can encode it using Boolean variables. For example, using the *log-encoding* method, a non-Boolean variable $v \in V$ with domain $d$ can be substituted by $\lceil log\ |d| \rceil$ Boolean variables, matching each value in $d$ to a unique assignment of the introduced $\lceil log\ |d| \rceil$ Boolean variables.

Each constraint $c_i \in C$ is hence a Boolean function defined over $s_i$, with the function mapping an assignment for $s_i$ to true iff the assignment belongs to $r_i$.

For $X = \{v_{x_1} := a_{x_1}, v_{x_2} := a_{x_2}, \ldots, v_{x_{|X|}} := a_{x_{|X|}}\}$, the Boolean function obtained by the conjunction of the variable assignments in $X$ is also denoted by $X$, i.e., $X = \bigwedge_{1 \leq i \leq |X|} (v_{x_i} = a_{x_i})$, which will be clear from the context.

Given a CSP with several constraints, some of the constraints' function might be represented by compact BDDs. The BDDs of some of the constraints might result in obtaining helpful inferences to speed-up the constraint solver. Hence, the BDDs has attracted attention as a constraint representation [2–8].

### 2.4 The Nogoods

A *nogood* [9, 10] of a CSP is a partial assignment $N$, such that for any solution $S$ of the CSP, $N \nsubseteq S$. Hence, a nogood $N$ cannot be part of any solution to the CSP. In a typical constraint solver, an initial empty assignment $X = \{\}$ will be extended by both the *branching decisions* and the variable assignments *implied* by the decisions, and the partial assignment $X$ will be reduced by the *backtracking steps*. The extensions and reductions will go on until either $X$ becomes a solution or all possible assignments are exhausted.

A backtracking step occurs when the assignment $X$ cannot be extended to a solution. The *nogood recording*, if implemented in a constraint solver, will be invoked just before each backtracking step. The nogood recording involves finding and storing a subset $N$ of the partial assignment X, such that $N$ is a nogood. Such nogoods can be used to prevent some bad branching choices in the future and hence speed-up the solution process. This paper focuses on a building block of nogood recording and can be understood independently. We refer the interested reader to [9–12, 7] for details on the whole nogood recording process.

### 2.5 The Reasons for Variable Assignment

A building block of nogood recording is to find a small subset $R$ of an assignment $X$ that is a *reason* for the implication of a variable. If $X \wedge c \Rightarrow (v := a)$, then the reason $R$ is a subset of $X$, $R \subseteq X$, such that $R \wedge c \Rightarrow (v := a)$. Heuristically,

smaller sized reasons are preferred, since that would lead to smaller nogoods resulting in better pruning. We show that when a BDD represents a constraint, the task of finding a *minimum* sized reason is NP-complete. We also show that a *minimal* sized reason can be found in time linear in the size of the BDD.

Given a BDD $b$, an assignment $X$ and $(v := a)$ such that $X \wedge b \Rightarrow (v := a)$, let $R_{all} = \{ R \mid R \subseteq X,\ R \wedge b \Rightarrow (v := a)\}$. The set $R_{all}$ contains all the reasons.

Now, we formally define the problems for finding minimum and minimal reasons in BDDs. We specify the decision version for the minimum problem.
MINIMUM BDD-REASON :
*Input*: A BDD $b$, an assignment $X$, and $(v := a)$, such that $X \wedge b \Rightarrow (v := a)$ and a positive integer $K$.
*Output*: *Yes*, if there is a $R$, such that $R \in R_{all}$, and $|R| \leq K$. *No*, otherwise.
MINIMAL BDD-REASON :
*Input*: A BDD $b$, an assignment $X$, and $(v := a)$, such that $X \wedge b \Rightarrow (v := a)$.
*Output*: $R$, such that $R \in R_{all}$, and $\forall R' \in R_{all}$. *if* $R' \subseteq R$ *then* $R = R'$.

## 3 The Minimum Bdd-Reason is Intractable

We prove that MINIMUM BDD-REASON is NP-complete by using reduction from the HITTING SET problem.
HITTING SET [13]:
*Input*: A collection $Q$ of subsets of a finite set $P$, and a positive integer $K \leq |P|$.
*Output*: *Yes*, if there is a set $P'$ with $|P'| \leq K$ such that $P'$ contains at least one element from each subset in $Q$. *No*, otherwise.

**Lemma 1.** *A relation $r$ with $q$ tuples, defined over $k$ Boolean variables, can be represented by a BDD of size at most $qk$ nodes.*

*Proof.* If the BDD $b$ represents the relation $r$, then there will be exactly $q$ solutions in $b$, one for each tuple in $r$. Since representing each solution by $b$ requires at most $k$ nodes, there will be at most $qk$ non-terminal nodes in $b$. □

**Lemma 2.** *Given a BDD $m$ of a function over the variables in $\{b_1, b_2, \ldots, b_k\}$, using the order $b_1 < b_2 < \ldots < b_k$, if $m \Rightarrow (b_k := \text{false})$ then the size of the BDD $m'$ representing $m \vee (b_k = \text{true})$ is $|m|$.*

*Proof.* Since the variable $b_k$ is at the end of the variable order, given $m$ we can obtain $m'$ by just the following two steps.

1. Add a new node $n'$ with $var(n') = b_k$. The *dashed* edge of $n'$ will reach the 0-terminal and the *solid* edge will reach the 1-terminal. The $n'$ represents the function ($b_k = true$). Now, for each *dashed* (resp. *solid*) edge of the form $(n, 0)$ for any node $n$, where $n \neq n'$, replace the *dashed* (resp. *solid*) edge $(n, 0)$ with a new *dashed* (resp. *solid*) edge $(n, n')$.
2. There will be only one $n''$ such that $var(n'') = b_k$ and $n'' \neq n'$, representing the function ($b_k = false$), otherwise $m \Rightarrow (b_k := false)$ is not possible. Remove $n''$ and make the incoming edges of $n''$ to reach the 1-terminal.

