Recherche dirigée par le dernier conflit

Christophe Lecoutre **Lakhdar Sais** **Sébastien Tabary** **Vincent Vidal**

CRIL - CNRS FRE 2499
Université d'artois
rue de l'université, SP 16
62307 Lens cedex, France

{lecoutre,sais,tabary,vidal}@cril.univ-artois.fr

Résumé

Dans ce papier, nous proposons une nouvelle approche pour guider la recherche vers la source des conflits. Son principe est le suivant : après chaque conflit, la dernière variable assignée est sélectionnée en priorité tant que le réseau de contraintes est inconsistant. Ceci permet de découvrir la variable coupable la plus récente (i.e. à l'origine de l'échec) en remontant la branche courante de la feuille vers la racine de l'arbre de recherche. Autrement dit, l'heuristique de choix de variables est violée jusqu'au moment où un retour-arrière sur la variable coupable est effectué et que l'on découvre une valeur singleton consistante. En conséquence, ce type de raisonnement, qui représente un moyen original d'éviter le thrashing, peut facilement être intégré à de nombreux algorithmes de recherche. Les expérimentations effectuées sur un large éventail d'instances démontrent l'efficacité de cette approche.

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

Pour résoudre les instances du problème de satisfaction de contrainte (CSP pour Constraint Satisfaction Problem), les algorithmes de recherche arborescente sont couramment utilisés. Pour limiter l'explosion combinatoire inhérente à ce type d'approche, de nombreuses améliorations ont été proposées. Ces améliorations concernent principalement les heuristiques de choix, les techniques de filtrage et l'analyse des conflits, et peuvent être conventionnellement classées entre méthodes prospectives et rétrospectives [9]. Les méthodes *prospectives* sont utilisées lors de la progression de la recherche vers une solution alors que les méthodes *rétrospectives* sont utilisées lors de la régression de la recherche lorsqu'une impasse est rencontrée.

Au début des années 90, l'algorithme prospectif Forward-Checking (FC) [13] combiné avec la technique rétrospective CBJ (pour Conflict-directed BackJumping) [24] était considéré comme l'approche la plus efficace pour résoudre les instances CSP. Cependant, quelques années plus tard, il a été montré que l'algorithme prospectif MAC [25], qui maintient la consistance d'arc durant la recherche était plus efficace que FC-CBJ pour des instances difficiles de grande taille [3].

Ensuite, la situation devint un peu plus confuse. Tout d'abord, [2] ont montré qu'incorporer des techniques rétrospectives développées pour les CSPs (telles que CBJ) à la procédure "Davis-Putnam" permettant de résoudre le problème de satisfiabilité d'une formule propositionnelle (SAT) rendait facile la résolution de nombreuses instances issues de problèmes réels. Ensuite, tandis qu'il était confirmé par des résultats théoriques [7] que plus la phase de progression était sophistiquée, moins il était utile que la phase de régression le soit, certaines expériences sur des problèmes structurés difficiles montraient qu'associer CBJ à MGAC (la version généralisée de MAC aux contraintes non binaires) pouvait présenter encore des améliorations significatives. Enfin, affiner la technique rétrospective [12, 1, 17] en associant une explication à chaque valeur plutôt qu'à chaque variable donne à l'algorithme de recherche une capacité de retour-arrière plus importante. Les résultats empiriques de [1, 17] montrent que MAC peut être surclassé par des algorithmes mettant en oeuvre de telles techniques rétrospectives.

Plus récemment, les techniques prospectives ont repris un certain avantage avec l'introduction de l'heuristique adaptative $dom/wdeg$ [4]. L'idée est d'associer un poids à chaque contrainte et d'incrémenter le poids d'une contrainte chaque fois que celle-ci est violée pendant la recherche. Au fur et à mesure de la recherche, le poids des contraintes le plus souvent impliquées dans des conflits augmente, ce qui permet à l'heuristique de sélectionner les variables apparaissant dans la partie difficile du réseau. Ceci respecte le principe "fail-first" : "Pour réussir, essayons d'abord là ou l'on est susceptible d'échouer" [13]. L'heuristique dirigée par les conflits $dom/wdeg$ est un moyen simple d'éviter le thrashing [4, 14] et est une alternative efficace aux techniques de retours-arrières intelligents [20].

