Artificial Intelligence 173 (2009) 1592–1614

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Reasoning from last conflict(s) in constraint programming

Christophe Lecoutre [a,b,*], Lakhdar Saïs [a,b], Sébastien Tabary [a,b], Vincent Vidal [c]

[a] CRIL-CNRS UMR 8188, Université Lille-Nord de France
[b] Artois, IUT de Lens, rue de l'université, SP 16, F-62307 Lens, France
[c] ONERA-DCSD 2, avenue Édouard Belin, BP 4025, F-31055 Toulouse Cedex 4, France

ARTICLE INFO

Article history:
Received 26 February 2008
Received in revised form 8 September 2009
Accepted 15 September 2009
Available online 18 September 2009

Keywords:
Constraint satisfaction
Conflicts
Nogoods
Intelligent backtracking
Planning

ABSTRACT

Constraint programming is a popular paradigm to deal with combinatorial problems in artificial intelligence. Backtracking algorithms, applied to constraint networks, are commonly used but suffer from *thrashing*, i.e. the fact of repeatedly exploring similar subtrees during search. An extensive literature has been devoted to prevent thrashing, often classified into *look-ahead* (constraint propagation and search heuristics) and *look-back* (intelligent backtracking and learning) approaches. In this paper, we present an original look-ahead approach that allows to guide backtrack search toward sources of conflicts and, as a side effect, to obtain a behavior similar to a backjumping technique. The principle is the following: after each conflict, the last assigned variable is selected in priority, so long as the constraint network cannot be made consistent. This allows us to find, following the current partial instantiation from the leaf to the root of the search tree, the culprit decision that prevents the last variable from being assigned. This way of reasoning can easily be grafted to many variations of backtracking algorithms and represents an original mechanism to reduce thrashing. Moreover, we show that this approach can be generalized so as to collect a (small) set of incompatible variables that are together responsible for the last conflict. Experiments over a wide range of benchmarks demonstrate the effectiveness of this approach in both constraint satisfaction and automated artificial intelligence planning.

1. Introduction

The BackTracking algorithm (BT) is a central algorithm for solving instances of the *constraint satisfaction problem* (CSP). A CSP instance is represented by a constraint network, and solving it usually involves finding one solution or proving that none exists. BT performs a depth-first search, successively instantiating the variables of the constraint network in order to build a solution, and backtracking, when necessary, in order to escape from dead-ends. Many works have been devoted to improve its forward and backward phases by introducing look-ahead and look-back schemes. The forward phase consists of the processing to perform when the algorithm must instantiate a new variable. One has to decide which variable assignment to perform and which propagation effort to apply. The backward phase consists of the processing to perform when the algorithm must backtrack after encountering a dead-end. One has to decide how far to backtrack and, potentially, what to learn from the dead-end.

The relationship between look-ahead and look-back schemes has been the topic of many studies. Typically, all the efforts made by researchers to propose and experiment sophisticated look-back and look-ahead schemes are related to *thrashing*. Thrashing is the fact of repeatedly exploring the same (fruitless) subtrees during search. Sometimes, thrashing can be

* Corresponding author at: CRIL-CNRS UMR 8188, Université Lille-Nord de France.
E-mail addresses: lecoutre@cril.fr (C. Lecoutre), sais@cril.fr (L. Saïs), tabary@cril.fr (S. Tabary), Vincent.Vidal@onera.fr (V. Vidal).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2009.09.002

prevented by the use of an appropriate search heuristic or by an important propagation effort, and sometimes, it can be explained by some bad choices made earlier during search.

Early in the 90's, the Forward-Checking (FC) algorithm, which maintains during search a partial form of a property called arc consistency (which allows to identify and remove some inconsistent values), associated with the *dom* variable ordering heuristic [19] and the look-back Conflict-directed BackJumping (CBJ) technique [32], was considered as the most efficient approach to solve CSP instances. Then, Sabin and Freuder [34] (re-)introduced the MAC algorithm which fully maintains arc consistency during search, while simply using chronological backtracking. This algorithm was shown to be more efficient than FC and FC-CBJ, and CBJ was considered as useless to MAC, especially, when associated with a good variable ordering heuristic [4].

Then, it became unclear whether both paradigms were orthogonal, i.e. counterproductive one to the other, or not. First, incorporating CSP look-back techniques (such as CBJ) to the "Davis–Putnam" procedure for the propositional satisfiability problem (SAT) renders the solution of many large instances derived from real-world problems easier [2]. Second, while it is confirmed by theoretical results [9] that the more advanced the forward phase is, the more useless the backward phase is, some experiments on hard, structured problems show that adding CBJ to MAC can still present significant improvements. Third, refining the look-back techniques [18,1,23] by associating a so-called eliminating explanation (or conflict set) with every value rather than with every variable gives to the search algorithm a more powerful backjumping capability. The empirical results in [1,23] show that MAC can be outperformed by algorithms embedding such look-back techniques.

More recently, the adaptive heuristic *dom*/*wdeg* has been introduced [6]. This heuristic is able to orientate backtrack search towards inconsistent or hard parts of a constraint network by weighting constraints involved in conflicts. As search progresses, the weight of constraints difficult to satisfy becomes more and more important, and this particularly helps the heuristic to select variables appearing in the hard parts of the network. It does respect the fail-first principle: "To succeed, try first where you are most likely to fail" [19]. The new conflict-directed heuristic *dom*/*wdeg* is a very simple way to reduce thrashing [6,20,26].

Even with an efficient look-ahead technique, there still remains situations where thrashing occurs. Consequently, one can still be interested in looking for the reason of each encountered dead-end as finding the ideal ordering of variables is intractable in practice. A dead-end corresponds to a sequence of decisions (variable assignments) that cannot be extended to a solution. A dead-end is detected after enforcing a given property (e.g. arc consistency), and the set of decisions in this sequence is called a *nogood*. It may happen that a subset of decisions of the sequence forms a conflict, i.e. is a nogood itself. It is then relevant (to prevent thrashing) to identify such a conflict set and to consider its most recent decision called the *culprit decision*. Indeed, once such a decision 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[1] (Dynamic BackTracking) [18].

In this paper, an extended revised version of [27], we propose a general scheme to identify a culprit decision from any sequence of decisions leading to a dead-end through the use of a pre-established set of variables, called *testing-set*. The principle is to determine the largest prefix of the sequence, from which it is possible to instantiate all variables of the testing-set without yielding a *domain wipe-out*,[2] when enforcing a given consistency. One simple policy that can be envisioned to instantiate this general scheme is to consider, after each encountered conflict, the variable involved in the last taken decision as the unique variable in the testing-set. This is what we call *last-conflict based reasoning* (LC).

LC is an original approach that allows to (indirectly) backtrack to the culprit decision of the last encountered dead-end. To achieve it, the last assigned variable X before reaching a dead-end becomes in priority the next variable to be selected as long as the successive assignments that involve it render the network inconsistent. In other words, considering that a backtracking algorithm maintains a consistency ϕ (e.g. arc consistency) during search, the variable ordering heuristic is violated, until a backtrack to the culprit decision occurs and a singleton ϕ-consistent value for X is found (i.e. a value can be assigned to X without immediately leading to a dead-end after applying ϕ).

We show that LC can be generalized by successively adding to the current testing-set the variable involved in the last detected culprit decision. The idea is to build a testing-set that may help backtracking higher in the search tree. With this mechanism, our intention is to identify a (small) set of incompatible variables, involved in decisions of the current branch, with many interleaved irrelevant decisions. LC allows to avoid the useless exploration of many subtrees.

Interestingly enough, contrary to sophisticated backjumping techniques, our approach can be very easily grafted to any backtrack search algorithm with a simple array (only a variable for the basic use of LC) as additional data structure. Also, this approach can be efficiently exploited in different application domains.[3] In particular, the experiments that we have conducted with respect to constraint satisfaction and automated planning [17] demonstrate the general effectiveness of last-conflict based reasoning.

The paper is organized as follows. After some preliminary definitions (Section 2), we introduce the principle of nogood identification through testing-sets (Section 3). Then, we present a way of reasoning based on the exploitation of the last encountered conflict (Section 4) as well as its generalization to several conflicts (Section 5). Next, we provide (Section 6)

[1] Strictly speaking, DBT does not backtrack but simply discards the culprit decision.

[2] By domain wipe-out, we mean a domain that becomes empty.

[3] It has also been implemented in the WCSP (Weighted CSP) platform toulbar2 (see http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro).

the results of a vast experimentation that we have conducted with respect to two domains: constraint satisfaction and automated planning, before some conclusions and prospects.

2. Technical background

A *constraint network* (CN) P is a pair $(\mathcal{X}, \mathcal{C})$ where $\mathcal{X}$ is a finite set of n variables and $\mathcal{C}$ a finite set of e constraints. Each *variable* $X \in \mathcal{X}$ has an associated *domain*, denoted by $dom(X)$, which contains the set of *values* allowed for X. Each *constraint* $C \in \mathcal{C}$ involves an ordered subset of variables of $\mathcal{X}$, called *scope* of C and denoted by $scp(C)$, and has an associated *relation*, denoted by $rel(C)$, which contains the set of tuples allowed for its variables. The *arity* of a constraint is the number of variables it involves. A constraint is *binary* if its arity is 2, and *non-binary* if its arity is strictly greater than 2. A binary constraint network is a network only involving binary constraints while a non-binary constraint network is a network involving at least one non-binary constraint.

A *solution* to a constraint network is the assignment of a value to each variable such that all the constraints are satisfied. A constraint network is said to be *satisfiable* if and only if it admits at least one solution. The *Constraint Satisfaction Problem* (CSP) is the NP-hard task of determining whether a given constraint network is satisfiable or not. A CSP instance is then defined by a constraint network, and solving it involves either finding one solution or proving its unsatisfiability. To solve a CSP instance, the constraint network is processed using inference or search methods [12,25]. In the context of many search algorithms and some inference algorithms, *decisions* must be taken. Even if other forms of decisions exist (e.g. domain splitting), we introduce the classical ones:

Definition 1. Let $P = (\mathcal{X}, \mathcal{C})$ be a constraint network. A decision δ on P is either an assignment $X = a$ (also called a positive decision) or a refutation $X \neq a$ (also called a negative decision) where $X \in \mathcal{X}$ and $a \in dom(X)$.

The variable involved in a decision δ is denoted by $var(\delta)$. Of course, $\neg(X = a)$ is equivalent to $X \neq a$ and $\neg(X \neq a)$ is equivalent to $X = a$. When decisions are taken, one obtains simplified constraint networks, i.e. networks with some variables whose domain has been reduced.

Definition 2. Let P be a constraint network and Δ be a set of decisions on P. $P|_\Delta$ is the constraint network obtained from P such that:

- for every positive decision $X = a \in \Delta$, all values but a are removed from $dom(X)$, i.e. $dom(X)$ becomes $dom(X) \cap \{a\}$;
- for every negative decision $X \neq a \in \Delta$, a is removed from $dom(X)$, i.e. $dom(X)$ becomes $dom(X) \setminus \{a\}$.

In the following two subsections, we introduce some background about the inference (consistency enforcing) and search methods to which we will refer later.

2.1. Consistencies

Usually, the domains of the variables of a given constraint network are reduced by removing *inconsistent* values, i.e. values that cannot occur in any solution. In particular, it is possible to filter domains by considering some properties of constraint networks. These properties are called *domain-filtering consistencies* [11], and *generalized arc consistency* (GAC) remains the central one. By exploiting consistencies (and more generally, inference approaches), the problem can be simplified (and even, sometimes solved) while preserving solutions.

Given a consistency ϕ, a constraint network P is said to be *ϕ-consistent* if and only if the property ϕ holds on P. Enforcing a domain-filtering consistency ϕ on a constraint network means taking into account inconsistent values (removing them from domains) identified by ϕ in order to make the constraint network ϕ-consistent. The new obtained constraint network, denoted by $\phi(P)$, is called the *ϕ-closure*[4] of P. If there exists a variable with an empty domain in $\phi(P)$ then P is clearly unsatisfiable, denoted by $\phi(P) = \bot$.

Given an ordered set $\{X_1, \ldots, X_k\}$ of k variables and a k-tuple $\tau = (a_1, \ldots, a_k)$ of values, a_i will be denoted by $\tau[i]$ and also $\tau[X_i]$ by abuse of notation. If C is a k-ary constraint such that $scp(C) = \{X_1, \ldots, X_k\}$, then the k-tuple τ is said to be:

- an *allowed* tuple of C iff $\tau \in rel(C)$;
- a *valid* tuple of C iff $\forall X \in scp(C)$, $\tau[X] \in dom(X)$;
- a *support* on C iff τ is a valid allowed tuple of C.

A pair (X, a) with $X \in \mathcal{X}$ and $a \in dom(X)$ is called a *value* (of P). A tuple τ is a support for a value (X, a) on C if and only if $X \in scp(C)$ and τ is a support on C such that $\tau[X] = a$.

[4] We assume here that $\phi(P)$ is unique. This is the case for usual consistencies [3].