Exactly one node $n'$ is added and one node $n''$ is removed. Hence, $|m'| = |m|$. □

**Theorem 1.** *The MINIMUM BDD-REASON is NP-complete.*

*Proof.* The problem is in NP, as we can quickly check the correctness of any $R$. Now, we reduce the HITTING SET problem into MINIMUM BDD-REASON.

Let the set $P = \{p_1, p_2, \ldots, p_{|P|}\}$, $Q = \{q_1, q_2, \ldots, q_{|Q|}\}$ with $q_i \subseteq P$ and an integer $K$ define an instance of the HITTING SET problem.

| $b_1$ | $b_2$ | . | . | . | $b_{\|P\|}$ | $b_{\|P\|+1}$ |
|---|---|---|---|---|---|---|
| $a_{11}$ | $a_{12}$ | . | . | . | $a_{1\|P\|}$ | *false* |
| $a_{21}$ | $a_{22}$ | . | . | . | $a_{2\|P\|}$ | *false* |
| . | . | . | . | . | . | . |
| . | . | . | . | . | . | . |
| . | . | . | . | . | . | . |
| $a_{\|Q\|1}$ | $a_{\|Q\|2}$ | . | . | . | $a_{\|Q\|\|P\|}$ | *false* |

**Fig. 1.** The relation $r$.

Let $r$ be a relation defined over the $|P| + 1$ Boolean variables in the set $\{b_1, b_2, \ldots, b_{|P|+1}\}$. The Figure 1 shows the structure of the relation $r$. There will be $|Q|$ rows in $r$. The row $i$ of $r$ will correspond to the $q_i \in Q$. Let the Boolean term $a_{ij}$ be *false* iff $p_j \in q_i$.

The row $i$ of the relation $r$ will contain the tuple $(a_{i1}, a_{i2}, \ldots, a_{i|P|}, \mathit{false})$. Let the BDD $b_r$ represents the function of $r$, using the order $b_1 < b_2 < \ldots < b_{|P|+1}$. Let the BDD $b'$ represents the function $(b_{|P|+1} = \mathit{true})$. The $b'$ is trivial with just one non-terminal node. Let the BDD $b$ represents $b_r \vee b'$, i.e., $b = b_r \vee b'$. Let $X = \{b_1 := true, b_2 := true, \ldots, b_{|P|} := true\}$.

By the definition of $r$, in each solution $S$ of $b_r$ the $b_{|P|+1}$ takes *false* value. Also, if $S$ is a solution of $b'$, then $b_{|P|+1}$ takes *true* value in $S$. Due to the different values for $b_{|P|+1}$, the solutions of $b_r$ and $b'$ are disjoint. So for any $S \in b$, either $S \in b_r$ or $S \in b'$, but not both.

For any $q_i \in Q$, $|q_i| \geq 1$, therefore, for each row $i$ of $r$ there exists a $p_j \in q_i$ such that $a_{ij} = \mathit{false}$. So, for any $S \in b$, $S \in b_r$ implies that there exists an $i$, $1 \leq i \leq |P|$, such that $a_{ij} = \mathit{false}$, and hence $b_i$ takes *false* value in $S$. As, for $1 \leq i \leq |P|$, $b_i$ takes *true* in $X$, $X \wedge b_r$ is *false*. So, $X \wedge b = X \wedge b'$ and since $b' = (b_{|P|+1} = \mathit{true})$, $X \wedge b \Rightarrow (b_{|P|+1} := \mathit{true})$.

So, the assignment $X$, the BDD $b$, the variable assignment $(b_{|P|+1} := true)$ and the integer $K$ define an instance of the MINIMUM BDD-REASON problem.

So given a HITTING SET instance $(P, Q, K)$, we can obtain a corresponding instance of MINIMUM BDD-REASON defined by $(X, b, (b_{|P|+1} := true), K)$.

We now have to show that given $(P, Q, K)$, we can obtain $(X, b, (b_{|P|+1} := true), K)$ in polytime and also that the output to $(X, b, (b_{|P|+1} := true), K)$ is *Yes* iff the output for $(P, Q, K)$ is *Yes*.

To show that we can obtain $(X, b, (b_{|P|+1} := true), K)$ in polytime, we just have to show that we can obtain $b$ in polytime. By Lemma 1, $|b_r|$ is bounded by $|Q|(|P| + 1)$. Also, by Lemma 2, $|b|$ which is equivalent to $b_r \vee (b_{|P|+1} = \mathit{true})$ is at most $|b_r|$. Hence, we can obtain $(X, b, (b_{|P|+1} := true), K)$ in polytime.

Now, we just have to show that the instance $(P, Q, K)$ has the *Yes* output iff the instance $(X, b, (b_{|P|+1} := true), K)$ has the *Yes* output.

($\Rightarrow$): Let $P' = \{p_{t_1}, p_{t_2}, \ldots, p_{t_{|P'|}}\}$, where $1 \leq t_i \leq |P|$, be an answer for the *Yes* output of $(P, Q, K)$. Then consider $R$ to be $\{b_{t_1} := \mathit{true}, b_{t_2} := \mathit{true}, \ldots, b_{t_{|P'|}} := \mathit{true}\}$. We show that $R \wedge b \Rightarrow (b_{|P|+1} := \mathit{true})$, which proves the ($\Rightarrow$) case.