Même si une technique prospective sophistiquée est utilisée, on peut tout de même s'intéresser aux raisons qui ont conduit à un conflit puisque réussir à déterminer l'ordre idéal d'assignation des variables n'est pas envisageable en pratique. En fait, une impasse dans l'arbre de recherche correspond à un conflit entre un sous-ensemble de décisions (assignations de variables) préalablement effectuées. Aussi, est-il pertinent (pour éviter le thrashing) d'identifier la décision la plus récente (que nous appellerons la décision coupable) qui participe au conflit. En effet, une fois que cette décision a été identifiée, il est possible d'effectuer un retour-arrière à la hauteur de celle-ci - c'est le rôle des techniques rétrospectives telles que CBJ et DBT.

Dans ce papier, nous proposons une approche originale pour effectuer (indirectement) un retour-arrière sur la variable coupable de la dernière impasse rencontrée. Pour l'atteindre, la dernière variable assignée ayant provoqué le conflit devient en priorité la prochaine variable à sélectionner tant que les assignations successives l'impliquant rendent le réseau arc inconsistant. Cela correspond alors à vérifier la singleton consistance de cette variable de la feuille à la racine de l'arbre de recherche jusqu'à ce qu'une valeur singleton consistante soit trouvée. En d'autres termes, l'heuristique de choix de variables est violée, jusqu'à ce que l'on atteigne la variable coupable et qu'une valeur singleton consistante soit trouvée. Il est à noter que notre approche est reliée à celle, appelée "quick shaving", proposée par O. Lhomme [21], puisque le principe de cette dernière est de tester lors d'un retour-arrière au niveau k la consistance des valeurs qui étaient singleton arc inconsistantes au niveau $k + 1$. Toutefois, notre approche consiste essentiellement à guider la recherche tandis que le "quick shaving" correspond à un mécanisme d'inférence (plus puissant que la consistance d'arc).

Pour résumer, l'approche que nous proposons a pour objectif de guider la recherche de manière à détecter dynamiquement la raison du dernier conflit rencontré. Il est important de remarquer que, contrairement aux techniques sophistiquées de backjumping, notre approche peut être greffée très facilement sur un algorithme de recherche sans structures de données supplémentaires à gérer.

2 Préliminaires

Définition 1. *Un réseau de contraintes (CN) P est un couple $(\mathscr{X}, \mathscr{C})$ où :*

- *$\mathscr{X}$ est un ensemble fini de n variables tel que chaque variable $X \in \mathscr{X}$ a un domaine associé, noté dom(X), qui contient l'ensemble des valeurs autorisées pour X.*
- *$\mathscr{C}$ est un ensemble fini de e contraintes tel que chaque contrainte $C \in \mathscr{C}$ implique un sous-ensemble de variables de $\mathscr{X}$, noté vars(C), et a une relation associée, notée rel(C), qui contient l'ensemble des tuples autorisés pour les variables de vars(C).*

Une solution d'un réseau de contraintes (CN pour Constraint Network) est une assignation de valeurs à l'ensemble des variables telle que toutes les contraintes soient satisfaites. Un réseau de contraintes est satisfiable ssi il admet au moins une solution. Le problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem), qui consiste à déterminer si un réseau de contraintes donné est satisfiable, est NP-complet. Une instance CSP est alors définie par un réseau de contraintes et le résoudre consiste à trouver une (ou plusieurs) solutions ou alors à prouver son insatisfiabilité. Pour résoudre une instance CSP, on peut modifier le réseau de contraintes en utilisant des méthodes d'inférence ou de recherche [9]. Très souvent, les domaines des variables sont réduits en supprimant des valeurs inconsistantes, i.e. des valeurs qui ne peuvent apparaître dans aucune solution. En effet, il est possible de filtrer les domaines en tenant compte de certaines propriétés du réseau de contraintes. La consistance d'arc (AC) [22] reste la propriété principale.