Definition 3. Let P be a constraint network.

- A value (X, a) of P is generalized arc-consistent, or GAC-consistent, iff for every constraint C involving X, there exists a support for (X, a) on C.
- A variable X of P is GAC-consistent iff $\forall a \in dom(X)$, (X, a) is GAC-consistent.
- P is GAC-consistent iff every variable of P is GAC-consistent.

For binary constraint networks, generalized arc consistency is simply called *arc consistency* (AC). To enforce (G)AC on a given constraint network, many algorithms have been proposed. For example, AC2001 [5] is an optimal generic algorithm that enforces AC on binary constraint networks: its worst-case time complexity is $O(ed^2)$ where e is the number of constraints and d is the greatest domain size.

On the other hand, many other domain-filtering consistencies have been introduced and studied in the literature. *Singleton arc consistency* (SAC) [10] is one such consistency which is stronger than AC: it means that SAC can identify more inconsistent values than AC. SAC guarantees that enforcing arc consistency after performing any variable assignment does not show unsatisfiability, i.e., does not entail a domain wipe-out. Note that to simplify, whether a given constraint network P is binary or non-binary, the constraint network obtained after enforcing (generalized) arc consistency on P will be denoted by $GAC(P)$.

Definition 4. Let P be a constraint network.

- A value (X, a) of P is singleton arc-consistent, or SAC-consistent, iff $GAC(P|_{X=a}) \neq \bot$.
- A variable X of P is SAC-consistent iff $\forall a \in dom(X)$, (X, a) is SAC-consistent.
- P is SAC-consistent iff every variable of P is SAC-consistent.

More generally, considering any domain-filtering consistency ϕ, *singleton ϕ-consistency* can be defined similarly to SAC. For example, a value (X, a) of P is singleton ϕ-consistent if and only if $\phi(P|_{X=a}) \neq \bot$.

2.2. Backtrack search algorithms

MAC [34] is the search algorithm which is considered as the most efficient generic complete approach to solve CSP instances. It simply maintains (generalized) arc consistency after each taken decision. A dead-end is encountered if the current network involves a variable with an empty domain (i.e. a domain wipe-out). 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 [21]. With binary branching, at each step of the search, a pair (X, a) is selected where X is an unassigned variable and a a value in $dom(X)$, and two cases are considered: the assignment $X = a$ and the refutation $X \neq a$. 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 (i.e. explored). Classically, MAC always starts by assigning variables before refuting values.

The order in which variables are assigned by a backtrack search algorithm has been recognized as a key issue for a long time. Using different *variable ordering heuristics* to solve the same CSP instance can lead to drastically different results in terms of efficiency. In this paper, we focus on some representative variable ordering heuristics. The well-known dynamic heuristic *dom* [19] selects, at each step of the search, one of the variables 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 dynamic degree of each variable, which corresponds to the number of constraints involving it as well as (at least) another unassigned variable. This is the heuristic called *bz* [7]. By directly combining domain sizes and dynamic variable degrees, one obtains *dom/ddeg* [4] which can substantially improve the search performance on some problems. Finally, in [6], the heuristic *dom/wdeg* has been introduced. The principle is to associate with each 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 as well as (at least) another unassigned variable.

On the other hand, two well-known non-chronological backtracking algorithms are Conflict-directed BackJumping (CBJ) [32] and Dynamic BackTracking (DBT) [18]. The idea 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 reason (culprit) of the dead-end. While BT systematically backtracks to the previously assigned variable, CBJ and DBT can identify a meaningful culprit decision by exploiting eliminating explanations. Of course, these different techniques can be combined; we obtain for example MAC-CBJ [33] and MAC-DBT [23].

3. Nogood identification through testing-sets

In this section, we present a general approach to identify a nogood from a so-called *dead-end sequence* of decisions through a *testing-set* which corresponds to a pre-established set of variables. The principle is to determine the largest prefix of the sequence from which it is possible to instantiate all variables of the testing-set without yielding a domain wipe-out when enforcing a consistency. The objective is to identify a nogood, smaller than the one corresponding to the dead-end sequence, by carefully selecting the testing-set.

First, we formally introduce the notion of *nogoods*. Our definition includes both positive and negative decisions as in [14,24].

Definition 5. Let P be a constraint network and Δ be a set of decisions on P.

- Δ is a nogood of P iff $P|_\Delta$ is unsatisfiable.
- Δ is a minimal nogood of P iff $\nexists \Delta' \subset \Delta$ such that Δ' is a nogood of P.

In some cases, a nogood can be obtained from a sequence of decisions. Such a sequence is called a *dead-end sequence*.

Definition 6. Let P be a constraint network and $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be a sequence of decisions on P. Σ is said to be a dead-end sequence of P iff $\{\delta_1, \ldots, \delta_i\}$ is a nogood of P.

Next, we introduce the notions of *culprit decision* and *culprit subsequence*. The culprit decision of a dead-end sequence $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ w.r.t. a testing-set S of variables and a consistency ϕ is the rightmost decision δ_j in Σ such that $\langle \delta_1, \ldots, \delta_j \rangle$ cannot be extended by instantiating all variables of S, without detecting an inconsistency with ϕ. More formally, it is defined as follows:

Definition 7. Let $P = (\mathcal{X}, \mathcal{C})$ be a constraint network, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be a sequence of decisions on P, ϕ be a consistency and $S = \{X_1, \ldots, X_r\} \subseteq \mathcal{X}$.

- A pivot of Σ w.r.t. ϕ and S is a decision $\delta_j \in \Sigma$ such that $\exists a_1 \in dom(X_1), \ldots, \exists a_r \in dom(X_r) \mid \phi(P|_{\{\delta_1,\ldots,\delta_{j-1},\neg\delta_j,X_1=a_1,\ldots,X_r=a_r\}}) \neq \bot$.
- The rightmost pivot subsequence of Σ w.r.t. ϕ and S is either the empty sequence $\langle\rangle$ if there is no pivot of Σ w.r.t. ϕ and S, or the sequence $\langle \delta_1, \ldots, \delta_j \rangle$ where δ_j is the rightmost pivot of Σ w.r.t. ϕ and S.

If Σ is a dead-end sequence then the rightmost pivot (if it exists) of Σ w.r.t. ϕ and S is called the culprit decision of Σ w.r.t. ϕ and S, and the rightmost pivot subsequence of Σ w.r.t. ϕ and S is called the culprit subsequence of Σ w.r.t. ϕ and S. S is called a testing-set.

Note that a variable may be involved both in a decision of the sequence Σ and in the testing-set S. For example, Σ may contain the negative decision $X \neq a$ while X being in S; X still has to be assigned (with a value different from a). Intuitively, one can expect that a culprit subsequence corresponds to a nogood. This is stated by the following proposition.

Proposition 1. *Let $P = (\mathcal{X}, \mathcal{C})$ be a constraint network, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be a dead-end sequence of P, ϕ be a consistency and $S \subseteq \mathcal{X}$ be a testing-set. The set of decisions contained in the culprit subsequence of Σ w.r.t. ϕ and S is a nogood of P.*

Proof. Let $S = \{X_1, \ldots, X_r\} \subseteq \mathcal{X}$ be the testing-set and let $\langle \delta_1, \ldots, \delta_j \rangle$ be the (non-empty) culprit subsequence of Σ. Let us demonstrate by induction that for all integers k such that $j \leqslant k \leqslant i$, the following hypothesis 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 by hypothesis, since Σ is a dead-end sequence. Then, let us show that, for $j < k \leqslant i$, if H(k) holds then H($k-1$) also holds. As $k > j$ and H(k) holds, we know that $\{\delta_1, \ldots, \delta_{k-1}, \delta_k\}$ is a nogood. Furthermore, δ_k is not a pivot of Σ (since $k > j$ and δ_j is the culprit decision of Σ). Hence, by Definition 7, we know that $\forall a_1 \in dom(X_1), \ldots, \forall a_r \in dom(X_r)$, $\phi(P|_{\{\delta_1,\ldots,\delta_{k-1},\neg\delta_k,X_1=a_1,\ldots,X_r=a_r\}}) = \bot$. As a result, the set $\{\delta_1, \ldots, \delta_{k-1}, \neg\delta_k\}$ is a nogood. By resolution [30], from $\{\delta_1, \ldots, \delta_{k-1}, \delta_k\}$ and $\{\delta_1, \ldots, \delta_{k-1}, \neg\delta_k\}$ being nogoods, we deduce that $\{\delta_1, \ldots, \delta_{k-1}\}$ is a nogood. So, H($k-1$) holds. For an empty culprit subsequence, we can easily adapt the previous reasoning to deduce that $\emptyset$ is a nogood. □

It is important to note that the new identified nogood may correspond to the original one. This is the case when the culprit decision of a sequence $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ is δ_i. On the other hand, when the culprit subsequence of Σ is empty, this means that P is unsatisfiable.

At this stage, one may wonder how Proposition 1 can be used in practice. When a conflict is encountered during a backtrack search, this means that a nogood has been identified: it corresponds to the set of decisions taken all along the current branch. One can then imagine to detect smaller nogoods using Proposition 1 in order to "backjump" in the search tree. There are as many ways to achieve that task as different testing-sets. The backjumping capability will depend upon the policy adopted to define the testing-sets. Different policies can thus be introduced to identify the source of the conflicts and so to reduce thrashing (as discussed in Section 4.2).

4. Reasoning from the last conflict

From now on, we consider a backtrack search algorithm (e.g. MAC) that uses a binary branching scheme and embeds an inference operator enforcing a consistency ϕ at each node of the search tree. One simple policy that can be applied to instantiate the general scheme presented in the previous section is to consider, after each encountered conflict (i.e. each time an inconsistency is detected after enforcing ϕ, which emphasizes a dead-end sequence), the variable involved in the last taken decision as forming the current testing-set. This is what we call *last-conflict based reasoning* (LC).

4.1. Principle

We first introduce the notion of *LC-subsequence*. It corresponds to a culprit subsequence identified by last-conflict based reasoning.

Definition 8. Let P be a constraint network, $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be a dead-end sequence of P and ϕ be a consistency. The LC-subsequence of Σ w.r.t. ϕ is the culprit subsequence of Σ w.r.t. ϕ and $\{X_i\}$ where $X_i = var(\delta_i)$. The testing-set $\{X_i\}$ is called the LC-testing-set of Σ.

In other words, the LC-subsequence of a sequence of decisions Σ (leading to an inconsistency) ends with the most recent decision such that, when negated, there exists a value that can be assigned, without yielding an inconsistency via ϕ, to the variable involved in the last decision of Σ. Note that the culprit decision δ_j of Σ may be a negative decision and, also, the last decision of Σ. If $j = i$, this simply means that we can find another value in the domain of the variable involved in the last decision of Σ which is compatible with all other decisions of Σ. More precisely, if δ_i is the culprit decision of Σ and δ_i is a negative decision $X_i \neq a_i$, then we necessarily have $\phi(P|_{\{\delta_1,\ldots,\delta_{i-1},X_i=a_i\}}) \neq \bot$. On the other hand, if δ_i is the culprit decision of Σ and δ_i is a positive decision $X_i = a_i$ then there exists a value $a'_i \neq a_i$ in $dom(X_i)$ such that $\phi(P|_{\{\delta_1,\ldots,\delta_{i-1},X_i \neq a_i,X_i=a'_i\}}) \neq \bot$.

LC allows identification of nogoods as shown by the following proposition.

Proposition 2. *Let P be a constraint network, Σ be a dead-end sequence of P and ϕ be a consistency. The set of decisions contained in the LC-subsequence of Σ w.r.t. ϕ is a nogood of P.*

Proof. Let δ_i be the last decision of Σ and $X_i = var(\delta_i)$. From Definition 8, the LC-subsequence of Σ w.r.t. ϕ is the culprit subsequence of Σ w.r.t. ϕ and $\{X_i\}$. We deduce our result from Proposition 1 with $S = \{X_i\}$. □

Note that the set of decisions contained in an LC-subsequence may not be a minimal nogood. Importantly, after each conflict encountered in a search tree, an LC-subsequence can be identified so as to safely backjump to its last decision. More specifically, the identification and exploitation of such nogoods can be easily embedded into a backtrack search algorithm thanks to a simple modification of the variable ordering heuristic. In practice, last-conflict based reasoning will be exploited only when a dead-end is 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 taken first. It means that LC will be used if and only if δ_i (the last decision of the sequence mentioned in Definition 8) is a positive decision. To implement LC, it is then sufficient (i) to register the variable whose assignment to a given value directly leads to an inconsistency, and (ii) always to prefer this variable in subsequent decisions (so long as it is unassigned) over the choice proposed by the underlying heuristic – whatever heuristic is used. Notice that LC does not require any additional space cost.