Since $P'$ is a *Yes* answer, by definition, for each row $i$ of $r$, there will be a $j$, such that $p_j \in P'$ and $(a_{ij} = \mathit{false})$. So for each row $i$, there will be a $j$, such that $(a_{ij} = \mathit{false})$ and $b_j$ takes *true* value in $R$. Hence, the solution $S \in b_r$ corresponding to any row $i$ cannot be a solution of $R \wedge b_r$. So, $R \wedge b_r = \mathit{false}$, which implies $R \wedge b = R \wedge (b_r \vee b') = ((R \wedge b_r) \vee (R \wedge b')) = ((\mathit{false}) \vee (R \wedge b')) = R \wedge b'$. Since, $(R \wedge b') \Rightarrow (b_{|P|+1} := \mathit{true})$, $R \wedge b \Rightarrow (b_{|P|+1} := \mathit{true})$. Hence the ($\Rightarrow$) case.

($\Leftarrow$): Let $R = \{b_{r_1} := \mathit{true}, b_{r_2} := \mathit{true}, \ldots, b_{r_{|R|}} := \mathit{true}\}$ be a solution for the *Yes* answer of $(X, b, (b_{|P|+1} := \mathit{true}), K)$. Let $P' = \{p_{r_1}, p_{r_2}, \ldots, p_{r_{|R|}}\}$. We have to show that $P'$ has at least one element $p_j \in q_i$ for each $q_i \in Q$.

Since $R \wedge b \Rightarrow (b_{|P|+1} := \mathit{true})$, $b' \Rightarrow (b_{|P|+1} := \mathit{true})$ and $b_r \Rightarrow (b_{|P|+1} := \mathit{false})$, $R \wedge b_r = \mathit{false}$. So, there is no solution $S$ such that $S \in (R \wedge b_r)$.

For each row $i$ of the relation $r$ there exists a $j$ such that $(a_{ij} = \mathit{false})$ and $(b_j := \mathit{true}) \in R$. Otherwise, i.e., if there does not exist such a $j$ for a row $i$ then, the solution $S$ corresponding to the row $i$ belongs to $(R \wedge b_r)$, which is a contradiction to $R \wedge b_r = \mathit{false}$.

So, for each row $i$, there exists a $j$ such that $(a_{ij} = \mathit{false})$ and $(b_j := \mathit{true}) \in R$, hence, $p_j \in q_i$ and $p_j \in P'$, which proves the ($\Leftarrow$) case. $\square$

## 4 A Linear-Time Algorithm for Minimal Bdd-Reason

A dashed edge $(n_1, n_2)$ in a BDD $b$ is a *conflicting edge* with respect to an assignment $X$ if $(\mathit{var}(n_1) := \mathit{true}) \in X$. Similarly, a solid edge $(n_1, n_2)$ in $b$ is a *conflicting edge* with respect to $X$ if $(\mathit{var}(n_1) := \mathit{false}) \in X$.

Suppose $X \wedge b \Rightarrow (v := a)$, then the removal of all the conflicting edges w.r.t $X$ in $b$ will result in removing each solution path $p$ with $(v := \neg a) \in p$. Otherwise, there will be a $p$ such that $X \in p$ and $(v := \neg a) \in p$, which is a contradiction to $X \wedge b \Rightarrow (v := a)$.

**Example 1** Consider the BDD $b$ in the Figure 2 (a) and the assignment $X = \{v := \mathit{true}, x := \mathit{true}, z := \mathit{false}\}$, then $X \wedge b \Rightarrow (y := \mathit{true})$. Hence, the removal of the conflicting edges, as shown in the Figure 2 (b), removes every solution path $p$ with $(y := \mathit{false}) \in p$.

**Example 2** Consider the BDD $b$ in the Figure 3 (a) and the assignment $X = \{v := \mathit{false}, w := \mathit{true}, y := \mathit{false}\}$, then $X \wedge b \Rightarrow (x := \mathit{false})$. Hence, the removal of the conflicting edges, as shown in the Figure 3 (b), removes every solution path $p$ with $(x := \mathit{true}) \in p$.

Suppose $X \wedge b \Rightarrow (v := a)$, a conflicting edge $(n_1, n_2)$ is a *frontier edge* if there exists a solution path $p$ using $(n_1, n_2)$, such that $(v := \neg a) \in p$, and the subpath of $p$ from $n_2$ to the 1-terminal does not use any conflicting edge.

In any solution path $p$ with $(v := \neg a) \in p$, the frontier edge is the conflicting edge nearest to the 1-terminal. Hence, removal of all the frontier edges will result in removing every solution path with $(v := \neg a)$. Otherwise, there will exist a

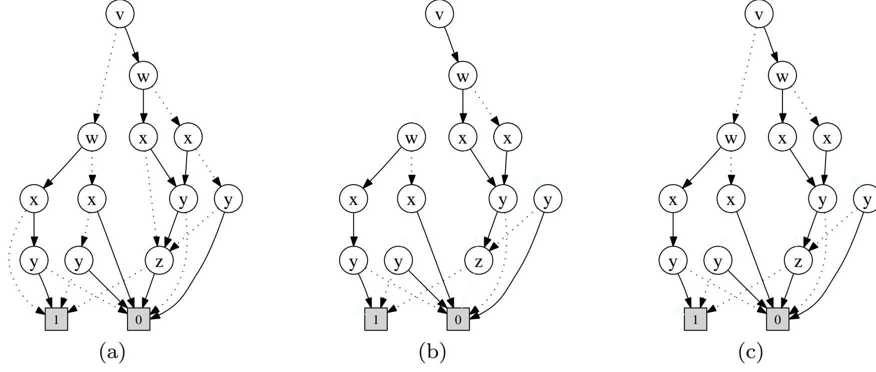


**Fig. 2.** *Example 1*, $X \wedge b \Rightarrow (y := \mathit{true})$. (a) The BDD $b$, (b) The BDD $b$ without the conflicting edges w.r.t $X$, (c) The BDD $b$ without the frontier edges.