Définition 2. *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN. Un couple (X, v), avec $X \in \mathscr{X}$ et $v \in dom(X)$, est arc consistant (AC) ssi $\forall C \in \mathscr{C} \mid X \in vars(C)$, il existe un support de (X, v) dans C, i.e., un tuple $t \in rel(C)$ tel que $t[X] = v$ et $t[Y] \in dom(Y)$ $\forall Y \in vars(C)$[1]. P est AC ssi $\forall X \in \mathscr{X}$, dom($X$) $\neq \emptyset$ et $\forall v \in dom(X)$, (X, v) est AC.*

[1] t[X] désigne la valeur dans t pour la variable X.

La singleton arc consistance (SAC) [8] est une consistance plus forte que AC, i.e. SAC peut identifier plus de valeurs inconsistantes que AC. SAC garantit que l'application de la consistance d'arc après toute assignation de variable ne provoque pas d'inconsistance, i.e., ne conduit pas à un domaine vide.

Algorithmes de recherche arborescente L'algorithme de base pour la résolution d'instances CSP est l'algorithme BT. Il utilise une recherche en profondeur d'abord afin d'assigner les variables et un mécanisme de retour-arrière lorsqu'une impasse apparaît. De nombreux travaux se sont focalisés sur l'amélioration des phases de progression et de régression en introduisant des schémas prospectifs et rétrospectifs [9].

MAC [25] est un algorithme prospectif considéré comme l'approche générique la plus efficace pour la résolution d'instances CSP. L'algorithme maintient simplement la consistance d'arc après chaque assignation de variable. Un échec est découvert si le réseau devient inconsistant (à cause d'un domaine vide). Lorsqu'on mentionne MAC, il est important de préciser le schéma de branchement utilisé. En effet on peut considérer un branchement binaire (2-way) ou non binaire (d-way). Ces deux schémas ne sont pas équivalents car il a été démontré que le branchement binaire était plus puissant (notamment pour réfuter l'insatisfiabilité des instances) que le branchement non binaire [15]. En utilisant un schéma de branchement binaire, à chaque étape de la recherche un couple (X,v) est choisi où X correspond à une variable qui n'est pas assignée et v à une valeur dans dom(X). Deux cas sont alors considérés : l'assignation $X = v$ et la réfutation $X \neq v$. L'exécution d'un algorithme MAC (utilisant un branchement binaire) s'apparente alors à la construction d'un arbre binaire. Pendant la recherche (i.e. quand l'arbre de recherche est en train d'être construit), nous pouvons distinguer les noeuds ouverts pour lesquels un seul cas a été considéré et les noeuds fermés, pour lesquels les deux cas ont été considérés. Au niveau des décisions, MAC commence toujours par celles qui sont positives (i.e. les assignations de variable) avant les négatives (i.e. les réfutations de valeur)[2].

Par ailleurs, parmi les principaux algorithmes rétrospectifs, on trouve notamment SBT (Standard Backtracking), CBJ (Conflict Directed Backjumping) [24] et DBT (Dynamic Backtracking) [12]. Le principe de ces algorithmes rétrospectifs est d'effectuer, après chaque échec, un retour-arrière sur l'assignation d'une variable qui doit être reconsidérée. Cette variable est suspectée être la variable coupable la plus récente impliquée dans l'échec. SBT effectue simplement un retour-arrière (chronologique) sur la variable précédemment assignée alors que CBJ et DBT peuvent identifier une décision coupable plus pertinente en exploitant des explications.

Heuristiques de recherche L'ordre dans lequel les variables sont assignées par un algorithme de recherche arborescente est une question incontournable depuis longtemps. L'utilisation de différentes heuristiques de choix de variables pour résoudre une instance CSP peut mener à des résultats très différents en terme d'efficacité. Dans ce papier, nous nous focaliserons sur les heuristiques représentatives suivantes : dom, bz, $dom/ddeg$ et $dom/wdeg$. La plus connue des heuristiques dynamiques, dom [13], consiste à sélectionner à chaque étape de la recherche la variable ayant le domaine le plus petit. Pour départager les variables jugées équivalentes par l'heuristique dom, on peut utiliser le degré courant de la variable : on obtient alors bz [6]. L'heuristique $dom/ddeg$ [3] est obtenue en combinant directement, sous la forme d'un ratio, la taille du domaine avec le degré courant de la variable, ce qui permet d'améliorer substantiellement la performance de la recherche sur certains problèmes. Finalement, en considérant le degré pondéré (ou poids) d'une variable, qui correspond à la somme du poids des contraintes impliquant cette variable (et au moins une autre variable non assignée), on obtient $dom/wdeg$ [4] ; le poids d'une contrainte étant incrémenté chaque fois que la contrainte est impliquée dans un échec.