Fig. 1 illustrates last-conflict based reasoning. The leftmost branch on this figure corresponds to the positive decisions $X_1 = a_1, \ldots, X_i = a_i$, such that $X_i = a_i$ leads to a conflict. With ϕ denoting the consistency maintained during search, we have: $\phi(P|_{X_1=a_1,\ldots,X_i=a_i}) = \bot$. At this point, X_i is registered by LC for future use, i.e. the testing-set is $\{X_i\}$, and a_i is removed from $dom(X_i)$, i.e. $X_i \neq a_i$. Then, instead of pursuing the search with a new selected variable, X_i is chosen to be assigned with a new value. In our illustration, this leads once again to a conflict, this value 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 (symbolized by a triangle labelled with X_i whose base is drawn using a solid line). The algorithm then backtracks to the assignment $X_{i-1} = a_{i-1}$, going to the right branch $X_{i-1} \neq a_{i-1}$. As X_i is still recorded by LC, it is selected in priority, and all values of $dom(X_i)$ are proved here to be singleton ϕ-inconsistent. The algorithm finally backtracks to the decision $X_j = a_j$, going to the right branch $X_j \neq a_j$. Then, as $\{X_i\}$ is still an active LC-testing-set, X_i is preferred again and the values of $dom(X_i)$ are tested. But, as one of them

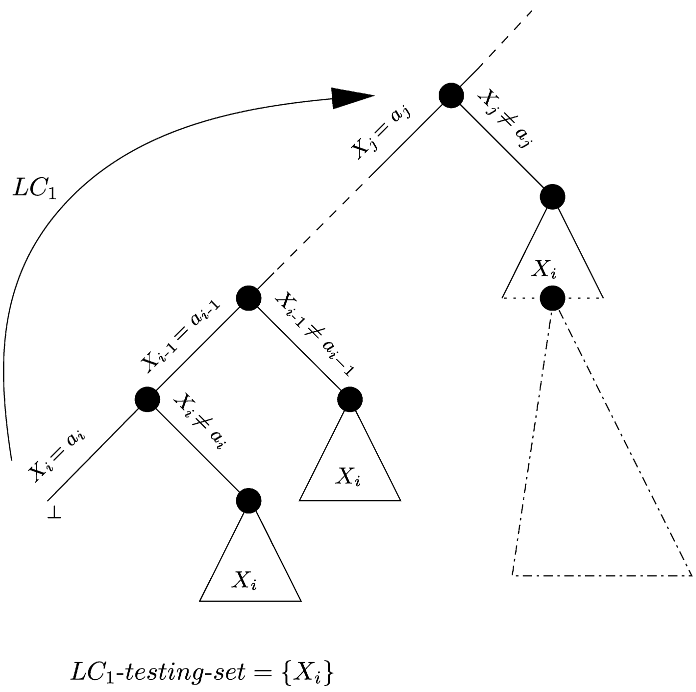

Fig. 1. Reasoning from the last conflict illustrated with a partial search tree. A consistency ϕ is maintained at each node. A triangle labelled with a variable X and drawn using a solid base line (resp. a dotted base line) represents the fact that no (resp. a) singleton ϕ-consistent value exists for X.

does not lead to a conflict (symbolized by a triangle labelled with X_i whose base is drawn using a dotted line), the search can continue with a new assignment for X_i. The variable X_i is then unregistered (the testing-set becomes empty), and the choice for subsequent decisions is left to the underlying heuristic, until the next conflict occurs.

As a more concrete example, consider a constraint network with the variables $\{X_0, X_1, X_2, X_3, X_4, X_5, X_6\}$ and the constraints $\{X_1 \neq X_4, X_1 \neq X_5, X_1 \neq X_6, X_4 \neq X_5, X_4 \neq X_6, X_5 \neq X_6\}$. Here, we have a clique of binary dis-equality constraints composed of four variables $\{X_1, X_4, X_5, X_6\}$, the domain of each one being $\{0, 1, 2\}$, and three variables $\{X_0, X_2, X_3\}$ involved in no constraint, the domain of each one being $\{0, 1\}$. Even if the introduction of isolated variables seems to be quite particular, it can be justified by the fact that it may happen during search (after some decisions have been taken). This phenomenon, and more generally the presence of several connected components, frequently occurs when solving structured instances. Fig. 2 depicts the search tree built by MAC where variables and values are selected in lexicographic order, which is used here to facilitate understanding of the example. In this figure, each leaf corresponds to a direct failure, after enforcing arc consistency; MAC explores 68 nodes to prove the unsatisfiability of this problem. Fig. 3 depicts the search tree built by MAC-LC$_1$ using the same lexicographic order, where LC$_1$ denotes the implementation of last-conflict based reasoning, as presented above. This time, MAC-LC$_1$ only explores 21 nodes. Indeed, reasoning from the last conflict allows search to focus on the hard part of the network (i.e. the clique).

By using an operator that enforces ϕ to identify LC-subsequences as described above, we obtain the following complexity result.

Proposition 3. *Let P be a constraint network, ϕ be a consistency and $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be a dead-end sequence of P. The worst-case time complexity of computing the LC-subsequence of Σ w.r.t. ϕ is $O(id\gamma)$ where γ is the worst-case time complexity of enforcing ϕ.*

Proof. The worst case happens when the computed LC-subsequence of Σ is empty. In this case, this means that, for each decision, we check the singleton ϕ-consistency of X_i. Checking the singleton ϕ-consistency of a variable corresponds to at most d calls to an algorithm enforcing ϕ, where d is the greatest domain size. Thus, the total worst-case time complexity is id times the complexity of the ϕ-enforcing algorithm, denoted by γ. We obtain $O(id\gamma)$. □

When LC is embedded in MAC, we obtain the following complexity.

Corollary 1. *Let P be a binary constraint network and $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ be a dead-end sequence of decisions that corresponds to a branch built by MAC. Assuming that the current LC-testing-set is $\{var(\delta_i)\}$, the worst-case time complexity, for MAC-LC$_1$, to backtrack up to the last decision of the LC-subsequence of Σ w.r.t. AC is $O(end^3)$.*

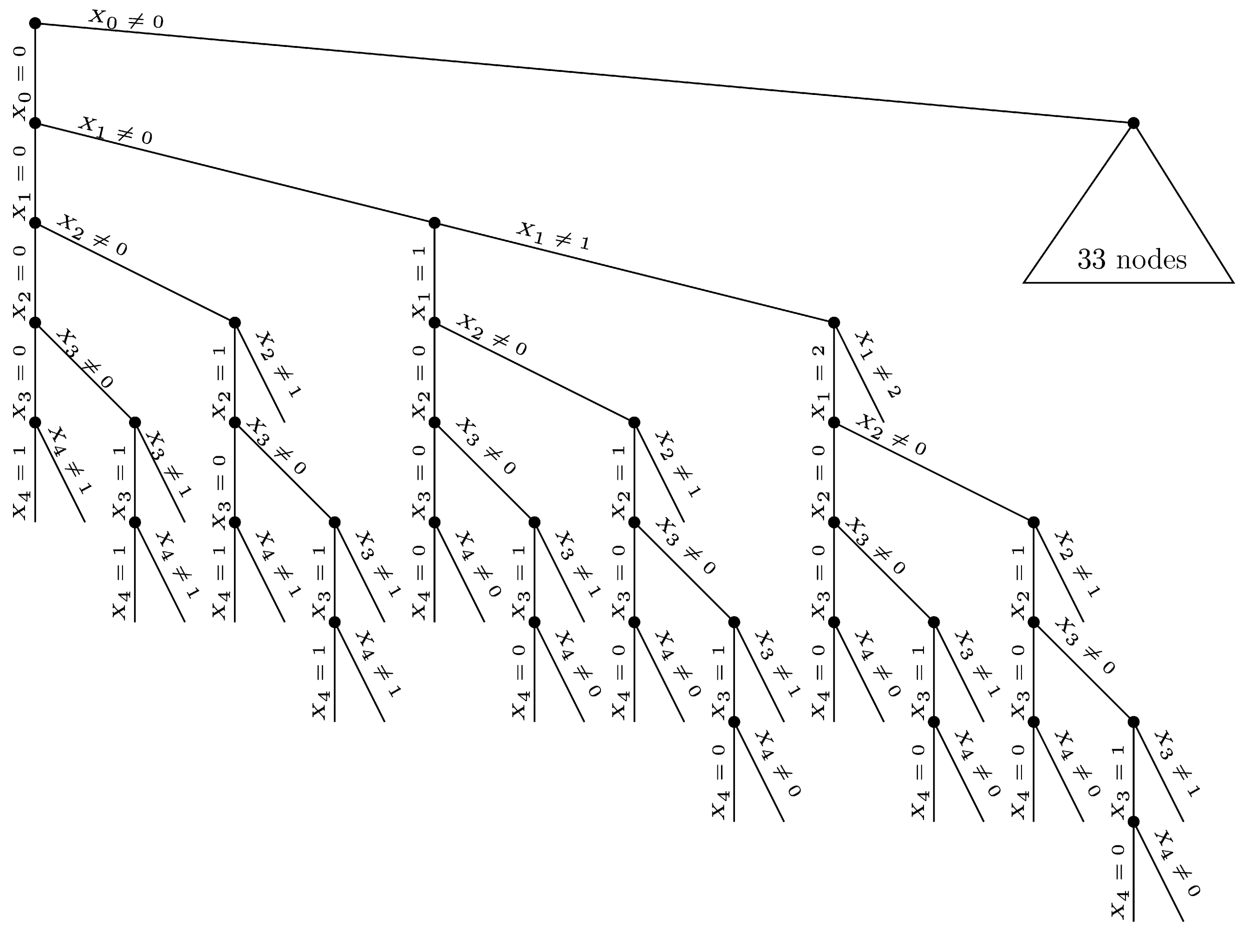

Fig. 2. Search tree built by MAC (68 explored nodes).

Proof. First, we know, as positive decisions are performed first by MAC, that 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 X_i since we have to directly backtrack. So, using an optimal AC algorithm in $O(ed^2)$, we obtain an overall complexity in $O(end^3)$. □

4.2. Preventing thrashing using LC

Thrashing is a phenomenon that deserves to be carefully studied because an algorithm subject to thrashing can be very inefficient. We know that whenever a value is removed from the domain of a variable, it is possible to compute an explanation of this removal by collecting 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 backjump 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 [6]. When the queens subproblem and the knights subproblem 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), MAC combined with a non-chronological backtracking 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), MAC-CBJ and MAC-DBT become subject to thrashing (when a standard variable ordering heuristic such as *dom*, *bz* or *dom/ddeg* is used) because the last assigned queen variable is considered as participating to the reason of 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, computing new ones [22] or managing several explanations [35,31]. However, this is far beyond the scope of this paper.

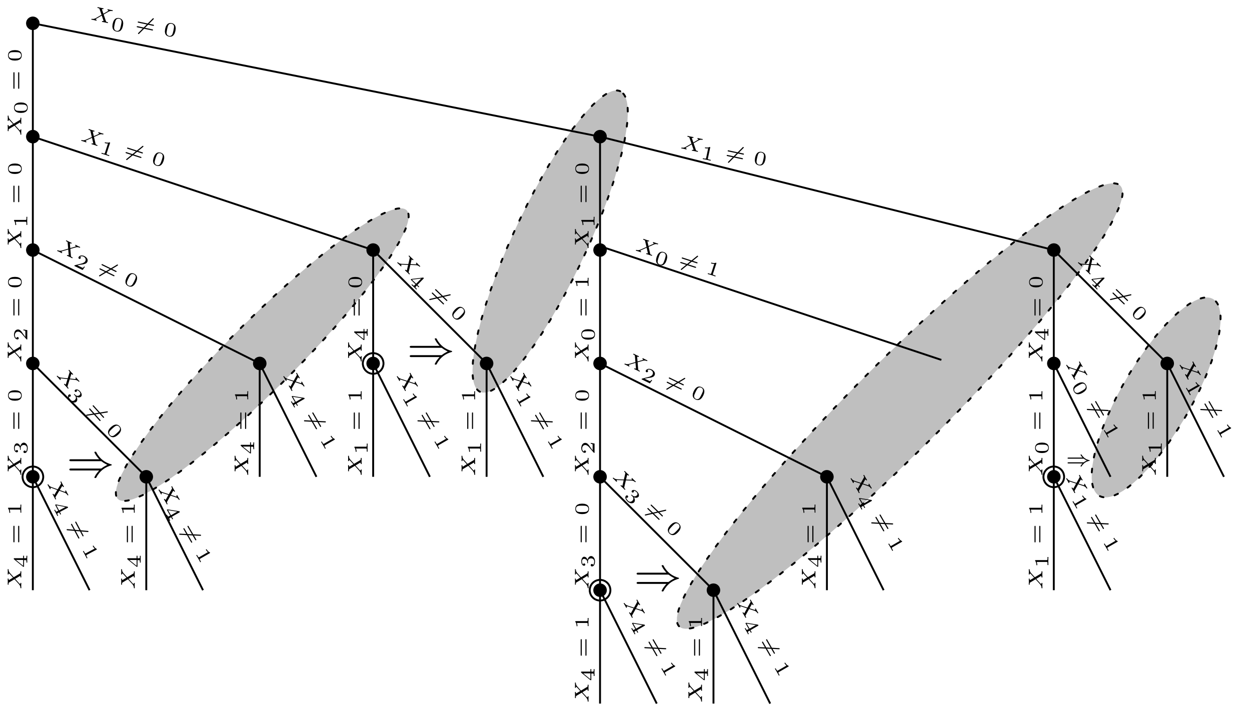

Fig. 3. Search tree built by MAC-LC$_1$ (21 explored nodes). Circled nodes identify variables forming testing-sets. They point to grey areas where a culprit subsequence is sought.

