Last Conflict based Reasoning

Christophe Lecoutre and **Lakhdar Sais** and **Sébastien Tabary** and **Vincent Vidal** [1]

Abstract. In this paper, we propose an approach to guide search to sources of conflicts. The principle is the following: the last variable involved in the last conflict is selected in priority, as long as the constraint network can not be made consistent, in order to find the (most recent) culprit variable, following the current partial instantiation from the leaf to the root of the search tree. In other words, the variable ordering heuristic is violated, until a backtrack to the culprit variable occurs and a singleton consistent value is found. Consequently, this way of reasoning can easily be grafted to many search algorithms and represents an original way to avoid thrashing. Experiments over a wide range of benchmarks demonstrate the effectiveness of this approach.

1 Introduction

For solving instances of the Constraint Satisfaction Problem (CSP), tree search algorithms are commonly used. To limit their combinatorial explosion, various improvements have been proposed. Such improvements mainly concern ordering heuristics, filtering techniques and conflict analysis, and can be conveniently classified as look-ahead and look-back schemes [8]. *Look-ahead* schemes are used when the current partial solution has to be extended and *Look-back* schemes are designed to manage encountered dead-ends.

Early in the 90's, the look-ahead Forward-Checking (FC) algorithm [12] combined with the look-back Conflict-directed Back-Jumping (CBJ) technique [20] was considered as the most efficient approach to solve CSP instances. However, some years later, the look-ahead MAC algorithm, which maintains arc-consistency during search, was shown to be more efficient than FC-CBJ [21, 2].

Then, it became unclear if both paradigms were orthogonal, i.e. counterproductive one to the other, or not. Indeed, while it is confirmed by theoretical results [6] that the more advanced the forward phase is, the less useful the backward phase is, some experiments on hard, structured problems show that adding CBJ to M(G)AC can still present significant improvements. Further, refining the look-back scheme [11, 1, 16] by associating a so-called eliminating explanation (or conflict) with any value rather than with any variable gives to the search algorithm a more powerful backjumping ability. The empirical results in [1, 16] show that MAC can be outperformed by algorithms embedding such look-back techniques.

More recently, look-ahead techniques have taken an advantage with the introduction of conflict-directed variable ordering heuristics [4]. The idea is to associate a weight with any constraint C and to increment the weight of C whenever C is violated during search. As search progresses, the weight of hard constraints become more and more important, and this particularly helps the heuristic to select variables appearing in the hard part of the network. It does respect the fail-first principle : "To succeed, try first where you are most likely to fail" [12]. The new conflict-directed heuristic $dom/wdeg$ is a very simple way to prevent thrashing [4, 13] and is an efficient alternative to backjump-based techniques [18].

Even if an advanced look-ahead technique is used, one can be interested by looking for the reason of an encountered dead-end as finding the ideal ordering of variables is intractable in practice. A dead-end corresponds to a conflict between a subset of decisions (variable assignments) performed so far. In fact, it is relevant (to prevent thrashing) to identify the most recent decision (let us call it the culprit one) that participates to the conflict. Indeed, once the culprit has been identified, we know that it is possible to safely backtrack up to it – this is the role of look-back techniques such as CBJ and DBT (Dynamic Backtracking) [11].

In this paper, we propose an original approach to (indirectly) backtrack to the culprit of the last encountered dead-end. To achieve it, the leaf conflict variable becomes in priority the next variable to be selected as long as the successive assignments that involves it render the network arc inconsistent. It then corresponds to checking the singleton consistency of this variable from the leaf toward the root of the search tree until a singleton value is found. In other words, the variable ordering heuristic is violated, until a backtrack to the culprit variable occurs and a singleton value is found.

In summary, the approach that we propose aims at guiding search to dynamically detect the conflict reason of the last encountered dead-end. It is important to remark that, contrary to sophisticated backjump techniques, our approach can be grafted in a very simple way to a tree search algorithm without any additional data structure.