3 Raisonner à partir des conflits

Dans cette partie, nous montrons qu'il est possible d'identifier un nogood à partir d'une séquence de décisions menant à un conflit, et d'exploiter ce nogood pendant la recherche. Nous discutons ensuite de l'impact de cette approche par rapport au phénomène du thrashing.

3.1 Identification de nogoods

Définition 3. *Soient* $P = (\mathscr{X},\mathscr{C})$ *un* CN *et (*X,v*) un couple tel que* $X \in \mathscr{X}$ *et* $v \in dom(X)$*. L'assignation* $X = v$ *est appelée une décision positive tandis que la réfutation* $X \neq v$ *est appelée décision négative.* $\neg(X = v)$ *représente* $X \neq v$ *and* $\neg(X \neq v)$ *représente* $X = v$.

A partir de maintenant, nous considérerons un opérateur d'inférence ϕ. Cet opérateur peut être employé à n'importe quelle étape d'une recherche, notée ϕ-algorithme de recherche, utilisant un schéma de branchement binaire. Par exemple, MAC correspond à un ϕ_{AC}-algorithme de recherche où ϕ_{AC} est l'opérateur qui permet d'établir AC.

[2] De manière rigoureuse, une réfutation de valeur ne correspond pas à une décision mais représente simplement une conséquence de l'exploration effectuée avec la décision positive.

Définition 4. *Soient P un CN et Δ un ensemble de décisions. $P|_\Delta$ est le CN obtenu à partir de P tel que, pour toute décision positive ($X = v$) $\in \Delta$, dom(X) est restreint à la valeur $\{v\}$, et, pour toute décision négative ($X \neq v$) $\in \Delta$, v est supprimé de dom(X). $\phi(P)$ est le CN obtenu après application de l'opérateur d'inférence ϕ sur P.*

S'il existe une variable avec un domaine vide dans $\phi(P)$ alors P est clairement insatisfiable, noté $\phi(P) = \bot$.

Définition 5. *Soient P un CN et Δ un ensemble de décisions. Δ est un nogood de P ssi $P|_\Delta$ est insatisfiable. Δ est un ϕ-nogood de P ssi $\phi(P|_\Delta) = \bot$. Δ est un ϕ-nogood minimal de P ssi $\nexists \Delta' \subset \Delta$ tel que $\phi(P|_{\Delta'}) = \bot$.*

Il est clair que les ϕ-nogoods sont des nogoods, mais que l'inverse n'est pas garanti. Remarquons que les nogoods sont souvent restreints à des ensembles de décisions positives (i.e. variables assignées). Notre définition inclut les décisions positives et négatives et peut être rapprochée de la définition de nogoods généralisés [10, 18]. Nous pouvons considérer un ϕ-nogood Δ comme déduit d'une séquence de décisions $\langle \delta_1, \ldots, \delta_i \rangle$ telle que $\Delta = \{\delta_1, \ldots, \delta_i\}$. Une telle séquence peut correspondre aux décisions prises le long d'une branche d'un arbre de recherche menant à une impasse.