Table 1
Cost of running variants of MAC with *bz* as variable ordering heuristic (time-out is 2 hours).

Instance		MAC	MAC-CBJ	MAC-DBT	MAC-LC$_1$
qk-25-25-5-add	CPU	> 2 h	11.7	12.5	58.9
	nodes	–	703	691	10 053
qk-25-25-5-mul	CPU	> 2 h	> 2 h	> 2 h	66.6
	nodes	–	–	–	9922

Reasoning from the last conflict is a new way of reducing 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.

Table 1 illustrates the thrashing prevention capability of LC on the two instances mentioned above. Clearly, MAC, MAC-CBJ and MAC-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, MAC-LC$_1$ can prove the unsatisfiability of this instance. The reason is that all values in the domain of knight variables are singleton arc-inconsistent. When such a variable is reached, LC guides search up to the root of the search tree.

5. A generalization: Reasoning from last conflicts

A generalization of the last conflict policy, previously introduced, can now be envisioned. As before, after each conflict, the testing-set is initially composed of the variable involved in the last taken decision. However, it is also updated each time a culprit decision is identified.

5.1. Principle

To define testing-sets, the policy previously introduced can be generalized as follows. At each dead-end the testing-set initially consists, as before, of the variable X_i involved in the most recent decision δ_i. When the culprit decision δ_j is identified, the variable X_j involved in δ_j is included in the testing-set. The new testing-set $\{X_i, X_j\}$ may help backtracking nearer the root of the search tree. Of course, this form of reasoning can be extended recursively. This mechanism is intended to identify a (small) set of incompatible variables involved in decisions of the current branch, although these may be interleaved with many irrelevant decisions. We now formalize this approach before illustrating it.

Definition 9. Let P be a constraint network, Σ be a dead-end sequence of P and ϕ be a consistency. We recursively define the kth LC-testing-set and the kth LC-subsequence of Σ w.r.t. ϕ, respectively called LC$_k$-testing-set and LC$_k$-subsequence and denoted by S_k and Σ_k, as follows:

- For $k = 1$, S_1 and Σ_1 respectively correspond to the LC-testing-set of Σ and the LC-subsequence of Σ w.r.t. ϕ.
- For $k > 1$, if $\Sigma_{k-1} = \langle\rangle$, then $S_k = S_{k-1}$ and $\Sigma_k = \Sigma_{k-1}$. Otherwise, $S_k = S_{k-1} \cup \{X_{k-1}\}$ where X_{k-1} is the variable involved in the last decision of Σ_{k-1} and Σ_k is the rightmost pivot subsequence of Σ_{k-1} w.r.t. ϕ and S_k.

The following proposition is a generalization of Proposition 2, and can be demonstrated by induction on k.

Proposition 4. *Let P be a constraint network, Σ be a dead-end sequence of P and ϕ be a consistency. For any $k \geqslant 1$, the set of decisions contained in Σ_k, which is the LC$_k$-subsequence of Σ w.r.t. ϕ, is a nogood of P.*

Proof. Let us demonstrate by induction that for all integers $k \geqslant 1$, the following hypothesis, denoted H(k), holds:

$$\mathrm{H}(k)\text{: the set of decisions contained in } \Sigma_k \text{ is a nogood.}$$

First, let us show that H(1) holds. From Proposition 2, we know that the set of decisions contained in Σ_1 is a nogood. Then, let us show that, for $k > 1$, if H($k-1$) holds then H(k) also holds. As $k > 1$ and H($k-1$) holds, we know that the set of decisions contained in Σ_{k-1} is a nogood and, consequently, Σ_{k-1} is a dead-end sequence. Using Definition 7, we know that the rightmost pivot subsequence Σ_k is a culprit subsequence. Hence, using Proposition 1, we deduce that the set of decisions contained in Σ_k is a nogood. □

For any $k > 1$ and any given dead-end sequence Σ, LC$_k$ will denote the process that consists in computing the LC$_k$-subsequence Σ_k of Σ. When computing Σ_k, we may have $\Sigma_k \neq \Sigma_{k-1}$ meaning that the original nogood has been reduced k times (and S_k is composed of k distinct variables). However, a fixed point may be reached at a level $1 \leqslant j < k$, meaning that $\Sigma_j = \Sigma_{j+1}$ and either $j = 1$ or $\Sigma_j \neq \Sigma_{j-1}$. The fixed point is reached when the current testing set is composed of $j + 1$ variables: no new variable can be added to the testing set because the identified culprit decision is the last decision of the current dead-end sequence.

In practice, we will use the generalized version of LC in the context of a backtrack search. If a fixed point is reached at a level $j < k$, the process of last-conflict based reasoning is stopped and the choice of subsequent decisions is left to the underlying heuristic until the next conflict occurs. On the other hand, we will restrict pivots to be positive decisions, only. Indeed, it is not relevant to consider a negative decision $X \neq a$ as a pivot because it would consist in building a third branch within the MAC search tree identical to the first one. The subtree under the opposite decision $X = a$ has already been refuted, since positive decisions are taken first.

As an illustration, Fig. 4 depicts a partial view of a search tree. The leftmost branch corresponds to a dead-end sequence of decisions Σ. By definition, the LC$_1$-testing-set of Σ is only composed of the variable X_i (which is involved in the last decision of Σ). So, the algorithm assigns X_i in priority in order to identify the culprit decision of Σ (and the LC$_1$-subsequence). In our illustration, no value in $dom(X_i)$ is found to be singleton ϕ-consistent until the algorithm backtracks up to the positive decision $X_j = a_j$. This decision is then identified as the culprit decision of Σ, and so, in order to compute the LC$_2$-subsequence, the LC$_2$-testing-set is built by adding X_j to the LC$_1$-testing-set. From now, X_i and X_j will be assigned in priority. The LC$_2$-subsequence is identified when backtracking to the decision $X_k = a_k$. Indeed, from $X_k \neq a_k$, it is possible to instantiate the two variables of the LC$_2$-testing-set. Then, X_k is added to the LC$_2$-testing-set, but as the variables of this new testing-set can now be assigned, last-conflict reasoning is stopped because a fixed point is reached (at level 2) and search continues as usual.

Let us consider again the example introduced in Section 4.1 and the search trees (see Figs. 2 and 3) built by MAC and MAC-LC$_1$. This time, Fig. 5 represents the search tree built by MAC-LC$_2$. We recall that with MAC-LC$_2$, the testing-sets may contain up to two variables. Here, after the first conflict (leftmost branch), the testing-set is initialized with $\{X_4\}$ and when the singleton arc consistent value $(X_4, 0)$ is found (after decisions $X_0 = 0$ and $X_1 \neq 0$), the testing-set becomes $\{X_4, X_1\}$. As any instantiation of these two variables systematically leads to a failure (when enforcing arc consistency), MAC-LC$_2$ is able to efficiently prove the unsatisfiability of this instance: MAC-LC$_2$ only explores 16 nodes (to be compared with the 68 and 21 explored nodes of MAC and MAC-LC$_1$).

5.2. A small example

Let us also introduce a toy problem, called the pawns problem, which illustrates the capability of generalized last-conflict reasoning to circumscribe the difficult parts of problem instances. The pawns problem consists in putting p pawns on squares of a chessboard of size $n \times n$ such that no two pawns can be put on the same square and the distance between two of them must be strictly less than $p - 1$. Here, in our modelling, each square of a chessboard is numbered from 1 to $n \times n$ and the distance between two squares is the absolute value of the difference of their numbers. Then, p variables represent the pawns of the problem and their domain represent the $n \times n$ squares of the chessboard. For $p \geqslant 2$, this problem

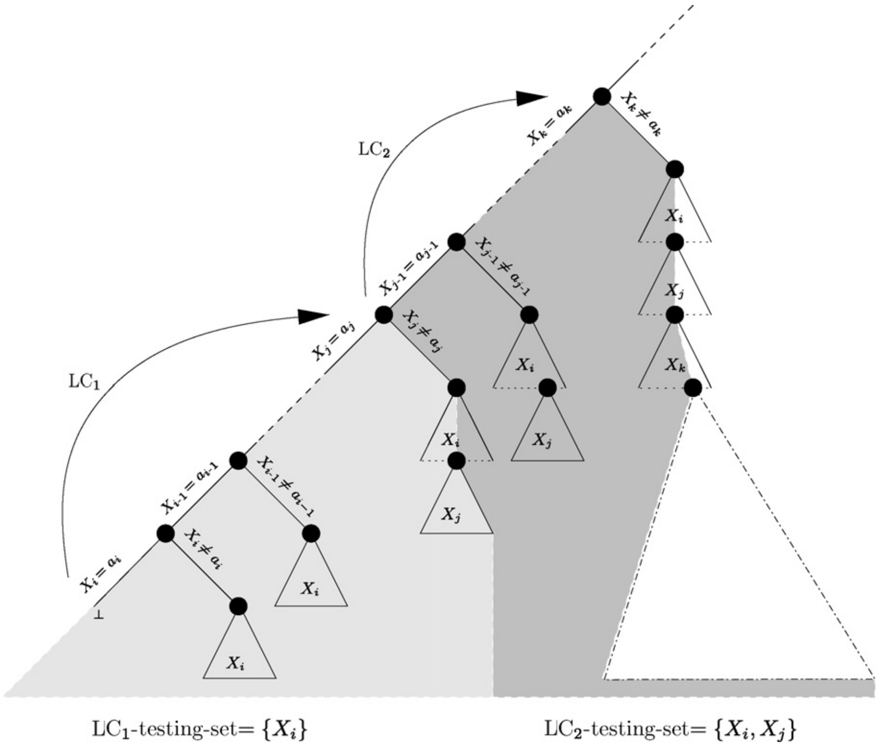

Fig. 4. Generalized reasoning from the last conflict illustrated with a partial search tree. A consistency ϕ is maintained at each node. A triangle labelled with a variable X and drawn using a solid base line (resp. a dotted base line) represents the fact that no (resp. a) singleton ϕ-consistent value exists for X.

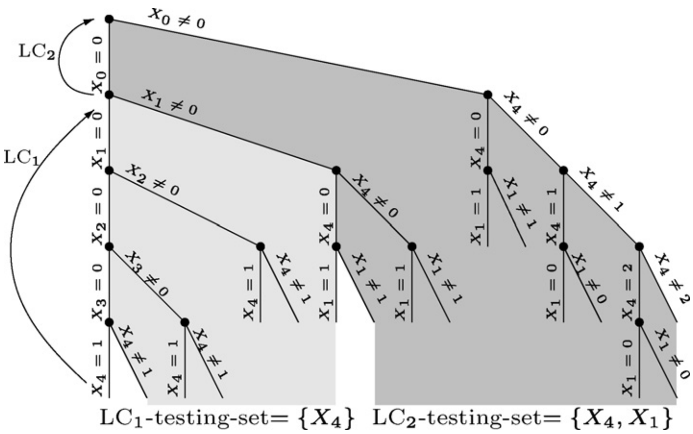

Fig. 5. Search tree built by MAC-LC$_2$ (16 explored nodes).

is unsatisfiable (since it is equivalent to put p pawns on $p-1$ squares). Interestingly, we can show that, during a search performed by MAC, we may have to instantiate up to $p-3$ variables.

We can merge this problem with the classical queens problem: pawns and queens cannot be put on the same square. Instances of this new queens–pawns problem are then denoted by qp-n-p with p the number of pawns and n the number of queens. This problem (like the queens–knights problem) produces a lot of thrashing. Indeed, in the worst case, the unsatisfiability of the pawns problem must be proved for each solution of the queens problem. Using LC$_{p-2}$, one can expect to identify the pawn incompatible variables and to use them as LC$_{p-2}$-testing-set.

Table 2 presents the results obtained with MAC equipped with LC reasoning (LC$_k$ with $k \in [1, 7]$) or not (LC$_0$) on instances qp-12-p with p ranging from 4 to 9. The size of the chessboard was set to 12×12 and the time limit was 2 hours. As expected, to solve an instance qp-12-p, it is better to use LC$_{p-2}$ as variables that correspond to pawns can be collected by this approach. Note that if we use LC$_k$ with $k \geqslant p-2$, whatever k is, the number of nodes does not change (significantly). If $k < p-2$, solving the problem is more difficult: one can only identify a subset of the $p-2$ incompatible variables.