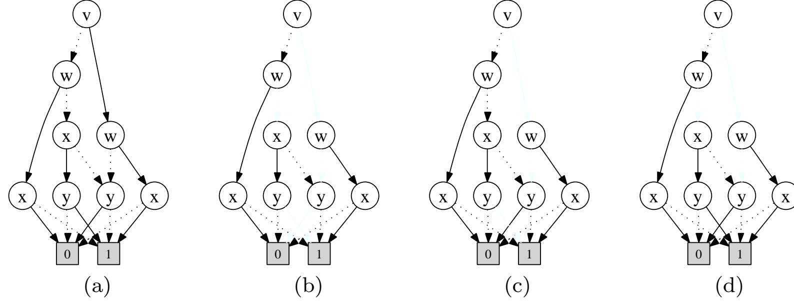


**Fig. 3.** *Example 2*, $X \wedge b \Rightarrow (x := \mathit{false})$. (a) The BDD $b$, (b) The BDD $b$ without the conflicting edges w.r.t $X$, (c) The BDD $b$ without the frontier edges, (d) The BDD $b$ without the conflicting edges w.r.t $R = \{(v := \mathit{false}), (w := \mathit{true})\}$.

solution path $p$ without using any frontier edge, such that $(v := \neg a) \in p$, which is a contradiction to $X \wedge b \Rightarrow (v := a)$. The Figure 2 (c) and Figure 3 (c) show the BDDs of the two examples without just the frontier edges. In both the cases, the removal of the frontier edges removes every solution path $p$ with the corresponding variable assignment.

The idea of our minimal reason algorithm is to first find the frontier edges. Then, to find a subset of the frontier edges such that the inclusion of the variable assignments conflicting the subset in $R$ will ensure that $R \wedge b \Rightarrow (v := a)$ and $R$ is minimal.

In the Example 1, as in the Figure 2 (c), all the frontier edges conflict with just $(x := \mathit{true})$. Hence, the set $R = \{(x := \mathit{true})\}$ is such that $R \wedge b \Rightarrow (y := \mathit{true})$.

In the Example 2, as in the Figure 3 (c), each assignment in $X$ has a frontier edge. There is only one solution path with a frontier edge of $(y := \mathit{false})$. Also, in that path there is a conflicting edge of $(w := \mathit{true})$. Hence, the inclusion of $(w := \mathit{true})$ in $R$ will make the frontier edge of $(y := \mathit{false})$ redundant. So, if $(w := \mathit{true}) \in R$ then $(y := \mathit{false})$ need not belong to $R$. This results in a minimal

reason $R = \{(v =: \mathit{false}), (w := \mathit{true})\}$. The Figure 3 (d) shows the BDD $b$ for the Example 2 after removal of the conflicting edges w.r.t. $R = \{(v := \mathit{false}), (w := \mathit{true})\}$. It can be observed that all the solution paths with $(x := \mathit{true})$ are removed in the figure. Also, the set $R$ is minimal, since for any $R' \subsetneq R$, there exists a solution path $p$ in the BDD $b$, with both $R' \in p$ and $(x := \mathit{true}) \in p$.

The idea of our algorithm is hence to find the frontier edges first. Then to look at the frontier edges, in the order of their variables, and decide on the inclusion of a matching variable assignment in $R$ if it is necessary to remove a solution path.

The Figure 4 presents the *MinimalReason* procedure. The *MinimalReason* uses the *FindFrontier* procedure in Figure 5 to mark the nodes with an outgoing frontier edge. The assumptions made in presenting the procedures are:

1. The BDD $b$ represents a function defined over the $k$ Boolean variables in the set $\{b_1, b_2, \ldots, b_k\}$, using the variable order $b_1 < b_2 < \ldots < b_k$. We assume $X \wedge b \Rightarrow (v := a)$ where $v = b_i$ for an $i$, $1 \leq i \leq k$.
2. The *visited*, *reach1*, and *frontier* are three Boolean arrays, indexed by the nodes in the BDD $b$. The entries in the three arrays are initially *false*.
3. The *reachedSet* is an array of sets indexed by the variables in the BDD. The entries in the *reachedSet* array are initially empty sets.
4. The set $V_X$ denotes the variables in $X$, i.e., $V_X := \{b_i \mid (b_i := a') \in X\}$.

The procedure *FindFrontier* visits all the nodes in the BDD $b$ in a depth first manner and if an edge $(n_1, n_2)$ is a frontier edge, then sets the entry $\mathit{frontier}[n_1]$ to *true*. The procedure uses the *visited* array to make sure it visits a node only once. At the end of the procedure, the entry $\mathit{reach1}[n]$ is *true* iff there exists a path from $n$ to the 1-terminal without using a conflicting edge or an edge corresponding to $(v := a)$.