2 Technical background

A Constraint Network (CN) P is a pair $(\mathscr{X},\mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints. Each variable $X \in \mathscr{X}$ has an associated domain, denoted $dom(X)$, which contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves a subset of variables of $\mathscr{X}$, called scope and denoted $vars(C)$, and has an associated relation, denoted $rel(C)$, which contains the set of tuples allowed for its variables. A solution to a CN is an assignment of values to all the variables such that all the constraints are satisfied. A CN is said to be satisfiable iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-complete task of determining whether a given CN is satisfiable. A CSP instance is then defined by a CN, and solving it involves either finding one (or more) solution or determining its unsatisfiability. To solve a CSP instance, one can modify the CN by using inference or search methods [8]. Usually, domains of variables are reduced by removing inconsistent values, i.e. values that cannot occur in any solution. Indeed, it is possible to filter domains by considering some properties of constraint networks. Arc Consistency (AC) remains the central one.

[1] CRIL-CNRS FRE 2499, rue de l'université, SP 16, 62307 Lens cedex, France. email: {lecoutre,sais,tabary,vidal}@cril.univ-artois.fr

Definition 1 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN. A pair (X, v), with $X \in \mathscr{X}$ and $v \in dom(X)$, is arc consistent iff $\forall C \in \mathscr{C} \mid X \in vars(C)$, there exists a support of (X, v) in C, i.e., a tuple $t \in rel(C)$ such that $t[X] = v$ and $t[Y] \in dom(Y)$ $\forall Y \in vars(C)$. P is arc consistent iff $\forall X \in \mathscr{X}$, dom(X) $\neq \emptyset$ and $\forall v \in dom(X)$, (X, v) is arc consistent.*

Singleton Arc Consistency (SAC) [7] is a stronger consistency than AC, i.e. SAC can identify more inconsistent values than AC can. SAC guarantees that enforcing arc consistency after performing any variable assignment does not show unsatisfiability, i.e., does not entail a domain wipe-out.

Backtracking Search Algorithms The backtracking algorithm (BT) is a central algorithm for solving CSP instances. It employs a depth-first search in order to instantiate variables and a backtrack mechanism when dead-ends occur. Many works have been devoted to improve its forward and backward phases by introducing look-ahead and look-back schemes [8].

MAC [21] is the look-ahead algorithm which is considered as the most efficient generic approach to solve CSP instances. It simply maintains (re-establishes) arc consistency after each variable assignment. A dead-end is encountered if the network becomes arc inconsistent (because a domain is wiped out, i.e. becomes empty). When mentioning MAC, it is important to indicate which branching scheme is employed. Indeed, it is possible to consider binary (2-way) branching or non binary (d-way) branching. These two schemes are not equivalent as it has been shown that binary branching is more powerful (to refute unsatisfiable instances) than non-binary branching [14]. With binary branching, at each step of the search, a pair (X,v) is selected where X is an unassigned variable and v a value in dom(X), and two cases are considered: the assignment $X = v$ and the refutation $X \neq v$. The MAC algorithm (using binary branching) can then be seen as building a binary tree. During search (i.e. when the tree is being built), we can make the difference between an opened node, for which only one case has been considered, and a closed node, for which both cases have been considered. Classically, MAC always starts by assigning variables before refuting values.

On the other hand, three representative look-back algorithms are SBT (Standard Backtracking), CBJ (Conflict Directed Backjumping) [20] and DBT (Dynamic Backtracking) [11]. The principle of these look-back algorithms is to jump back to a variable assignment that must be reconsidered as it is suspected to be the most recent culprit of the dead-end. SBT simply backtracks to to the previously assigned variable whereas CBJ and DBT can identify a meaningful culprit decision by exploiting eliminating explanations.

Search Heuristics The order in which variables are assigned by a backtracking search algorithm has been recognized as a key issue for a long time. Using different variable ordering heuristics to solve a CSP instance can lead to drastically different results in terms of efficiency. In this paper, we will focus on the following representative variable ordering heuristics: dom, bz, $dom/ddeg$ and $dom/wdeg$. The well-known dynamic heuristic dom [12] selects, at each step of the search, one variable with the smallest domain size. To break ties (which correspond to sets of variables that are considered as equivalent by the heuristic), one can use the current (also called dynamic) degree of the variable. This is the heuristic called bz [5]. By directly combining domain sizes and dynamic variable degrees, one obtains $dom/ddeg$ [2] which can substantially improve the performance of the search on some problems. Finally, in [4], the heuristic $dom/wdeg$ has been introduced. The principle is to associate with any constraint of the problem a counter which is incremented whenever the constraint is involved in a dead-end. Hence, $wdeg$ that refers to the weighted degree of a variable corresponds to the sum of the weights of the constraints involving this variable (and at least another unassigned variable).