Table 2
Results obtained with MAC-LC_k with $k \in [0, 7]$, using *bz* and *dom/wdeg* as heuristics, on the *queens–pawns* problem.

		bz							
		LC_0	LC_1	LC_2	LC_3	LC_4	LC_5	LC_6	LC_7
qp-12-4	CPU	815	1.19	0.98	1.09	1.25	1.22	1.12	1.12
	nodes	6558 K	2713	2719	2719	2719	2719	2719	2719
qp-12-5	CPU	2620	3.16	2.66	2.24	2.95	2.87	2.33	2.39
	nodes	28 M	13,181	13,140	12,523	12,523	12,523	12,523	12,523
qp-12-6	CPU	*time-out*	471	11.0	10.7	9.39	9.61	9.69	10.2
	nodes		5271 K	66,701	75,812	67,335	67,335	67,335	67,335
qp-12-7	CPU	*time-out*	*time-out*	74.5	469	62.7	55.9	54.8	55.2
	nodes			584 K	5144 K	432 K	418 K	418 K	418 K
qp-12-8	CPU	*time-out*	*time-out*	*time-out*	5587	710	669	385	389
	nodes				63 M	6003 K	5820 K	2978 K	2978 K
qp-12-9	CPU	*time-out*	*time-out*	*time-out*	*time-out*	*time-out*	6944	*time-out*	3126
	nodes						67 M		24 M
		dom/wdeg							
		LC_0	LC_1	LC_2	LC_3	LC_4	LC_5	LC_6	LC_7
qp-12-4	CPU	1.12	1.34	1.15	2.07	1.19	1.58	1.19	1.13
	nodes	4273	3530	3255	2719	2719	2719	2719	2719
qp-12-5	CPU	2.36	2.77	2.95	2.13	2.33	2.57	2.39	2.33
	nodes	12,847	14,497	16,064	12,523	12,523	12,523	12,523	12,523
qp-12-6	CPU	9.88	12.4	13.4	10.2	9.39	9.21	9.55	9.28
	nodes	79,191	80,794	94,225	70,832	67,335	67,335	67,335	67,335
qp-12-7	CPU	67.0	80.2	89.0	71.8	66.2	54.9	55.4	51.8
	nodes	568 K	544 K	638 K	515 K	478 K	418 K	418 K	418 K
qp-12-8	CPU	744	589	687	554	538	459	390	364
	nodes	6240 K	4083 K	4897 K	3961 K	3841 K	3390 K	2978 K	2978 K
qp-12-9	CPU	5884	4887	5651	4743	4722	4328	3689	2947
	nodes	49 M	34 M	39 M	33 M	32 M	31 M	27 M	24 M

Algorithm 1: *solve*()

Input: a constraint network P
Output: *true* iff P is satisfiable

```
1  P ← φ(P)
2  if P = ⊥ then
3  └ return false
4  if ∀X ∈ P, |dom(X)| = 1 then
5  └ return true
6  X ← selectVariable(P)
7  a ← selectValue(X)
8  if solve(P|X=a) then
9  └ return true
10 if candidate = null ∧ |testingSet| < k ∧ X ∉ testingSet then
11 └ candidate ← X
12 return solve(P|X≠a)
```

5.3. Implementation details

Reasoning from last conflicts can be implemented by slight modifications of a classical backtrack search algorithm (see function *solve* described in Algorithm 1) and its associated variable selection procedure (see function *selectVariable*, Algorithm 2). The function *solve* works the following way. First, an inference operator establishing a consistency ϕ such as AC is applied on a constraint network P (line 1). To simplify the presentation, we suppose here that ϕ is a domain filtering consistency at least as strong as the partial form of arc consistency established (maintained) by the FC algorithm [19]. If the resulting constraint network is trivially inconsistent (a variable has an empty domain), *solve* returns *false* (lines 2–3). Else, if the domain of all variables in P is reduced to only one value, a solution is found and *solve* returns true (lines 4–5). If P is not proved inconsistent by ϕ and there remains several possible values for at least one variable, a new decision has to be taken. A variable X is thus selected by a call to *selectVariable* (line 6), and a value a is picked from $dom(X)$ by a call to *selectValue*. Two branches are then successively explored by recursive calls to *solve*: the assignment $X = a$ (lines 8–9) and the refutation $X \neq a$ (line 12). Between these two calls, two lines have been introduced (lines 10–11) in order to manage

Algorithm 2: *selectVariable*()

```
   Input: a constraint network P
   Output: a variable X to be used for branching
1  foreach X ∈ testingSet do
2      if |dom(X)| > 1 then
3          return X
4  if candidate ≠ null ∧ |dom(candidate)| > 1 then
5      X ← candidate
6      testingSet ← testingSet ∪ {X}
7  else
8      X ← variableOrderingHeuristic.selectVariable(P))
9      testingSet ← ∅
10 candidate ← null
11 return X
```

LC. We will discuss them below. Apart from these two lines, most of the modifications lie in *selectVariable*, Algorithm 2. Classically, this function selects the best variable to be assigned thanks to the given variable ordering heuristic implemented by the function *variableOrderingHeuristic.selectVariable*. The algorithm we propose here modifies this selection mechanism to reflect the different possible states of search:

1. Some variables have been collected in a testing-set, and we look for an instantiation of them which is consistent with the current node of the search tree. Variables of this testing-set are then preferred over all other variables (lines 1–3), until the domains of the variables in the testing-set are all reduced to singletons. The order in which the variables of the testing-set are picked is not crucial, as the maximal size of a testing-set is limited by k and is kept relatively low in practice. This step can be viewed as a complete local exploration of a small subtree until the variables of the testing-set are all assigned (their domains are reduced to singletons).
2. When all variables of a testing-set are assigned, there may exist a candidate variable to be added to the testing-set (lines 4–5). In that case, the variable *candidate* corresponds to a variable whose domain contains more than one value. This candidate has been pointed out in the function *solve* (lines 10–11), just before the refutation of a given value from its domain, under the following conditions:
   - Firstly, there was no candidate yet ($candidate = null$). This happens when a conflict has been encountered under the assignment $X = a$ in the left branch: variables of the testing-set are going to be explored in the right branch under the refutation $X \neq a$, and X will then be potentially added later to the testing-set.
   - Secondly, the maximal size k of a testing-set has not been reached ($|testingSet| < k$).
   - Thirdly, X must not be already present in the testing-set ($X \notin testingSet$). $X \in testingSet$ may happen when X has just been entered into the testing-set and search focuses on it.

   A candidate will enter the testing-set only if an instantiation of the variables currently in the testing-set is found. If no instantiation of the testing-set can be found, the candidate is not added to the testing-set and will be replaced by another one after having backtracked higher in the search tree.
3. If an instantiation of the testing-set has already been found (possibly, the testing-set being empty) and if there is no candidate or the candidate is already assigned, then the classical heuristic chooses a new variable to assign, and the testing-set is emptied.

6. Experiments

In order to show the practical interest of the approach described in this paper, we have conducted an extensive experimentation on a cluster of Xeon 3.0 GHz with 1 GB of RAM under Linux, with respect to two research domains: constraint satisfaction and automated artificial intelligence planning. To do this, we have respectively equipped the constraint solver Abscon [28] and the planner CPT [37] with last-conflict based reasoning.

6.1. Results with the CSP solver Abscon

We first present the results obtained with the solver Abscon. For our experimentation, we have used MAC (using chronological backtracking) and studied the impact of LC w.r.t. various variable ordering heuristics (*dom/ddeg*, *dom/wdeg*, *bz*). Recall that LC_0 denotes MAC alone and LC_k denotes the approach that consists in computing LC_k-subsequences, i.e. the generalized last-conflict based approach where at most k variables are collected. Performance is measured in terms of the number of

Table 3
Results obtained with MAC, MAC-LC_1 and MAC-LC_2 on random instances (time-out is 20 minutes).

		dom/ddeg			*dom/wdeg*			*bz*		
		LC_0	LC_1	LC_2	LC_0	LC_1	LC_2	LC_0	LC_1	LC_2
Random instances from Model D (100 instances per series)										
$\langle 40, 8, 753, 0.1\rangle$	CPU	12.1	60.8	85.3	10.5	55.6	78.8	12.1	59.8	83.4
	nodes	45,388	232 K	326 K	45,393	241 K	322 K	45,388	232 K	326 K
$\langle 40, 11, 414, 0.2\rangle$	CPU	12.8	44.6	59.7	14.6	54.2	68.9	16.0	47.4	60.8
	nodes	58,443	203 K	266 K	70,560	253 K	312 K	73,004	213 K	280 K
$\langle 40, 16, 250, 0.35\rangle$	CPU	12.2	33.3	44.1	14.5	35.9	48.2	21.0	41.2	47.6
	nodes	59,448	158 K	215 K	72,556	182 K	237 K	104 K	200 K	233 K
$\langle 40, 25, 180, 0.5\rangle$	CPU	16.7	27.3	41.2	17.0	34.7	41.6	46.7	44.9	44.2
	nodes	82,836	134 K	205 K	81,921	173 K	200 K	238 K	227 K	225 K
$\langle 40, 40, 135, 0.65\rangle$	CPU	11.8	15.3	23.2	11.0	16.1	21.5	52.21	22.65	26.02
	nodes	52,814	70,113	110 K	47,665	72,547	101 K	242 K	102 K	123 K
$\langle 40, 80, 103, 0.8\rangle$	CPU	13.9	10.9	16.2	6.45	9.62	14.0	129 (5)	15.3	19.5
	nodes	49,923	39,926	67,513	20,994	34,375	57,227	487 K	57,115	74,583
$\langle 40, 180, 84, 0.9\rangle$	CPU	21.7	15.8	26.3	8.48	11.9	19.8	111 (3)	16.4	21.1
	nodes	55,403	39,281	79,280	17,348	29,047	62,003	317 K	40,407	61,516
Random forced instances from Model RB (5 instances per series)										
frb35-17	CPU	4.30	5.01	5.52	3.26	4.94	7.39	6.39	5.35	5.89
	nodes	15,844	18,983	21,439	10,160	18,816	29,564	24,872	20,518	22,952
frb40-19	CPU	32.7	111	64.2	25.3	98.7	126	47.3	106	128
	nodes	135 K	463 K	271 K	103 K	452 K	564 K	196 K	447 K	549 K
geom (100 instances per series)										
geom	CPU	10.2	24.5	32.8	6.92	27.4	34.3	41.6 (1)	26.6	33.8
	nodes	30,847	76,706	103 K	21,712	85,865	115 K	179 K	85,396	106 K

Table 4
Results obtained with MAC, MAC-LC_1 and MAC-LC_2 on academic and patterned instances (time-out is 20 minutes).

		dom/ddeg			*dom/wdeg*			*bz*		
		LC_0	LC_1	LC_2	LC_0	LC_1	LC_2	LC_0	LC_1	LC_2
Aim (24 instances per series)										
aim-100	CPU	636 (10)	30.2	35.6	0.60	0.54	0.53	647 (12)	50.4	50.2
	nodes	9150 K	428 K	489 K	3106	2485	2330	9488 K	718 K	718 K
aim-200	CPU	977 (18)	737 (13)	740 (13)	5.82	4.75	4.85	985 (18)	740 (14)	738 (14)
	nodes	12 M	8455 K	8598 K	64,798	52,857	55,387	12 M	9071 K	9165 K
Composed instances (10 instances per series)										
25-1-40	CPU	1,200 (10)	0.51	0.54	0.51	0.53	0.49	0.47	0.42	0.48
	nodes	13 M	74	74	161	74	74	4	4	4
25-10-20	CPU	27.1	0.64	0.63	0.58	0.60	0.58	229 (1)	0.77	0.74
	nodes	272 K	161	160	200	159	159	2599 K	220	198
Coloring instances (22 instances per series)										
dsjc/myciel	CPU	6.88	4.50	6.77	13.6	10.2	10.7	105	9.54	7.52
...	nodes	41,020	32,046	38,065	150 K	110 K	108 K	1500 K	93,070	75,256
Sadeh job-shop instances (10 instances per series)										
e0ddr1-10	CPU	960 (8)	548 (4)	501 (4)	511 (4)	445 (3)	498 (4)	720 (6)	600 (5)	492 (4)
	nodes	9811 K	5412 K	4591 K	4588 K	4164 K	4615 K	6487 K	5608 K	4647 K
enddr1-10	CPU	600 (5)	142 (1)	124 (1)	123 (1)	124 (1)	124 (1)	360 (3)	259 (2)	243 (2)
	nodes	6535 K	1345 K	1191 K	1101 K	1127 K	1162 K	3162 K	2274 K	2270 K
Ehi instances (100 instances per series)										
ehi-85-297	CPU	475 (13)	0.91	0.62	0.87	0.43	0.43	301 (8)	0.69	0.53
	nodes	529 K	557	281	1292	146	146	362 K	311	172
ehi-90-315	CPU	601 (23)	1.17	0.65	0.85	0.44	0.48	402 (14)	0.70	0.53
	nodes	616 K	674	264	1210	140	140	431 K	282	155
QCP (15 instances per series)										
qcp-10-67	CPU	98.2	0.56	0.50	0.47	0.52	0.47	80.3	0.54	0.49
	nodes	1038 K	885	366	171	169	168	897 K	854	369
qcp-15-120	CPU	736 (7)	704 (7)	637 (6)	34.4	36.5	35.2	727 (7)	729 (7)	628 (6)
	nodes	3377 K	3594 K	3334 K	232 K	254 K	241 K	3907 K	3845 K	3491 K