Définition 6. *Soient P un CN et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions telle que $\{\delta_1, \ldots, \delta_i\}$ est un ϕ-nogood de P. Une décision $\delta_j \in \Sigma$ est dite coupable ssi $\exists v \in dom(X_i) \mid \phi(P|_{\{\delta_1,\ldots,\delta_{j-1},\neg\delta_j,X_i=v\}}) \neq \bot$ où X_i représente la variable impliquée (positivement ou négativement) dans δ_i. Nous définissons la sous-séquence coupable de Σ comme étant soit la séquence vide si aucune décision coupable n'existe, soit la séquence $\langle \delta_1, \ldots, \delta_j \rangle$ où δ_j est la dernière décision coupable de Σ.*

En d'autres termes, la sous-séquence coupable d'une séquence de décisions Σ menant à une inconsistance se termine par la décision la plus récente telle que, quand on considère la négation de celle-ci, il existe une valeur qui peut être assignée à la variable impliquée dans la dernière décision de Σ, sans pour autant aboutir à une inconsistance avec l'opérateur ϕ. Nous pouvons montrer que cela correspond à un nogood.

Proposition 1. *Soient P un CN et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions telle que $\{\delta_1, \ldots, \delta_i\}$ soit un ϕ-nogood de P. L'ensemble des décisions contenues dans la sous-séquence coupable de Σ est un nogood de P*

Preuve. Soit $\langle \delta_1, \ldots, \delta_j \rangle$ une sous-séquence coupable de Σ. Démontrons par récurrence que pour tout entier k tel que $j \leq k \leq i$, l'hypothèse suivante, notée H(k), est vérifiée:

H(k): $\{\delta_1, \ldots, \delta_k\}$ est un nogood

Tout d'abord montrons que H(i) est vrai. Nous savons que $\{\delta_1, \ldots, \delta_i\}$ est un nogood puisque par hypothèse, $\{\delta_1, \ldots, \delta_i\}$ est un ϕ-nogood de P. De plus montrons que pour $j < k \leq i$, si H(k) est vrai alors H($k-1$) est également vrai. Comme $k > j$ et H(k) est vrai, nous savons que par hypothèse de récurrence, $\{\delta_1, \ldots, \delta_{k-1}, \delta_k\}$ est un nogood. De plus, δ_k n'est pas une variable coupable (car $k > j$). En conséquence, grâce à la définition 6, nous savons que $\forall v \in dom(X_i)$, $\phi(P|_{\{\delta_1,\ldots,\delta_{k-1},\neg\delta_k,X_i=v\}}) = \bot$. On en déduit donc que l'ensemble $\{\delta_1, \ldots, \delta_{k-1}, \neg\delta_k\}$ est un nogood. □

La propriété suivante, qui identifie une sous-séquence coupable vide nous permet de prouver l'insatisfiabilité d'un CN.

Corollaire 1. *Soient P un CN et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions telle que $\{\delta_1, \ldots, \delta_i\}$ soit un ϕ-nogood de P. Si la sous-séquence coupable de Σ est vide, alors P est insatisfiable.*

Lorsque l'on obtient un ϕ-nogood à partir d'une séquence de décisions Σ extraite d'une branche construite par un ϕ-algorithme de recherche, on peut revenir à la dernière décision contenue dans la sous-séquence coupable de Σ qui correspond à un nogood. Nous pouvons également remarquer que l'ensemble des décisions de la sous-séquence coupable n'est pas forcément minimal, et peut être mis en relation avec la notion de first Unique Implications Point (1UIP) utilisée dans les solveurs SAT [27].

3.2 Raisonner à partir du dernier conflit

L'identification et l'exploitation des nogoods décrits plus haut peut facilement être intégré dans un ϕ-algorithme de recherche grâce à une simple modification de l'heuristique de choix de variable. Nous appellerons cette approche *Le raisonnement à partir du dernier conflit* (LC).

En pratique, nous n'exploiterons LC que lorsqu'une impasse a été atteinte à partir d'un noeud ouvert de l'arbre de recherche, c'est-à-dire, à partir d'une décision positive. En effet, dans une recherche avec branchement binaire, les décisions positives sont toujours prises en premier lieu. Cela veut dire que LC sera utilisée si et seulement si δ_i (la dernière décision de la séquence mentionné en définition 6) est une décision positive. Pour implémenter LC, il est alors suffisant de (i) enregistrer la variable dont l'assignation a mené directement à une inconsistance, et de (ii) toujours sélectionner en priorité cette variable dans les décisions suivantes plutôt que de considérer les choix proposés par l'heuristique de choix de variables – quelle que soit l'heuristique utilisée. Remarquons également que LC ne nécessite l'utilisation d'aucune structure de données.