3 Reasoning from conflicts

In this section, we show that it is possible to identify a nogood from a sequence of decisions leading to a conflict, and to exploit this nogood during search. We then discuss the impact of such a reasoning on thrashing prevention.

3.1 Nogood identification

Definition 2 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN and (X,v) be a pair such that $X \in \mathscr{X}$ and $v \in dom(X)$. The assignment $X = v$ is called a positive decision whereas the refutation $X \neq v$ is called a negative decision. $\neg(X = v)$ denotes $X \neq v$ and $\neg(X \neq v)$ denotes $X = v$.*

¿From now on, we will consider an inference operator ϕ assumed to satisfy some usual properties. This operator can be employed at any step of a tree search, denoted ϕ-search, using a binary branching scheme. For example, MAC corresponds to the ϕ_{AC}-search algorithm where ϕ_{AC} is the operator that establishes AC.

Definition 3 *Let P be a CN and Δ be a set of decisions. $P|_\Delta$ is the CN obtained from P such that, for any positive decision ($X = v$) $\in \Delta$, dom(X) is restricted to $\{v\}$, and, for any negative decision ($X \neq v$) $\in \Delta$, v is removed from dom(X). $\phi(P)$ is the CN obtained after applying the inference operator ϕ on P.*

If there exists a variable with an empty domain in $\phi(P)$ then P is clearly unsatisfiable, denoted $\phi(P) = \bot$.

Definition 4 *Let P be a CN and Δ be a set of decisions. Δ is a nogood of P iff $P|_\Delta$ is unsatisfiable. Δ is a ϕ-nogood of P iff $\phi(P|_\Delta) = \bot$. Δ is a minimal ϕ-nogood of P iff $\nexists \Delta' \subset \Delta$ such that $\phi(P|_{\Delta'}) = \bot$.*

Obviously, ϕ-nogoods are nogoods, but the opposite is not necessarily true. Our definition includes both positive and negative decisions as in [9, 17]. Note that we can consider a ϕ-nogood Δ as deduced from a sequence of decisions $\langle \delta_1, \ldots, \delta_i \rangle$ such that $\Delta = \{\delta_1, \ldots, \delta_i\}$. Such a sequence can correspond to the decisions taken along a branch in a search tree leading to a dead-end.

Definition 5 *Let P be a CN and $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be a sequence of decisions such that $\{\delta_1, \ldots, \delta_i\}$ is a ϕ-nogood of P. A decision $\delta_j \in \Sigma$ is said to be culprit iff $\exists v \in dom(X_i) \mid \phi(P|_{\{\delta_1, \ldots, \delta_{j-1}, \neg\delta_j, X_i = v\}}) \neq \bot$ where X_i is the variable involved in δ_i. We define the culprit subsequence of Σ to be either the empty sequence if no culprit decision exists, or the sequence $\langle \delta_1, \ldots, \delta_j \rangle$ where δ_j is the latest culprit decision in Σ.*

In other words, the culprit subsequence of a sequence of decisions Σ leading to an inconsistency ends in the most recent decision such that, when negated, there exists a value that can be assigned, without yielding an inconsistency with ϕ, to the variable involved in the last decision of Σ. We can show that it corresponds to a nogood.

Proposition 1 *Let P be a CN and $\Sigma = \langle\delta_1, \ldots, \delta_i\rangle$ be a sequence of decisions s.t. $\{\delta_1, \ldots, \delta_i\}$ is a ϕ-nogood of P. The set of decisions contained in the culprit subsequence of Σ is a nogood of P.*

Proof. Let $\langle\delta_1, \ldots, \delta_j\rangle$ be the culprit subsequence of Σ. Let us demonstrate by recurrence that for any integer k such that $j \leq k \leq i$, the following hypothesis, denoted H(k), holds:

H(k): $\{\delta_1, \ldots, \delta_k\}$ is a nogood

First, let us show that H(i) holds. We know that $\{\delta_1, \ldots, \delta_i\}$ is a nogood since, by hypothesis, $\{\delta_1, \ldots, \delta_i\}$ is a ϕ-nogood of P. Then, let us show that, for $j < k \leq i$, if H(k) holds then H($k-1$) also holds. As $k > j$ and H(k) holds, we know that, by recurrence hypothesis, $\{\delta_1, \ldots, \delta_{k-1}, \delta_k\}$ is a nogood. Furthermore, δ_k is not a culprit variable (since $k > j$). Hence, by Definition 5, we know that $\forall v \in dom(X_i)$, $\phi(P|_{\{\delta_1,\ldots,\delta_{k-1},\neg\delta_k,X_i=v\}}) = \bot$. As a result, the set $\{\delta_1, \ldots, \delta_{k-1}, \neg\delta_k\}$ is a nogood. By resolution, $\{\delta_1, \ldots, \delta_{k-1}\}$ is a nogood. □

The following property states that identifying an empty culprit subsequence allows proving the unsatisfiability of a CN.

Corollary 1 *Let P be a CN and $\Sigma = \langle\delta_1, \ldots, \delta_i\rangle$ be a sequence of decisions such that $\{\delta_1, \ldots, \delta_i\}$ is a ϕ-nogood of P. If the culprit subsequence of Σ is empty, then P is unsatisfiable.*

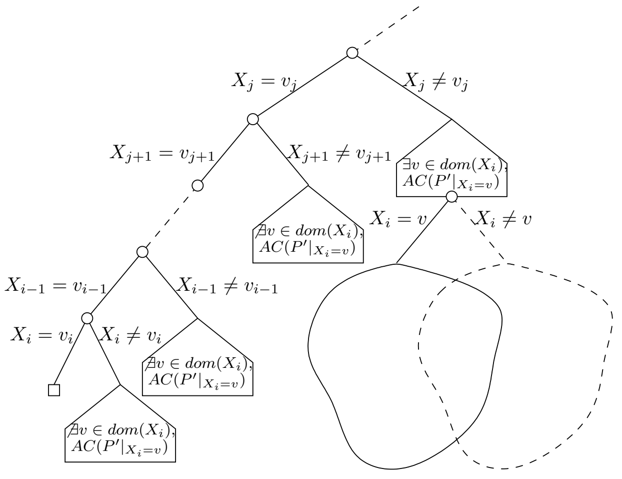
When we obtain a ϕ-nogood from a sequence of decisions Σ taken along a branch built by a ϕ-search algorithm, it is safe to backjump to the last decision contained in the culprit subsequence of Σ which corresponds to a nogood. We can also remark that the set of decisions contained in a culprit subsequence may not be a minimal nogood, and can be related to the nogood computed using the first Unique Implication Point (1UIP) scheme used in SAT solvers [22].

3.2 Last Conflict reasoning

The identification and exploitation of nogoods as described above can be easily embedded into a ϕ-search algorithm thanks to a simple modification of the variable ordering heuristic. We will call this approach *Last Conflict reasoning* (LC).

In practice, we will exploit LC only when a dead-end has been reached from an opened node of the search tree, that is to say, from a positive decision since when a binary branching scheme is used, positive decisions are generally taken first. It means that LC will be used iff δ_i (the last decision of the sequence mentioned in Definition 5) is a positive decision. To implement LC, it is then sufficient to (i) register the variable whose assignment to a given value directly leads to an inconsistency, and (ii) always prefer this variable in subsequent decisions (if it is still unassigned) over the choice given by the underlying heuristic – whatever heuristic is used. Notice that LC does not require any additional space cost.

Figure 1 illustrates *Last Conflict* reasoning. The leftmost branch on the figure corresponds to the positive decisions $X_1 = v_1$, ..., $X_i = v_i$, such that $X_i = v_i$ leads to a conflict. At this point, X_i is registered by LC for future use. As a consequence, v_i is removed from $dom(X_i)$, i.e. $X_i \neq v_i$. Then, instead of pursuing the search with a new selected variable, X_i is chosen to be assigned with a new value v'. In our illustration, this leads once again to a conflict, v' is removed from $dom(X_i)$, and the process loops until all values are removed from $dom(X_i)$, leading to a domain wipe-out. The algorithm then backtracks to the assignment $X_{i-1} = v_{i-1}$, going to the right branch $X_{i-1} \neq v_{i-1}$. As X_i is still recorded by LC, it is then selected in priority, and all values of $dom(X_i)$ are excluded due to