Table 5
Results obtained with MAC, MAC-LC_1 and MAC-LC_2 on real-world instances (time-out is 20 minutes per instance).

		dom/ddeg			dom/wdeg			bz		
		LC_0	LC_1	LC_2	LC_0	LC_1	LC_2	LC_0	LC_1	LC_2
FAPP instances (11 instances per series)										
fapp02	CPU	564 (5)	8.03	7.71	9.14	7.51	7.42	318 (2)	7.20	7.04
	nodes	688 K	582	415	966	369	337	244 K	291	270
fapp03	CPU	115 (1)	7.83	7.74	7.48	7.79	7.74	115 (1)	8.41	8.09
	nodes	9694	153	208	168	181	147	11,023	237	211
fapp04	CPU	225 (2)	9.60	9.79	12.0	9.15	20.2	658 (6)	96.8	42.5
	nodes	123 K	297	364	738	319	1693	397 K	17,771	3836
RLFAP Graphs (12 and 14 instances per series)										
graphMods	CPU	800 (8)	315 (3)	51.5	2.28	3.80	1.50	1000 (10)	303 (3)	2.53
	nodes	1585 K	858 K	140 K	5509	14,601	2208	2350 K	1642 K	3185
graphs	CPU	1.35	1.37	1.37	1.19	1.28	1.26	86.8 (1)	1.59	1.46
	nodes	313	313	313	313	313	313	521 K	497	378
Radar surveillance (50 instances per series)										
radar-8-24-3-2	CPU	408 (17)	1.70	0.48	0.18	0.17	0.16	210 (8)	0.84	0.22
	nodes	4651 K	14,699	3085	122	106	107	2214 K	5804	657
radar-8-30-3-0	CPU	423 (17)	24.8 (1)	8.03	0.21	0.19	0.21	101 (4)	0.94	1.92
	nodes	4727 K	141 K	43,635	219	209	213	1001 K	6067	10,217

visited nodes (nodes) and the CPU time in seconds. Importantly, all CSP instances that have been experimented come from the second constraint solver competition[5] where they can be downloaded.

First, we experimented LC_1 and LC_2 on different series of random problems. Seven classes of binary instances near crossover points have been generated following Model D [36,16]. For each class $\langle n, d, e, t\rangle$, the number of variables n is 40, the domain size d lies between 8 and 180, the number of constraints e lies between 753 and 84 (so the density is between 0.96 and 0.1) and the tightness t lies between 0.1 and 0.9. Here, tightness t is the probability that a pair of values is disallowed by a relation. The first class $\langle 40, 8, 753, 0.1\rangle$ corresponds to dense instances involving constraints of low tightness whereas the seventh one $\langle 40, 180, 84, 0.9\rangle$ corresponds to sparse instances involving constraints of high tightness. It is important to note that a significant sampling of domain sizes, densities and tightnesses is provided. Two series of random instances generated using Model RB [39] and forced to be satisfiable as described in [38] were also tested. We finally experimented the series of "geometric" instances proposed by R. Wallace. Constraint relations are generated in the same way as for homogeneous random CSP instances, but instead of a density parameter, a "distance" parameter is used.

The results that we have obtained are given in Table 3. The number of unsolved instances within 20 minutes is given into brackets, in this case the CPU time must be considered as a lower bound. Broadly, using LC on random instances is penalizing because these instances do not contain any structure. MAC alone is better than LC_1, itself being better than LC_2. However on series geom and classes $\langle 40, 80, 103, 0.8\rangle$ and $\langle 40, 180, 84, 0.9\rangle$, this is less obvious. Indeed, one can consider that such instances have a little structure. This is true for the geom instances by construction, and also for the random instances of the two classes $\langle 40, 80, 103, 0.8\rangle$ and $\langle 40, 180, 84, 0.9\rangle$ since their constraint graph is sparse.

Tables 4 and 5 show the practical interest of LC_1 and LC_2 on structured instances. Table 4 reports results on classical series of academic instances from the literature: graph coloring, job-shop scheduling, quasi-group completion problem, aim and ehi SAT instances converted to CSP. Table 5 reports results on series of instances issued from real-world problems:

- The frequency assignment problem with polarization constraints (FAPP) is an optimization problem that was part of the ROADEF'2001 challenge.[6] In this problem, there are constraints concerning frequencies and polarization of radio links. Progressive relaxation of these constraints is explored: the relaxation level is between 0 (no relaxation) and 10 (maximum relaxation). Progressive relaxation produces eleven CSP instances from any single original FAPP optimization instance.
- The radio link frequency assignment problem (RLFAP) is the task of assigning frequencies to a set of radio links satisfying a large number of constraints and using as few distinct frequencies as possible. In 1993, the CELAR (the French "centre d'electronique de l'armement") built a suite of simplified versions of radio link frequency assignment problems starting from data on a real network [8]. Series of binary RLFAP instances are identified as either scen or graph.
- The Swedish institute of computer science (SICS) has proposed a model of realistic radar surveillance.[7] The problem is to adjust the signal strength (from 0 to s) of a given number of fixed radars w.r.t. six geographic sectors. Each cell of the geographic area of size $p \times p$ must be covered exactly by k radar stations, except for a number i of forbidden cells

[5] http://www.cril.univ-artois.fr/CPAI06/.
[6] http://uma.ensta.fr/conf/roadef-2001-challenge/.
[7] www.ps.uni-sb.de/~walser/radar/radar.html.

Table 6
Results obtained with MAC-LC_k with $k \in [0, 4]$, using *bz* and *dom/wdeg* as heuristics, on academic and real-world instances.

		bz					*dom/wdeg*				
		LC_0	LC_1	LC_2	LC_3	LC_4	LC_0	LC_1	LC_2	LC_3	LC_4
cc-7-7-3	CPU	154	36.1	46.3	42.0	41.7	23.6	30.3	26.6	27.1	30.9
	nodes	732 K	171 K	217 K	198 K	194 K	131 K	174 K	146 K	144 K	165 K
cc-9-9-2	CPU	89.9	4.94	4.68	5.15	5.13	1.01	1.00	0.96	0.98	0.99
	nodes	216*K*	10,823	10,823	10,823	10,823	3387	3457	3457	3457	3457
e0ddr2-1	CPU	*time-out*	*time-out*	463	374	379	110	281	191	272	724
	nodes			3052 K	2471 K	2757 K	812 K	2024 K	1329 K	2085 K	5319 K
enddr1-10	CPU	*time-out*	186	30.0	34.9	23.9	20.4	33.6	31.9	28.1	24.5
	nodes		1472 K	254 K	282 K	210 K	141 K	268 K	261 K	233 K	184 K
fapp02-0250-5	CPU	*time-out*	7.82	7.00	7.23	7.26	7.99	8.80	8.37	8.48	8.57
	nodes		323	323	323	323	851	685	632	638	638
fapp04-0300-5	CPU	*time-out*	753	344	264	315	18.8	9.74	9.67	10.5	14.2
	nodes		97,127	32,394	23,368	28,227	1734	353	332	318	1078
langford-3-12	CPU	25.7	207	383	370	344	23.2	120	129	108	115
	nodes	157 K	1186 K	2086 K	1837 K	1548 K	90,122	441 K	433 K	416 K	384 K
langford-4-12	CPU	6.34	29.2	54.2	61.4	49.4	5.22	17.6	18.6	17.2	16.2
	nodes	18,608	94,941	162 K	172 K	124 K	12,437	36,742	39,112	35,844	33,769
qcp-15-120-12	CPU	*time-out*	*time-out*	611	14.3	94.9	0.55	0.52	0.51	0.52	0.45
	nodes			3206 K	84,304	477 K	782	660	505	531	531
qcp-20-187-11	CPU	*time-out*	*time-out*	*time-out*	*time-out*	*time-out*	3.91	1.46	1.40	1.25	1.13
	nodes						15,992	4992	5083	3860	3753
qa-5	CPU	9.94	3.62	3.13	1.96	3.17	1.42	2.79	2.19	3.1	2.13
	nodes	93,677	31,272	24,995	16,107	22,937	10,533	19,482	14,189	14,990	14,295
qa-6	CPU	*time-out*	290	401	195	420	130	120	143	263	81.4
	nodes		1980 K	2407 K	1217 K	2689 K	769 K	676 K	760 K	1450 K	431 K
graph9-f10	CPU	*time-out*	*time-out*	7.49	5.54	4.22	1.53	1.52	1.49	1.43	1.63
	nodes			14,661	15,520	14,950	2041	2693	2754	2477	3716
scen11	CPU	558	7.80	2.32	1.87	2.61	1.87	7.47	5.90	4.86	2.60
	nodes	2456 K	31,793	5134	4103	5854	4540	35,028	29,465	21,120	4948
ruler-34-9-a3	CPU	24.5	14.2	13.8	15.2	16.8	13.2	9.30	9.57	10.7	12.5
	nodes	18,230	8011	8296	9254	11,066	9144	7740	8647	10,671	12,767
ruler-34-9-a4	CPU	83.8	20.5	35.5	34.2	30.1	16.7	17.5	26.3	24.1	28.9
	nodes	55,129	11,159	22,480	22,908	20,840	8723	9163	15,645	15,714	20,536
tsp-20-366	CPU	12.6	11.3	6.09	6.06	4.61	1.67	1.77	2.32	1.35	1.15
	nodes	26,777	24,013	12,286	10,444	7564	2029	2261	3063	1457	1175
tsp-25-190	CPU	78.0	29.3	213	88.7	272	66.6	118	175	205	64.9
	nodes	147 K	56,091	336 K	133 K	404 K	83,894	133 K	232 K	246 K	83,311

that must not be covered. Sets of 50 instances with non-binary constraints have been generated artificially; each series is denoted by radar-p-k-s-i.

Tables 4 and 5 show that the efficiency of MAC combined with a standard heuristic (i.e. *dom/ddeg*, *bz*) is increased when LC is used, both in terms of CPU time and number of solved instances. LC_2 is even better than LC_1, especially on job-shop and RLFAP series. These instances are structured and a blind search (i.e. without analyzing the reasons of the conflicts) is subject to thrashing. As expected, last-conflict reasoning allows us to reduce the appearance of this phenomenon without modifying the general behavior of the heuristics. When the heuristic *dom/wdeg* is used, the results are less impressive since this heuristic already reduces thrashing.

In Table 6, we can observe the impact of LC on some representative instances from the second constraint solver competition. Results are mentioned for LC_k with k ranging from 0 to 4, and the time limit was 1 hour. Once again, it clearly appears that using LC with a standard heuristic greatly improves the efficiency of the MAC algorithm. This is not always true when the *dom/wdeg* heuristic is used for the reasons previously mentioned. Note that some of these instances cannot be solved efficiently using a backjumping technique such as CBJ or DBT combined with a standard heuristic. This aspect has been shown in [26]. Broadly, LC_2 and LC_3 offer the best trade-off.

We have also focused on the most difficult real-world instances that are currently available (see the results of the second and third constraint solver competitions). These instances are unsatisfiable and belong to the RLFAP series scen11-fX with X $\in [1, 8]$. Figs. 6 and 7 depict the CPU time required to solve these instances using LC_k with k ranging from 0 to 8. Missing points mean that unsatisfiability is not proved within 48 hours. For example, MAC alone (LC_0) with *dom/ddeg* cannot solve any instance of this series within 48 hours. On these difficult structured instances, CPU time generally decreases with increasing values of k. This is particularly true for *dom/ddeg* (see Fig. 6) but still observable with *dom/wdeg* (see Fig. 7).