La figure 1 illustre le raisonnement à partir du *dernier conflit*. La branche la plus à gauche sur la figure correspond aux décisions positives $X_1 = v_1, \ldots, X_i = v_i$, telles que $X_i = v_i$ mène à un conflit. A ce stade, X_i est enregistré par LC pour une utilisation ultérieure. De plus, v_i est supprimé de $dom(X_i)$, c'est à dire $X_i \neq v_i$. Ensuite, au lieu de poursuivre la recherche avec (potentiellement) une autre variable, X_i est assigné avec une nouvelle valeur v'. Sur notre illustration, ceci mène une nouvelle fois à un conflit, v' est supprimé de $dom(X_i)$, et le procédé est réitéré jusqu'à ce que toutes les valeurs de $dom(X_i)$ soient supprimées, ce qui nous amène à un domaine vide. L'algorithme effectue alors un retour-arrière sur l'assignation $X_{i-1} = v_{i-1}$, et poursuit la recherche dans la branche de droite $X_{i-1} \neq v_{i-1}$. Comme la variable X_i est toujours enregistrée par LC, elle reste sélectionnée en priorité et toutes les valeurs de $dom(X_i)$ sont éliminées suivant le procédé décrit précédemment. L'algorithme aboutit finalement au niveau de la décision $X_j = v_j$, et poursuit sur la branche de droite $X_j \neq v_j$. Puisque X_i est toujours enregistrée, celle-ci est encore selectionnée prioritairement et les valeurs de $dom(X_i)$ considérées. Comme l'une d'entre elles (v) ne mène pas directement à un échec, la recherche peut continuer avec l'assignation $X_i = v$. La variable X_i n'est alors plus enregistrée, et le choix des décisions suivantes est laissé à l'heuristique de choix de variables (et aussi à l'heuristique de choix de valeurs) jusqu'au prochain conflit.

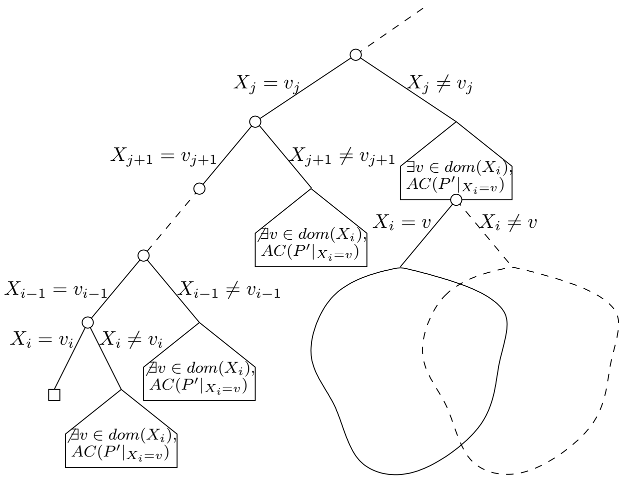

Figure 1: Raisonnement à partir du dernier conflit illustré sur une partie d'un arbre de recherche. P' représente le réseau de contraintes obtenu à chaque noeud après avoir effectué les décisions de la branche courante et appliqué l'opérateur d'inférence ϕ.

En utilisant un opérateur ϕ_{AC} pour identifier la sous-séquence coupable comme décrit ci-dessus, nous obtenons les résultats de complexité suivants.

Proposition 2. *Soient $P = (\mathscr{X}, \mathscr{C})$ un CN et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions telle que $\{\delta_1, \ldots, \delta_i\}$ est un ϕ_{AC}-nogood de P. Le calcul de la sous-séquence coupable de Σ est, dans le pire des cas, $O(eid^3)$, où $e = |\mathscr{C}|$ et $d = |dom(X_i)|$ avec X_i la variable impliquée dans δ_i.*