$X_j = v_j$
$X_j \neq v_j$
$X_{j+1} = v_{j+1}$
$X_{j+1} \neq v_{j+1}$
$\exists v \in dom(X_i), AC(P'|_{X_i=v})$
$\nexists v \in dom(X_i), AC(P'|_{X_i=v})$
$X_i = v$
$X_i \neq v$
$X_{i-1} = v_{i-1}$
$X_{i-1} \neq v_{i-1}$
$\nexists v \in dom(X_i), AC(P'|_{X_i=v})$
$X_i = v_i$
$X_i \neq v_i$
$\nexists v \in dom(X_i), AC(P'|_{X_i=v})$

Figure 1. Last Conflict reasoning illustrated in a partial binary branching search tree. P' denotes the constraint network obtained at each node after performing the previous decisions and applying an inference operator ϕ.

the same process as above. The algorithm finally backtracks to the decision $X_j = v_j$, going to the right branch $X_j \neq v_j$. Then, as X_i is still the registered variable, it is preferred again and the values of $dom(X_i)$ are tested. But, as one of them (v) does not lead to a conflict, the search can continue with the assignment $X_i = v$. The variable X_i is then unregistered, and the choice for subsequent decisions is left to the underlying heuristic, until the next conflict occurs.

By using the ϕ_{AC} operator to identify culprit subsequences as described above, we obtain the following complexity results.

Proposition 2 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN and $\Sigma = \langle\delta_1, \ldots, \delta_i\rangle$ be a sequence of decisions s.t. $\{\delta_1, \ldots, \delta_i\}$ is a ϕ_{AC}-nogood of P. The worst case time complexity of computing the culprit subsequence of Σ is $O(eid^3)$, where $e = |\mathscr{C}|$ and d is the greatest domain size of P.*

Proof. The worst case is when the computed culprit subsequence of Σ is empty. In this case, it means that, for each decision, we check the singleton arc consistency of the variable involved in δ_i. As checking the singleton arc consistency of a variable corresponds to at most d calls to an arc consistency algorithm, the worst case time complexity is id times the complexity of the used arc consistency algorithm. As the optimal worst case time complexity of establishing arc consistency is $O(ed^2)$ (e.g. AC2001/3.1 [3]), we obtain the overall time complexity $O(eid^3)$.□

Corollary 2 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN and $\Sigma = \langle\delta_1, \ldots, \delta_i\rangle$ be a sequence of decisions that corresponds to a branch built by MAC leading to a failure. The worst case time complexity of computing the culprit subsequence of Σ is $O(end^3)$, where $n = |\mathscr{X}|$, $e = |\mathscr{C}|$ and d is the greatest domain size of P.*

Proof. First, we know, that as positive decisions are performed first by MAC, the number of opened nodes in a branch of the search tree is at most n. Second, for each closed node, we do not have to check the singleton arc consistency of the variable involved in δ_i since we have to directly backtrack. So we obtain $O(end^3)$. □

3.3 Preventing thrashing using LC

Thrashing is the fact of repeatedly exploring the same subtrees. This phenomenon deserves to be carefully studied as an algorithm subject to thrashing can be very inefficient.

Sometimes, thrashing can be explained by some bad choices made earlier during search. In fact, whenever a value is removed from the domain of a variable, it can be explained (with more or less precision). It simply means that it is possible to indicate the decisions (i.e. variable assignments in our case) that entailed removing this value. By recording such so-called eliminating explanations and exploiting this information, one can hope to backtrack up to a level where a culprit variable will be re-assigned, this way, avoiding thrashing.

In some cases, no pertinent culprit variable(s) can be identified by a backjumping technique although thrashing occurs. For example, let us consider some unsatisfiable instances of the queens+knights problem as proposed in [4]. When the two subproblems are merged without any interaction (there is no constraint involving both a queen variable and a knight variable as in the qk-25-25-5-add instance), a backjumping technique such as CBJ or DBT is able to prove the unsatisfiability of the problem from the unsatisfiability of the knights subproblem (by backtracking up to the root of the search tree). When the two subproblems are merged with an interaction (queens and knights cannot be put on the same square as in the qk-25-25-5-mul instance), CBJ and DBT become subject to thrashing (when they are used with a standard variable ordering heuristic such as dom, bz and $dom/ddeg$) because the last assigned queen variable is considered as participating to the failure. The problem is that, even if there exists different eliminating explanations for a removed value, only the first one is recorded. One can still imagine to improve existing backjumping algorithms by updating eliminating explanations or computing new ones [15]. However, it is far beyond the scope of this paper.