The overall results obtained on the full suite of instances used for the second constraint solver competition are given in Table 7. Each line of the table corresponds to a category of instances (academic, Boolean, patterned, etc.). For each category, the number given between brackets represents the total number of instances of this category, and we provide the number of solved instances (within 20 minutes) using LC_0 and LC_1 and the heuristics *bz*, *dom/ddeg* and *dom/wdeg*. Whatever the

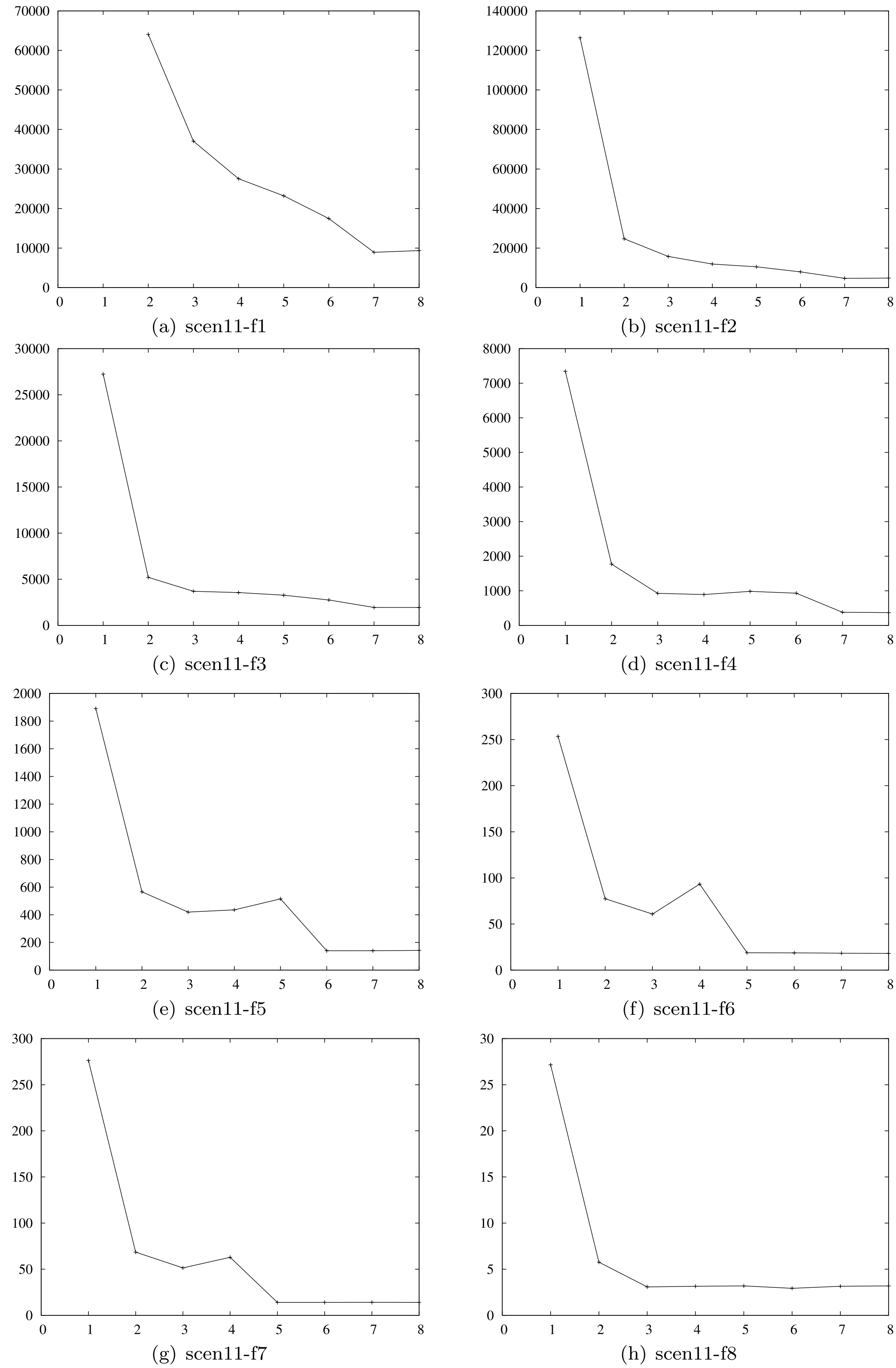

Fig. 6. CPU time (y-axis) to solve the RLFAP instances of series scen11-fX with MAC-LC$_k$, with k (x-axis) ranging from 0 to 8. The variable ordering heuristic is *dom/ddeg* and the time-out to solve each instance is 48 hours.

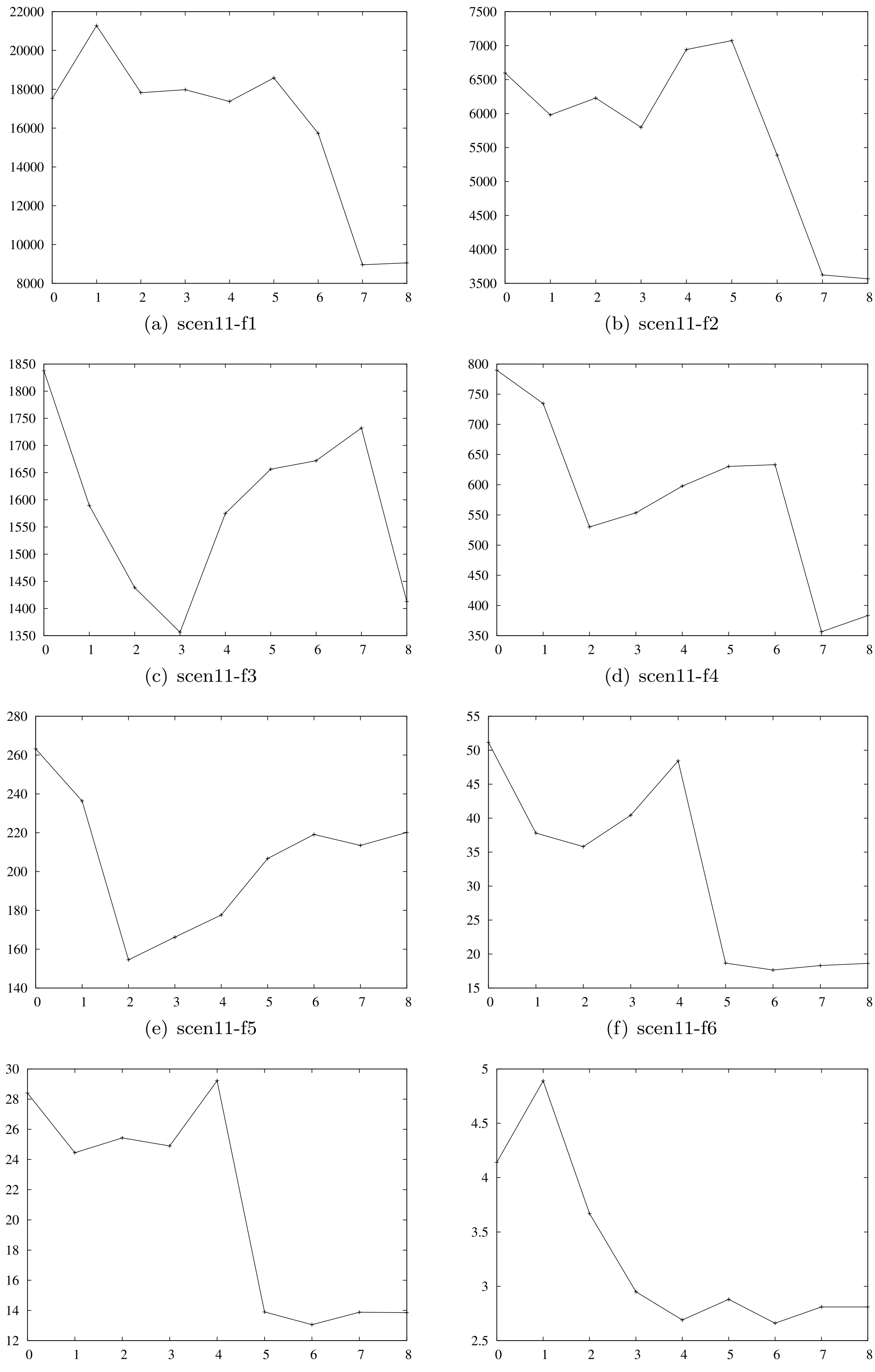

Fig. 7. CPU time (y-axis) to solve the RLFAP instances of series scen11-fX with MAC-LC$_k$, with k (x-axis) ranging from 0 to 8. The variable ordering heuristic is *dom/wdeg* and the time-out to solve each instance is 48 hours.

Table 7
Number of instances from the second constraint solver competition solved within 20 minutes, given by category.

	bz		*dom/ddeg*		*dom/wdeg*	
	LC_0	LC_1	LC_0	LC_1	LC_0	LC_1
Categories of structured instances						
ACAD (#242)	136	146	123	136	132	138
BOOL (#660)	306	336	312	342	388	390
PATT (#846)	379	425	390	431	451	455
QRND (#400)	378	400	290	400	400	400
REAL (#400)	291	319	292	322	326	330
Category of random instances						
RAND (#745)	520	490	535	498	539	493
Total (#3293)	2010	2116	1942	2129	2236	2206

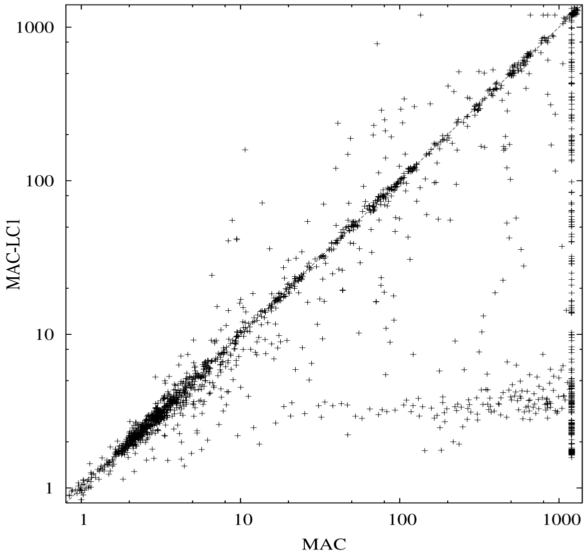

Fig. 8. Pairwise comparison (CPU time) on the 3293 instances used as benchmarks of the second constraint solver competition. The variable ordering heuristic is *dom/ddeg* and the time-out to solve an instance is 20 minutes.

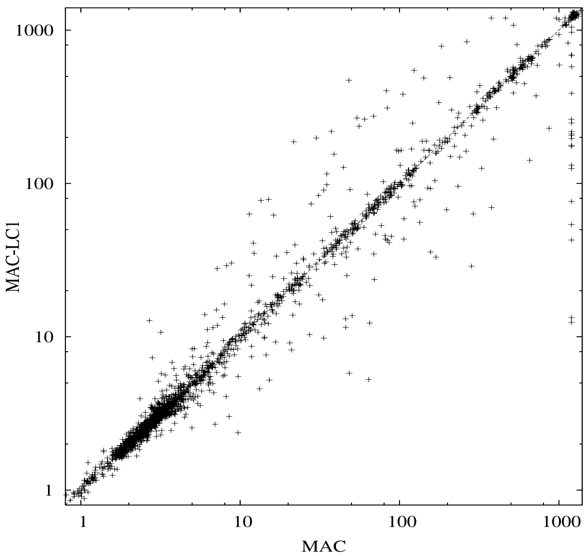

Fig. 9. Pairwise comparison (CPU time) on the 3293 instances used as benchmarks of the second constraint solver competition. The variable ordering heuristic is *dom/wdeg* and the time-out to solve an instance is 20 minutes.

Table 8
Number of instances solved for planning domains (500 instances per domain, time-out is 30 minutes) and total time for instances solved by both.

		CPT		
		LC_0	LC_1	Both
BlocksWorld	#-instances	383	417	383
	CPU	78,333	42,504	–
Depots	#-instances	338	401	338
	CPU	40,606	14,978	–
DriverLog	#-instances	384	439	384
	CPU	64,704	14,613	–
Logistics	#-instances	399	462	399
	CPU	107,552	45,387	–
Rovers	#-instances	347	396	347
	CPU	53,245	26,371	–
Satellite	#-instances	442	464	442
	CPU	63,406	41,149	–

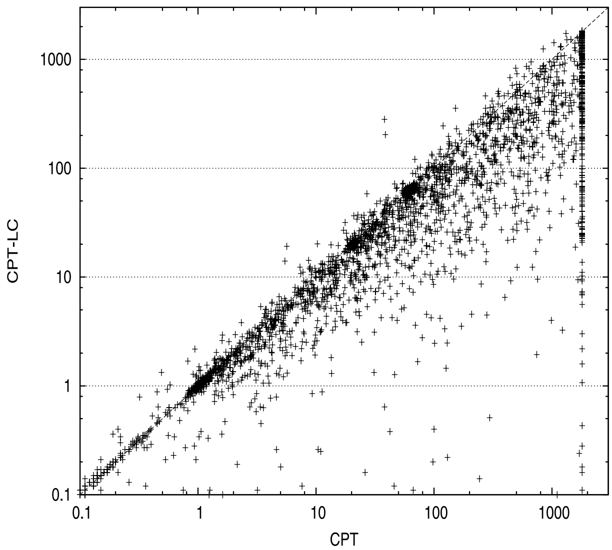

Fig. 10. Pairwise comparison (CPU time) on the 3000 instances from the six planning domains tested in Table 8. The time-out to solve an instance is 30 minutes.

heuristic is used, LC_1 allows to solve more instances than LC_0 on categories of structured instances (Academic, Boolean, Patterned, QuasiRandom and Real). As previously mentioned (see Table 3), LC_1 is not very efficient to solve instances of the random category. Finally, Figs. 8 and 9 depict the same results for *dom/ddeg* and *dom/wdeg* with scatter plots. Each dot represents an instance and its coordinates are defined by: on the horizontal axis, the CPU time required to solve the instance with MAC, and on the vertical axis, the CPU time required to solve the instance with MAC-LC_1. Many dots are located on the right side of the graphs, which means that LC_1 solves more instances than LC_0.