The lines 1-2 of the *MinimalReason* procedure appropriately initializes the *reach1* and *visited* entries for the two terminal nodes and makes a call to *FindFrontier*. The lines 1-3 of the *FindFrontier* procedure ensure that a node is visited only once and the child nodes are processed first. In the case $(\mathit{var}(n) = v)$ at line-4, based on the value of '$a$' the procedure appropriately sets $\mathit{reach1}[n]$, ignoring the edge corresponding to $(v := a)$. Since we are just interested in removing solution paths with $(v := \neg a)$, we can ignore the edge corresponding to $(v := a)$. In the case $(\mathit{var}(n) \notin V_X)$ at line-9, the procedure sets the $\mathit{reach1}[n]$ to *true* if any of the child nodes of $n$ has *true* entry in *reach1*. The lines 12-13 correspond to the case where $\mathit{var}(n) \in V_X$, in which an outgoing edge of the node $n$ could be a frontier edge. Based on the value $\mathit{var}(n)$ takes in $X$ and the *reach1* entries of the child nodes, the procedure decides whether $\mathit{frontier}[n]$ is *true* or not. Note, the value $\mathit{frontier}[n]$ becomes *true* if an outgoing edge of the node $n$ is a frontier edge.

At the end of the first recursive call made to *FindFrontier* at the line-2 of *MinimalReason*, all the nodes with an outgoing frontier edge are identified by the entries in the *frontier* array. At the line-3 of the *MinimalReason* procedure, the set $\mathit{reachedSet}[\mathit{var}(b)]$ is assigned a set with just the root node. At the end of *MinimalReason*, if a node $n$ belongs to the set $\mathit{reachedSet}[\mathit{var}[n]]$, then it

```
MinimalReason (X, b, (v := a))
1 : reach1[0] := false ; reach1[1] := true ; visited[0] := true ; visited[1] := true
2 : FindFrontier(b)
3 : reachedSet[var(b)] := {b} ; R = { } ; T := {0,1} // T - terminal nodes
4 : for i := 1 to k // i.e., for each variable b_i
5 :       foundFrontier := false
6 :       for each n ∈ reachedSet[b_i]
7 :             if (frontier[n]) foundFrontier := true
8 :       if (foundFrontier)
9 :             if ((b_i := true) ∈ X)
10:                   R.Add((b_i := true))
11:                   for each n ∈ reachedSet[b_i]
12:                         if (solid(n) ∉ T) reachedSet[var(solid(n))].Add(solid(n))
13:             else // i.e., ((b_i := false) ∈ X)
14:                   R.Add((b_i := false))
15:                   for each n ∈ reachedSet[b_i]
16:                         if (dashed(n) ∉ T) reachedSet[var(dashed(n))].Add(dashed(n))
17:       else // i.e., (foundFrontier = false)
18:             for each n ∈ reachedSet[b_i]
19:                   if (solid(n) ∉ T) reachedSet[var(solid(n))].Add(solid(n))
20:                   if (dashed(n) ∉ T) reachedSet[var(dashed(n))].Add(dashed(n))
21: return R
```

**Fig. 4.** The MinimalReason Procedure.

```
FindFrontier (n)
1 : visited[n] := true
2 : if (¬visited[solid(n)]) FindFrontier(solid(n))
3 : if (¬visited[dashed(n)]) FindFrontier(dashed(n))
4 : if (var(n) = v)
5 :       if (a)
6 :             if (reach1[dashed(n)]) reach1[n] := true
7 :       else // i.e., (a = false)
8 :             if (reach1[solid(n)]) reach1[n] := true
9 : else if (var(n) ∉ V_X)
10:       if (reach1[dashed(n)] ∨ reach1[solid(n)]) reach1[n] := true
11: else // i.e., var(n) ∈ V_X
12:       if((var(n) := true) ∈ X)
13:             if (reach1[dashed(n)]) frontier[n] := true
14:             if (reach1[solid(n)]) reach1[n] := true
15:       else
16:             if (reach1[solid(n)]) frontier[n] := true
17:             if (reach1[dashed(n)]) reach1[n] := true
```

**Fig. 5.** The FindFrontier Procedure.

means the node $n$ could be reached from the root node $b$ without using any conflicting edge w.r.t $R$, where $R$ is the output minimal reason. At the line-3 of the procedure, the set $R$ is initialized to be empty and $T$ is initialized to a set

with both the terminal nodes. At the line-4, the procedure starts to loop over each variable in the BDD, in the variable order. During each loop, if any node $n$ belongs to the *reachedSet*[$var(n)$] with (*frontier*[$n$] = *true*), then the procedure adds the assignment of $var(n)$ in $X$ to $R$ and ignores the child node of $n$ which can be reached by the frontier edge of $n$ by not adding it to the *reachedSet*. In the case there was no frontier node in *reachedSet*[$b_i$], then the lines 18-20 adds both the child nodes of each $n \in$ *reachedSet*[$b_i$] to the *reachedSet* if they are not terminal nodes. At the line-21, the procedure returns the obtained minimal reason $R$, such that $R \wedge b \Rightarrow (v := a)$.

**Lemma 3.** *If* $(n_f, n_{f+1})$ *is a frontier edge, then the FindFrontier results in* $frontier[n_f] = true$.

*Proof.* Let a solution path $p$ for which $(n_f, n_{f+1})$ is a frontier edge be $p =< n_1, n_2, \ldots, n_f, n_{f+1}, \ldots, n_l >$, where $n_1 = b$ and $n_l = 1$. We know $(v := \neg a) \in p$.