Last Conflict reasoning is a new way of preventing thrashing, while still being a look-ahead technique. Indeed, guiding search to the last decision of a culprit subsequence behaves similarly to using a form of backjumping to that decision. For example, we can show that when a backjump to a culprit decision occurs with the Gaschnig's technique [10], then LC, in the same context, also reaches this decision in polynomial time.

Table 1 illustrates the powerful thrashing prevention capability of LC on the two instances mentioned above. SBT, CBJ and DBT cannot prevent thrashing for the qk-25-25-5-mul instance as, within 2 hours, the instance remains unsolved (even when other standard heuristics are used). On the other hand, in about 1 minute, LC (with SBT) can prove the unsatisfiability of this instance. The reason is that all knight variables are singleton arc inconsistent. When such a variable is reached, LC guides search up to the root of the search tree.

$Instance$		SBT	CBJ	DBT	LC
qk-25-25-5-add	cpu	$> 2h$	11.7	12.5	58.9
	$nodes$	$-$	703	691	$10,053$
qk-25-25-5-mul	cpu	$> 2h$	$> 2h$	$> 2h$	66.6
	$nodes$	$-$	$-$	$-$	9922

Table 1. Cost of running MAC-bz (time out set to 2 hours)

	$dom/ddeg$		$dom/wdeg$		bz	
	$\neg LC$	LC	$\neg LC$	LC	$\neg LC$	LC
Composed random instances (10 instances per series)						
25-1-40	3600 (10)	0.03	0.05	0.03	0.01	0.01
25-10-20	1789 (4)	0.32	0.09	0.08	1255 (3)	0.10
75-1-40	3600 (10)	0.10	0.10	0.90	0.02	0.02
Random 3-SAT instances (100 instances per series)						
ehi-85	1726	2.43	2.21	0.43	1236	1.34
ehi-90	3919	3.17	2.34	0.43	2440	1.37
Balanced QWH instances (100 instances per series)						
15-106	3.72	2.6	0.27	0.35	3.8	2.9
18-141	528 (4)	267 (1)	4.96	6.87	542 (4)	274 (1)
Radar Surveillance instances (50 instances per series)						
rs-24-2	948 (13)	0.28	0.01	0.01	1989 (26)	0.52
rs-30-0	242 (3)	0.35	0.01	0.01	1108 (15)	18

Table 2. Average cost (cpu) of running MAC without LC ($\neg LC$) and with LC on different series of problems

4 Experiments

In order to show the practical interest of the approach described in this paper, we have conducted an extensive experimentation (on a PC Pentium IV 2,4GHz 512Mo under Linux). Performances are measured in terms of the number of visited nodes (nodes) and the cpu time in seconds (cpu). Remark that for our experimentation, we have used MAC (with SBT, i.e. chronological backtracking), and studied the impact of LC wrt various variable ordering heuristics.

First, we have experimented different series of problems. The results that we have obtained are given in Table 2. The series corresponding to composed random instances, random 3-SAT instances and balanced QWH instances were used as benchmarks[2] for the first CSP solver competition. Each composed random instance contains a main (under-constrained, here) fragment and some auxiliary fragments, each one being grafted to the main fragment by introducing some binary constraints. Each random 3-SAT instance embeds a small unsatisfiable part and has been converted to CSP using the dual encoding method. Each balanced QWH instance corresponds to a satisfiable balanced quasi-group instance with holes. Finally, the last series correspond to some realistic radar surveillance instances[3] as proposed by the Swedish Institute of Computer Science (SICS). The problem is to adjust the signal strength (from 0 to 3) of a given number of fixed radars wrt 6 geographic sectors. Moreover, each cell of the geographic area must be covered exactly by 3 radar stations, except for some insignificant cells that must not be covered. Each set is denoted rs-i-j where i and j represent the number of radars and the number of insignificant cells, respectively.