6.2. Results with the optimal temporal planner CPT

Reasoning from last-conflicts can be easily adapted to other research domains. Here we discuss the adaptation of LC_1 to automated Artificial Intelligence planning, more precisely planning using a STRIPS formulation [13,17]. The classical planning problem is the task of determining a sequence of actions (that is to say a plan), allowing the evolution from an initial state of the world to a final state satisfying a set of goals. A state is represented with a set of atoms, called *fluents*. STRIPS actions, classically represented by a triple of sets of fluents — preconditions, *add* effects, *del* effects — can make the current representation of the world evolve from one state to another one. An action can be applied to a state if its preconditions are satisfied into that state, and yields a new state by removing its *del* effects and inserting its *add* effects. Planning problems are defined using a representation language: PDDL [15], which has been developed for the international planning competitions[8] held every two years. The temporal planning problem is an extension of the classical planning paradigm, where each action has a fixed execution time and allows some forms of concurrency between non-conflicting actions.

[8] http://ipc.icaps-conference.org.

Table 9
CPU time (in seconds) required by CPT and CPT-LC_1 to solve instances from the fourth international planning competition.

	CPT	
	LC_0	LC_1
PipesWorld/NoTankage-NonTemporal		
p08-net1-b12-g7	0.58	0.76
p09-net1-b14-g6	174.00	121.00
p13-net2-b12-g3	2.94	5.71
p15-net2-b14-g4	527.96	1450.65
p17-net2-b16-g5	25.44	94.57
p21-net3-b12-g2	466.36	385.90
p24-net3-b14-g5	425.08	159.02
PipesWorld/NoTankage-Temporal-Deadlines-Compiled		
p09-p09-net1-b14-g6-dl	127.29	0.47
p11-p11-net2-b10-g2-dl	–	435.82
p13-p13-net2-b12-g3-dl	–	79.13
p17-p17-net2-b16-g5-dl	189.30	–
Promela/Optical-Telegraph		
p04-opt5	4.21	3.18
p05-opt6	12.58	7.81
p06-opt7	50.84	17.46
p07-opt8	177.78	39.33
p08-opt9	633.42	107.76
p09-opt10	–	277.54
p10-opt11	–	720.61
p11-opt12	–	1740.76
PSR/Small		
p22-s37-n3-l3-f30	48.31	9.11
p31-s49-n4-l2-f30	312.82	282.06
p33-s51-n4-l2-f70	1.04	0.40
p35-s57-n5-l2-f30	1.33	0.69
p46-s97-n5-l2-f30	–	253.37
p47-s98-n5-l2-f50	4.63	1.90
p48-s101-n5-l3-f30	763.24	45.85
Satellite/Time		
p08-pfile8	3.35	1.59
p09-pfile9	1.30	1.06
p10-pfile10	70.56	0.95
p14-pfile14	–	1563.55
p15-pfile15	–	1205.17
p17-pfile17	55.61	62.81
p18-pfile18	12.27	7.49
Satellite/Time-TimeWindows-Compiled		
p04-pfile4	42.66	24.71
p07-pfile7	478.59	365.33
p08-pfile8	7.80	1.14
p09-pfile9	–	0.89
p17-pfile17	103.81	74.99
p18-pfile18	6.82	5.91

The planner CPT [37] is an optimal temporal planning system which combines a branching scheme based on Partial Order Causal Link (POCL) planning with powerful and sound pruning rules implemented as constraints. It minimizes the makespan of the plan, which is the overall execution time of that plan w.r.t. action durations and ordering relations between them. CPT competed in the optimal tracks of the fourth and fifth international planning competitions, where it respectively got a second place and distinguished performance in temporal domains. The key novelty in CPT is its formulation of a planning problem as a constraint satisfaction problem involving the use of supports threats, precedence relations and mutex threats, to deal with actions that are not yet included in a partial plan. The adaptation of last-conflict reasoning (LC_1) to this kind of planning system is quite immediate. The choice for the inclusion of new instances of actions in a partial plan is expressed through support variables $S(p,a)$ associated to couples precondition p – action a, whose domain is the set of actions that can produce the precondition p for the action a. The variable selection heuristic is modified in the same way as in Abscon: the last support variable involved in a conflict is selected in priority as long as a failure is detected.

Table 8 shows the results obtained with CPT on some series of problems from the second and third international planning competitions (domains BlocksWorld, Depots, DriverLog, Logistics, Rovers, Satellite). Some of these domains (Satellite and

Rovers) are also used in the fourth and fifth international planning competitions. Each series contains 500 problems generated using the problem generators implemented for the competitions, with diverse parameters. We have compared standard CPT (noted CPT in the table) with CPT embedding last-conflict reasoning (noted CPT-LC_1 in the table). The time limit was 30 minutes per instance and results have been compared in terms of number of solved instances (#-*intances*) and cumulated CPU time for instances solved by both methods. First note that CPT-LC_1 solves more instances in all problem series. Indeed, broadly, CPT-LC_1 solves 286 instances that CPT cannot solve. Moreover, the total time for solving instances of every series has been greatly improved.

Fig. 10 depicts with a scatter plot the results described above. Each dot represents an instance. The coordinates of this dot are defined by: on the horizontal axis, the CPU time required to solve the instance with CPT and on the vertical axis, the CPU time required to solve the instance with CPT-LC_1. CPT embedding last-conflict reasoning is clearly more efficient than standard CPT. Indeed most of the dots are located under the diagonal, that is to say solving a given instance with CPT-LC_1 is most often faster than with CPT. Moreover, note that many dots are located on the right-hand side of the graph. These dots represent instances solved by CPT-LC_1 but not by CPT.

On instances from the fourth international planning competition,[9] the difference between CPT alone and CPT-LC_1 is generally less significant. Table 9 only provides results on instances for which there is a substantial difference between the two approaches. On these instances, CPT-LC_1 behaves generally better than standard CPT.

7. Conclusion

In this paper, we have introduced the concept of reasoning from last conflicts that can be regarded as an original look-ahead approach which allows to guide search toward sources of conflicts. 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 constraint network cannot be made consistent. This way of reasoning allows to reduce thrashing by backtracking to the most recent identified culprit decision of the last conflict and, as a consequence, simulates a backjumping effect by a form of lazy identification of culprit decisions. A generalization of this reasoning is also proposed, allowing the identification of more relevant culprit decisions (located higher in the search tree). This mechanism computes small sets of hard variables, called testing-sets, that are involved in decisions of the current branch and interleaved with many other irrelevant decisions. Consequently, search is improved by focusing on variables of testing-sets. Our method can be grafted to any search algorithm based on a depth-first exploration without any additional cost in space. The interest of this approach has been shown in practice by an extensive experimentation in both constraint satisfaction and automated artificial intelligence planning.

In our approach, the variable ordering heuristic is violated, until a backtrack to the culprit decision occurs and a singleton consistent value is found for each variable of the testing-set. However, an alternative is not to 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 variables of the testing-set (and so involved in the last conflict). This would be related to the “Quick Shaving” technique [29] whose principle is to check, when a backtrack occurs at depth k, the consistency of values that were shavable (i.e. singleton arc-inconsistent) at depth $k+1$.

Acknowledgements

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, 2000, pp. 35–51.
[2] R.J. Bayardo, R.C. Shrag, Using CSP look-back techniques to solve real-world SAT instances, in: Proceedings of AAAI’97, 1997, pp. 203–208.
[3] C.Bessiere, Constraint propagation, in: Handbook of Constraint Programming, Elsevier, 2006 (Chapter 3).
[4] C. Bessiere, J. Régin, MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems, in: Proceedings of CP’96, 1996, pp. 61–75.
[5] C. Bessiere, J.C. Régin, R. Yap, Y. Zhang, An optimal coarse-grained arc consistency algorithm, Artificial Intelligence 165 (2) (2005) 165–185.
[6] F. Boussemart, F. Hemery, C. Lecoutre, L. Sais, Boosting systematic search by weighting constraints, in: Proceedings of ECAI’04, 2004, pp. 146–150.
[7] D. Brelaz, New methods to color the vertices of a graph, Communications of the ACM 22 (1979) 251–256.
[8] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, J.P. Warners, Radio link frequency assignment, Constraints 4 (1) (1999) 79–89.
[9] X. Chen, P. van Beek, Conflict-directed backjumping revisited, Journal of Artificial Intelligence Research 14 (2001) 53–81.
[10] R. Debruyne, C. Bessiere, Some practical filtering techniques for the constraint satisfaction problem, in: Proceedings of IJCAI’97, 1997, pp. 412–417.
[11] R. Debruyne, C. Bessiere, Domain filtering consistencies, Journal of Artificial Intelligence Research 14 (2001) 205–230.
[12] R. Dechter, Constraint Processing, Morgan Kaufmann, 2003.
[13] R. Fikes, N. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (1971) 189–208.
[14] F. Focacci, M. Milano, Global cut framework for removing symmetries, in: Proceedings of CP’01, 2001, pp. 77–92.
[15] M. Fox, D. Long, PDDL2.1: An extension to PDDL for expressing temporal planning domains, Journal of Artificial Intelligence Research (JAIR) 20 (2003) 61–124.
[16] I.P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, T. Walsh, Random constraint satisfaction: Flaws and structure, Constraints 6 (4) (2001) 345–372.

[9] We have not included results from the fifth and sixth international planning competitions because (1) generators for the fifth do not produce plain STRIPS problems and no generator were available for the sixth, and (2) official instances are designed for suboptimal planners, so we could not get very significant results (instances are either too easy or too difficult).

[17] M. Ghallab, D. Nau, P. Traverso, Automated Planning, Theory and Practice, Morgan Kaufmann, 2004.
[18] M.L. Ginsberg, Dynamic backtracking, Journal of Artificial Intelligence Research 1 (1993) 25–46.
[19] R.M. Haralick, G.L. Elliott, Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence 14 (1980) 263–313.
[20] T. Hulubei, B. O'Sullivan, Search heuristics and heavy-tailed behaviour, in: Proceedings of CP'05, 2005, pp. 328–342.
[21] J. Hwang, D.G. Mitchell, 2-way vs d-way branching for CSP, in: Proceedings of CP'05, 2005, pp. 343–357.
[22] U. Junker, QuickXplain: Preferred explanations and relaxations for over-constrained problems, in: Proceedings of AAAI'04, 2004, pp. 167–172.
[23] N. Jussien, R. Debruyne, P. Boizumault, Maintaining arc-consistency within dynamic backtracking, in: Proceedings of CP'00, 2000, pp. 249–261.
[24] G. Katsirelos, F. Bacchus, Generalized nogoods in CSPs, in: Proceedings of AAAI'05, 2005, pp. 390–396.
[25] C. Lecoutre, Constraint Networks: Techniques and Algorithms, ISTE/Wiley, 2009.
[26] C. Lecoutre, F. Boussemart, F. Hemery, Backjump-based techniques versus conflict-directed heuristics, in: Proceedings of ICTAI'04, 2004, pp. 549–557.
[27] C. Lecoutre, L. Sais, S. Tabary, V. Vidal, Last conflict-based reasoning, in: Proceedings of ECAI'06, 2006, pp. 133–137.
[28] C. Lecoutre, S. Tabary, Abscon 109: A generic CSP solver, in: Proceedings of the 2006 CSP Solver Competition, 2007, pp. 55–63.
[29] O. Lhomme, Quick shaving, in: Proceedings of AAAI'05, 2005, pp. 411–415.
[30] D.G. Mitchell, Resolution and constraint satisfaction, in: Proceedings of CP'03, 2003, pp. 555–569.
[31] S. Ouis, N. Jussien, P. Boizumault, *k*-relevant explanations for constraint programming, in: Proceedings of the Workshop on User-Interaction in Constraint Satisfaction (UICS'02) Held with CP'02, 2002, pp. 109–123.
[32] P. Prosser, Hybrid algorithms for the constraint satisfaction problems, Computational Intelligence 9 (3) (1993) 268–299.
[33] P. Prosser, MAC-CBJ: Maintaining arc consistency with conflict-directed backjumping, Technical report, Department of Computer Science, University of Strathclyde, 1995.
[34] D. Sabin, E.C. Freuder, Contradicting conventional wisdom in constraint satisfaction, in: Proceedings of CP'94, 1994, pp. 10–20.
[35] T. Schiex, G. Verfaillie, Stubborness: A possible enhancement for backjumping and nogood recording, in: Proceedings of ECAI'94, 1994, pp. 165–172.
[36] B.M. Smith, M.E. Dyer, Locating the phase transition in binary constraint satisfaction problems, Artificial Intelligence 81 (1996) 155–181.
[37] V. Vidal, H. Geffner, Branching and pruning: An optimal temporal POCL planner based on constraint programming, Artificial Intelligence 170 (3) (2006) 298–335.
[38] K. Xu, F. Boussemart, F. Hemery, C. Lecoutre, Random constraint satisfaction: Easy generation of hard (satisfiable) instances, Artificial Intelligence 171 (8–9) (2007) 514–534.
[39] K. Xu, W. Li, Exact phase transitions in random constraint satisfaction problems, Journal of Artificial Intelligence Research 12 (2000) 93–103.