It can be observed that the *FindFrontier* procedure ensures that, for $f < j \leq l$, *reach1*[$n_j$] = *true*. Since $n_l = 1$, this trivially holds for $n_l$, as initialized at the line-1 of the *MinimalReason* procedure. For $f < j < l$, the edge $(n_j, n_{j+1})$ is not a conflicting edge by frontier edge definition, also $(n_j, n_{j+1})$ does not correspond to the assignment $(v := a)$ as $(v := \neg a) \in p$. Hence, for $f < j < l$, the *FindFrontier* procedure ensures that *reach1*[$n_{j+1}$] $\Rightarrow$ *reach1*[$n_j$]. Therefore, for $f < j \leq l$, *reach1*[$n_j$] = *true*, which implies *reach1*[$n_{f+1}$] = *true*.

Since *reach1*[$n_{f+1}$] = *true* during the call *FindFrontier*($n_f$), the lines 12-17 of the procedure will ensure that *frontier*[$n_f$] = *true*. □

**Theorem 2.** *If MinimalReason* $(X, b, (v := a))$ *returns* $R$ *then* $R \wedge b \Rightarrow (v := a)$.

*Proof.* We show that in any solution path $p$ in the BDD $b$ with $(v := \neg a) \in p$, there exists a conflicting edge w.r.t. $R$. Hence, for any solution $S \in b$, if $(v := \neg a) \in S$, then $S \notin (R \wedge b)$, which proves the theorem.

The proof is by contradiction. Suppose there exists a solution path $p$ in the BDD $b$ with $(v := \neg a) \in p$. Let $p =< n_1, n_2, \ldots, n_f, n_{f+1}, \ldots, n_l >$, where $n_1 = b$, $n_l = 1$ and $(n_f, n_{f+1})$ is the frontier edge. Lets assume the path does not use any conflicting edge w.r.t $R$. Then, we show that $n_f \in$ *reachedSet*[$var(n_f)$] and hence the assignment of $var(n_f)$ in $X$, which conflicts $(n_f, n_{f+1})$, belongs to $R$, which is a contradiction.

Since by assumption the path $p$ does not contain any conflicting edge w.r.t $R$, for any edge $(n_i, n_{i+1})$ in $p$, if the assignment corresponding to the edge is $(var(n_i) := a')$, then $(var(n_i) := \neg a') \notin R$.

Then for $1 \leq i \leq f$, $n_i \in$ *reachedSet*[$var(n_i)$]. This holds trivially for $i = 1$ as initialized at the line-3. For $1 \leq i < f$, since by assumption the edge $(n_i, n_i + 1)$ is not a conflicting edge w.r.t $R$, the procedure would have added $n_{i+1}$ to *reachedSet*[$var(n_{i+1})$], irrespective of the value of the *foundFrontier* flag during the loop at the line-4 for $var(n_i)$. Hence, $n_f \in$ *reachedSet*[$var(n_f)$].

During the loop corresponding to $var(n_f)$, at the line-4 of the *MinimalReason* procedure, since $n_f \in$ *reachedSet*[$var(n_f)$] and by Lemma 3, *frontier*[$n_f$] = *true*,

the *foundFrontier* flag will be *true*. Hence, the assignment to $var(n_f)$ in $X$ will be in $R$, with $(n_f, n_{f+1})$ being a conflicting edge w.r.t. R, which is a contradiction. □

**Theorem 3.** *If MinimalReason* $(X, b, (v := a))$ *returns* $R$ *then* $R$ *is a minimal reason.*

*Proof.* Let $(v' := a') \in R$. The *MinimalReason* includes $(v' := a')$ in $R$ only if there exists a node $n$ with *frontier*$[n] =$ *true*, $var(n) = v'$ and $n \in$ *reachedSet*$[v']$. Hence, by the frontier edge definition, an edge of the form $(n, n')$ is the only conflicting edge w.r.t $R$ in a solution path $p$ with $(v := \neg a) \in p$. Hence, the removal of $(v' := a')$ from $R$ would imply $R \wedge b \Rightarrow (v := a)$ is not true. Therefore, $R$ is minimal. □

**Theorem 4.** *The MinimalReason procedure takes time at most linear in* $|b|$.

*Proof.* The total amount of space used by all the used data-structures is at most linear in $b$. We can ignore the number of variables $k$ when compared with $|b|$, as $|b|$ could be exponential in $k$.

After excluding time taken by the descendant calls, each call to the *FindFrontier* procedure takes constant time. Hence, the call *FindFrontier*$(b)$ in total takes time at most linear in $|b|$.

The running time of *MinimalReason* procedure, excluding the call to *FindFrontier*, is dominated by the loop at line-4. The loop iterates $k$ times. Since a node $n$ in the BDD $b$ is added to *reachedSet*$[var(n)]$ at most once during all the $k$ iterations, the total time required for all the $k$ loops is linear in $b$.

Hence, the *MinimalReason* procedure takes time at most linear in $|b|$ to find a minimal reason. □

## 5 Related Work

A method for finding minimal reasons in BDDs was presented in [7], which we call as the PADL06 method. The authors did not specify the worst case running time of the PADL06 method. But, the PADL06 method uses existential quantification operations on BDDs and hence quite costly when compared to our new linear-time method. If the BDD $b$ is defined over the variables in $V_b$, the PADL06 method existentially quantifies the variables in $(V_b \backslash V_X)$ from the BDD $b$ for finding a minimal reason. Note, the time and space complexity of each existential quantification operation in the worst case could even be *quadratic* [1] in $|b|$. Precisely, some of the advantages of our new method over the PADL06 [7] method are:

1. Worst case linear running time.
2. No costly BDD operations like existential quantifications.
3. No creation of new BDD nodes, the BDDs remain *static* during our solution process. Our new minimal reason method just uses the underlying directed acyclic graph of the BDDs, and hence does not require a full BDD package, while the PADL06 method requires a full BDD package.

In [14], the authors did not give details of their method for generating minimal reasons in BDDs, even the complexity of their method was not mentioned.