In Table 2, we can observe the impact of LC when using MAC and different heuristics. Note that the time limit was fixed to 1 hour (except for the random 3-SAT instances, all solved in reasonable time) and that the number of expired instances, i.e. instances not solved within 1 hour of allowed cpu time, is given between brackets. Also, remark that, in case of expired instances, the indicated cpu must be considered as a lower bound.

On the one hand, when standard heuristics $dom/ddeg$ and bz are used, it clearly appears that LC allows improving the efficiency of MAC in both CPU time and the number of solved instances, especially on composed and radar surveillance instances. In fact, these instances have a structure and a too blind search is subject to thrashing. Using LC avoids this phenomena without disturbing the main behaviour of the heuristics. On the other hand, when the conflict-directed heuristic $dom/wdeg$ is used, LC is not so important since thrashing was already prevented by the heuristic.

Finally, we present in Table 3 some representative results obtained for some selected instances of the first international CSP competition. The time limit was also fixed to 1 hour. Once again, it appears that using LC with a standard heuristic greatly improves the efficiency of MAC. This is not always the case for MAC-$dom/wdeg$. Note that most of these instances cannot be efficiently solved using a backjumping technique such as CBJ or DBT combined with a standard heuristic as shown in [18].

[2] `http://cpai.ucc.ie/05/Benchmarks.html`
[3] `http://www.ps.uni-sb.de/~walser/radar/radar.html`

		$dom/ddeg$		$dom/wdeg$		dom		bz	
		$\neg LC$	LC	$\neg LC$	LC	$\neg LC$	LC	$\neg LC$	LC
Academic instances									
$Golomb$-11-sat	cpu	438	146	584	149	379	134	439	147
	$nodes$	55,864	19,204	51,055	12,841	58,993	21,858	60,352	21,262
$BlackHole$-4-4-0010	cpu	3.89	3.60	3.01	5.26	$> 1h$	36.45	$> 1h$	28.50
	$nodes$	6,141	5,573	6,293	9,166	–	$162K$	–	$106K$
cc-10-10-2	cpu	1238	35.63	3.10	4.11	99.95	8.45	1239	35.50
	$nodes$	$543K$	14,656	2,983	3,526	$180K$	9,693	$543K$	14,656
qcp-20-$balanced$−23	cpu	$> 1h$	201	17.11	1.00	26.70	2.62	1.11	3.39
	$nodes$	–	$141K$	19,835	1,210	$100K$	6,606	431	3,470
qk-25-25-5-add	cpu	$> 1h$	57.30	135	63.64	$> 1h$	57.72	$> 1h$	57.35
	$nodes$	–	10,052	24,502	11,310	–	10,053	–	10,053
qk-25-25-5-mul	cpu	$> 1h$	66.34	134	68.31	$> 1h$	65.64	$> 1h$	66.23
	$nodes$	–	9,922	22,598	9,908	–	9,922	–	9,922
Real-world instances									
$e0ddr1$-10	cpu	$> 1h$	87.47	18.85	30.49	102	1.06	$> 1h$	52.11
	$nodes$	–	$157K$	37,515	56,412	$307K$	1,390	–	94,213
$enddr1$-1	cpu	$> 1h$	1.35	0.72	0.57	0.20	0.20	$> 1h$	0.78
	$nodes$	–	2,269	1,239	733	50	50	–	1,117
$graph2$-$f25$	cpu	$> 1h$	77.86	29.10	3.53	$> 1h$	344	$> 1h$	72.23
	$nodes$	–	54,255	31,492	3,140	–	$436K$	–	51,246
$graph8$-$f11$	cpu	$> 1h$	5.00	7.54	0.49	$> 1h$	57.73	$> 1h$	49.58
	$nodes$	–	1,893	5,075	152	–	41,646	–	22,424
$scen11$	cpu	95.84	1.65	0.97	0.96	$> 1h$	1104	$> 1h$	2942
	$nodes$	31,81	905	911	936	–	$804K$	–	$1672K$
$scen6$-$w2$	cpu	$> 1h$	0.45	0.51	0.29	$> 1h$	0.51	$> 1h$	0.46
	$nodes$	–	405	741	272	–	706	–	318

Table 3. Cost of running MAC without LC ($\neg LC$) and with LC on academic and real-world instances

5 Conclusion

In this paper, we have introduced an original approach that can be grafted to any search algorithm based on a depth-first exploration. The principle is to select in priority the variable involved in the last conflict (i.e. the last assignment that failed) as long as the network cannot be made consistent. This way of reasoning allows preventing thrashing by backtracking to the most recent culprit of the last conflict. It can be done without any additional cost in space and with a worst-case time complexity in O(end^3). Our extensive experimentation clearly shows the interest of this approach.

In our approach, the variable ordering heuristic is violated, until a backtrack to the culprit variable occurs and a singleton consistent value is found. However, there is an alternative which is to not consider the found singleton consistent value as the next value to be assigned. In this case, the approach becomes a pure inference technique which corresponds to (partially) maintaining a singleton consistency (SAC, for example) on the variable involved in the last conflict. This would be related to the recent "quick shaving" technique [19].

Acknowledgments

This paper has been supported by the CNRS, the "Planevo" project and the "IUT de Lens".

REFERENCES

[1] F. Bacchus, 'Extending Forward Checking', in *Proceedings of CP'00*, pp. 35–51, (2000).

[2] C. Bessière and J. Régin, 'MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems', in *Proceedings of CP'96*, pp. 61–75, (1996).

[3] C. Bessière, J.C. Régin, R.H.C. Yap, and Y. Zhang, 'An optimal coarse-grained arc consistency algorithm', *Artificial Intelligence*, **165**(2), 165–185, (2005).

[4] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, 'Boosting systematic search by weighting constraints', in *Proceedings of ECAI'04*, pp. 146–150, (2004).

[5] D. Brelaz, 'New methods to color the vertices of a graph', *Communications of the ACM*, **22**, 251–256, (1979).

[6] X. Chen and P. van Beek, 'Conflict-directed backjumping revisited', *Journal of Artificial Intelligence Research*, **14**, 53–81, (2001).

[7] R. Debruyne and C. Bessière, 'Some practical filtering techniques for the constraint satisfaction problem', in *Proceedings of IJCAI'97*, pp. 412–417, (1997).

[8] R. Dechter, *Constraint processing*, Morgan Kaufmann, 2003.

[9] F. Focacci and M. Milano, 'Global cut framework for removing symmetries', in *Proceedings of CP'01*, pp. 77–92, (2001).

[10] J. Gaschnig, 'Performance measurement and analysis of search algorithms.', Technical Report CMU-CS-79-124, Carnegie Mellon, (1979).

[11] M. Ginsberg, 'Dynamic backtracking', *Artificial Intelligence*, **1**, 25–46, (1993).

[12] R.M. Haralick and G.L. Elliott, 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, **14**, 263–313, (1980).

[13] T. Hulubei and B. O'Sullivan, 'Search heuristics and heavy-tailed behaviour', in *Proceedings of CP'05*, pp. 328–342, (2005).

[14] J. Hwang and D.G. Mitchell, '2-way vs d-way branching for CSP', in *Proceedings of CP'05*, pp. 343–357, (2005).

[15] U. Junker, 'QuickXplain: preferred explanations and relaxations for over-constrained problems', in *Proceedings of AAAI'04*, pp. 167–172, (2004).

[16] N. Jussien, R. Debruyne, and P. Boizumault, 'Maintaining arc-consistency within dynamic backtracking', in *Proceedings of CP'00*, pp. 249–261, (2000).

[17] G. Katsirelos and F. Bacchus, 'Generalized nogoods in csps', in *Proceedings of AAAI'05*, pp. 390–396, (2005).

[18] C. Lecoutre, F. Boussemart, and F. Hemery, 'Backjump-based techniques versus conflict-directed heuristics', in *Proceedings of ICTAI'04*, pp. 549–557, (2004).

[19] O. Lhomme, 'Quick shaving', in *Proceedings of AAAI'05*, pp. 411–415, (2005).

[20] P. Prosser, 'Hybrid algorithms for the constraint satisfaction problems', *Computational Intelligence*, **9**(3), 268–299, (1993).

[21] D. Sabin and E. Freuder, 'Contradicting conventional wisdom in constraint satisfaction', in *Proceedings of CP'94*, pp. 10–20, (1994).

[22] L. Zhang, C.F. Madigan, M.W. Moskewicz, and S. Malik, 'Efficient conflict driven learning in a Boolean satisfiability solver', in *Proceedings of ICCAD'01*, pp. 279–285, (2001).