## 6 Experiments

We have implemented our new minimal reason algorithm as part of a constraint solver with nogood learning. Our solver uses the BuDDy[1] BDD package. Our solver just uses the lexicographic variable order.

Given a CSP instance in the CLab [15] input format, we use the CLab tool to compile BDDs, one BDD for each constraint in the CSP. This will convert the input CSP instance into a list of Boolean variables and a set of BDDs defined over those variables. Our tool takes the set of BDDs as input and uses our constraint solver to find a solution. Our tool is designed after the BDD-based hybrid SAT solver in [14], which requires a method for MINIMAL BDD-REASON .

We use the 34 CSPs modelling *power supply restoration* problem in our experiments. The instances are available online[2], in the CLab format. All the instances are satisfiable.

We have also implemented the PADL06 [7] method in our tool for comparison. To study the scalability of the PADL06 method and our new method, for each input CSP, we create three types of instances in BDD format with increasing BDD sizes. The first type called *Group*-1 instance, as mentioned above, is obtained by building one BDD for each constraint in the CSP. The second type called *Group*-5 instance is obtained by first partitioning the constraints into $\lceil |C|/5 \rceil$ disjoint groups of constraints in the CSP. Each group will have at most five consecutive constraints, in lexicographic order. Then one BDD will be built to represent the conjunction of the constraints in each group. The third type called *Group*-10 instance is created similar to Group-5, but by using groups of size ten. Since the size of a BDD representing conjunction of five constraints will be usually larger than the sum of the sizes of five BDDs representing each one of the five constraints, the sizes of the BDDs in a Group-5 instance will usually be larger than those in the matching Group-1 instance. Hence, by using Group-1, Group-5 and Group-10 instances of an input CSP, we can study the scalability of the new method and the PADL06 method over increasing BDD sizes.

All our experiments are done in a Cygwin environment with Intel Centrino 1.6 GHz processor and 1 GB RAM.

The conversion of the 34 CSPs into Group-$k$ types, for $k \in \{1, 5, 10\}$, resulted in 102 instances in BDD representation. To compare our new method with the PADL06 method, we used our solver to find a solution for each one of the 104 instances, first using our new method and then using the PADL06 method. We repeated each experiment thrice and obtained the average values. For each instance, we noted the total time taken to find a solution, and the total time taken for the calls made to the corresponding minimal reason method. We used the *gprof* tool to measure the time taken by the minimal reason procedure calls.

[1] http://buddy.sourceforge.net/

[2] http://www.itu.dk/research/cla/externals/clib

**Table 1.** Instance Characteristics. $|V|$: the number of variables in the input CSP. $|V'|$: the number of Boolean variables required to encode the original variables. $|C|$: the number of constraints. **Max**: the size of the largest BDD in the corresponding Group-$k$ instance. **Total**: the sum of the sizes of all the BDDs in the corresponding Group-$k$ instance.

| Instance | | | | BDD Size | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Group-1 | | Group-5 | | Group-10 | |
| Name | $\|V\|$ | $\|V'\|$ | $\|C\|$ | Max | Total | Max | Total | Max | Total |
| *and-break-complex* | 414 | 998 | 852 | 755 | 38808 | 3540 | 110340 | 62735 | 459564 |
| *complex-P1* | 299 | 731 | 592 | 755 | 24055 | 13523 | 77414 | 139048 | 356546 |
| *complex.10* | 414 | 998 | 849 | 631 | 33923 | 4271 | 89448 | 38901 | 262059 |
| *complex.11* | 414 | 998 | 849 | 608 | 32937 | 4371 | 89168 | 40235 | 276547 |
| *complex.12* | 414 | 998 | 849 | 724 | 37902 | 5443 | 108263 | 55494 | 349829 |
| *complex* | 414 | 998 | 849 | 755 | 38804 | 5823 | 112873 | 60903 | 381951 |

**Table 2.** Solution Time (**ST**) and Minimal Reason Time (**MRT**).

| Name | Group-1 | | Group-5 | | Group-10 | |
|---|---|---|---|---|---|---|
| | PADL06 | NEW | PADL06 | NEW | PADL06 | NEW |
| | ST, MRT | ST, MRT | ST, MRT | ST, MRT | ST, MRT | ST, MRT |
| *and-break-complex* | 3.20, 1.03 | 2.94, 0.00 | 13.12, 7.49 | 7.40, 1.21 | 50.54, 41.02 | 14.62, 4.40 |
| *complex-P1* | 1.24, 0.76 | 1.14, 0.03 | 3.88, 2.27 | 2.98, 0.16 | 37.13, 21.52 | 18.17, 2.32 |
| *complex.10* | 5.04, 1.48 | 4.44, 0.01 | 9.19, 5.27 | 5.55, 0.90 | 58.01, 45.10 | 15.96, 4.89 |
| *complex.11* | 5.81, 1.54 | 5.14, 0.01 | 6.47, 3.86 | 3.95, 0.60 | 17.26, 12.81 | 6.73, 1.31 |
| *complex.12* | 2.65, 1.21 | 2.14, 0.04 | 3.15, 2.43 | 2.07, 0.27 | 22.40, 18.10 | 6.96, 1.75 |
| *complex* | 3.19, 1.08 | 2.94, 0.01 | 19.91, 9.94 | 12.29, 1.88 | 227.75, 189.04 | 41.77, 15.20 |

Since we do not have space to list the details for all the 34 instances, we picked five relatively large instances and present their characteristics in Table 1. The Table 2 presents the time taken for finding a solution and the total time taken for finding minimal reasons in both the type of experiments on the five instances.

The Figure 6 and Figure 7 plots the solution time and minimal reason time for the both the minimal reason methods, for all the instances.

The tables and figures show that the new method is at least as fast as the PADL06 method in all the used instances. The new method is even 10 times faster than the PADL06 method. Also, the new method scales better than the PADL06 method as the run-time difference between the new method and the PADL06 method widens from a Group-1 instance to the matching Group-10 instance.

In the case of the *complex* Group-10 instance, the PADL06 method dominates the solution time taking 83% of the solution time, while the usage of the new method reduces the solution time to less than a fifth.

## 7 Conclusion

We have shown that the problem of finding a minimum reason for an implication by a BDD is NP-complete. We have also presented a linear-time algorithm for

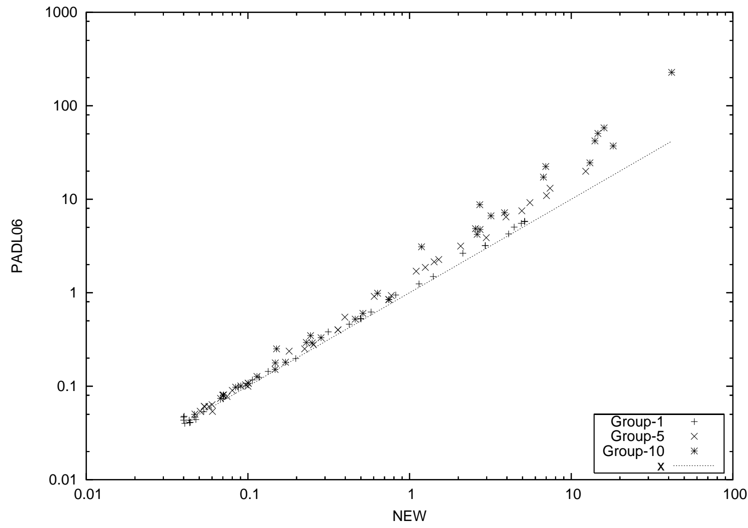


**Fig. 6.** Solution Time.

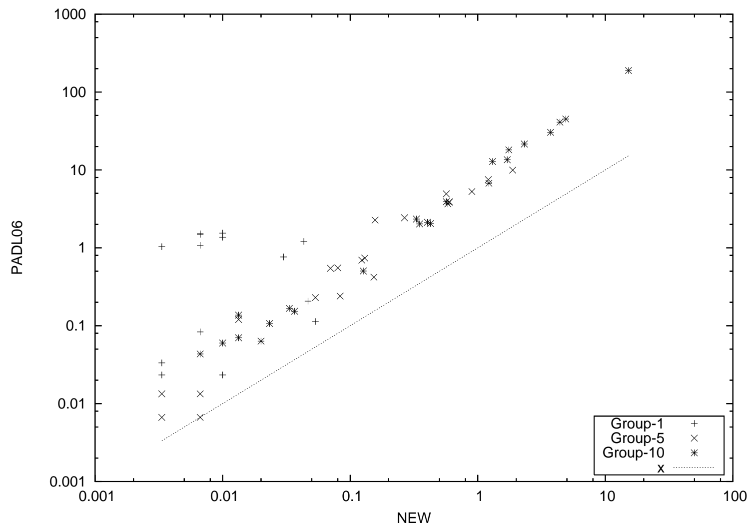


**Fig. 7.** Minimal Reason Time.

finding minimal reasons, which can be used to improve the nogood reasoning process in hybrid constraint solvers using BDDs. Our experiments shows that the new method for finding minimal reasons is better than the previous method for several instances and also scales well due to linear time complexity.

## Acknowledgement

Thanks to Lucas Bordeaux and Youssef Hamadi for discussions related to this work.

## References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Transactions on Computers **8** (1986) 677–691
2. Bouquet, F., Jégou, P.: Using OBDDs to handle dynamic constraints. Information Processing Letters **62** (1997) 111–120
3. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: PETO. (2004) 131–138
4. van der Meer, E.R., Andersen, H.R.: BDD-based recursive and conditional modular interactive product configuration. In: CP 2004 CSPIA Workshop. (2004) 112–126
5. Subbarayan, S., Jensen, R.M., Hadzic, T., Andersen, H.R., Hulgaard, H., Møller, J.: Comparing two implementations of a complete and backtrack-free interactive configurator. In: CP 2004 CSPIA Workshop. (2004) 97–111
6. Lagoon, V., Stuckey, P.: Set domain propagation using ROBDDs. In: CP. (2004) 347–361
7. Hawkins, P., Stuckey, P.J.: A hybrid BDD and SAT finite domain constraint solver. In: PADL. (2006) 103–117
8. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In: ECAI. (2006) 78–82
9. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. Artificial Intelligence **41** (1990) 273–312
10. Schiex, T., Verfaillie, G.: Nogood recording for static and dynamic constraint satisfaction problems. International Journal of Artificial Intelligence Tools **3** (1994) 187–207
11. Katsirelos, G., Bacchus, F.: Unrestricted nogood recording in CSP search. In: CP. (2003) 873–877
12. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in boolean satisfiability solver. In: ICCAD. (2001) 279–285
13. Garey, M.R., Johnson, D.S.: Computers and Intractability-A Guide to the Theory of NP-Completeness. W H Freeman and Co (1979)
14. Damiano, R.F., Kukula, J.H.: Checking satisfiability of a conjunction of BDDs. In: DAC. (2003) 818–823
15. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. In: CP. (2004) 816 `http://www.itu.dk/people/rmj/clab/`.