Preuve. Le pire des cas se produit lorsque la sous-séquence coupable de Σ est vide. Dans ce cas, nous vérifions la singleton arc consistance de X_i pour chaque décision. Vérifier la singleton consistance d'arc d'une variable correspond à appeler d fois l'algorithme de consistance d'arc. Dans le pire des cas, la complexité temporelle est donc id fois la complexité de l'algorithme d'arc consistance utilisé. Comme l'optimalité d'un algorithme établissant la consistance d'arc est $O(ed^2)$ [23], nous obtenons alors une complexité temporelle, dans le pire des cas, en $O(eid^3)$. $\square$

Corollaire 2. *Soient $P = (\mathscr{X}, \mathscr{C})$ un CN et $\Sigma = \langle \delta_1, \ldots, \delta_i \rangle$ une séquence de décisions correspondant à une branche construite par MAC menant à un échec. Dans le pire des cas, la sous-séquence coupable de Σ se calcule en $O(end^3)$, ou $n = |\mathscr{X}|$, $e = |\mathscr{C}|$ et $d = |dom(X_i)|$ avec X_i représentant la variable impliquée dans δ_i.*

Preuve. Tout d'abord, nous savons que les décisions positives sont choisies en premier lieu par MAC, le nombre de noeuds ouverts dans une branche de l'arbre de recherche est donc au plus n. De plus, pour chaque noeud fermé, nous ne devons pas vérifier la singleton consistance de la variable X_i puisque nous devons effectuer un retour-arrière. Donc, nous obtenons $O(end^3)$. $\square$

De manière intéressante, quand LC est greffé à MAC, nous obtenons le résultat suivant.

Corollaire 3. *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN tel que $\exists X \in \mathscr{X} \mid \forall v \in dom(X), \phi(P|_{X=v}) = \bot$. Si on utilise MAC combiné à LC alors dés que X est sélectionné par l'heuristique de choix de variable, la réfutation de P se termine en temps polynomial.*

Évidemment, P peut être vu comme le CN obtenu à chaque noeud de l'arbre de recherche. Pour illustrer ceci, considérons à nouveau la figure 1, et supposons qu'au noeud j, X_j soit singleton inconsistant. Dés que X_j est sélectionnée, la réfutation du sous-arbre dont la racine est le noeud j se termine en $O(end^3)$.

3.3 Prévenir le thrashing avec LC

Le thrashing est le fait d'explorer de façon répétitive les mêmes sous-arbres. Ce phénomène doit être étudié avec soin car un algorithme sujet au thrashing peut être vraiment très inefficace. Parfois, le thrashing peut être expliqué par les mauvais choix effectués au préalable. Pour chaque valeur supprimée du domaine d'une variable, on peut trouver une explication (avec plus ou moins de précision) : il est possible de déterminer les décisions (dans

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

Table 1: Coût de MAC-bz (temps limite fixé à 2 heures)

notre cas, les variables assignées) qui provoquent la suppression de cette valeur. En enregistrant de telles explications et en exploitant ces informations, on peut espérer effectuer un retour-arrière au niveau de la variable coupable, afin d'éviter le thrashing.

Dans certains cas, les techniques de backjumping ne peuvent identifier des variables coupables pertinentes même s'il existe bien un phénomène de thrashing. Par exemple, considérons une instance insatisfiable du problème des reines et cavaliers comme proposé en [4]. Quand les deux sous-problèmes sont fusionnés sans aucune interaction (il n'y a pas de contraintes impliquant à la fois une variable des reines et une variable des cavaliers comme pour l'instance qk-25-25-5-add), une technique de backjumping telle que CBJ ou DBT permet de prouver l'insatisfiabilité du problème à partir de l'insatisfiabilité du sous-problème des cavaliers. Quand les deux sous-problèmes sont fusionnés avec des interactions (une reine et un cavalier ne peuvent pas être placés sur la même case de l'échiquier comme pour l'instance qk-25-25-5-mul), CBJ et DBT deviennent sujets au thrashing (quand elles sont utilisées avec une heuristique de choix de variable classique telle que dom, bz et $dom/ddeg$) car la dernière reine assignée est considérée comme faisant partie des raisons de l'échec. Le problème est que, même s'il existe plusieurs explications pour la suppression d'une valeur, seule la première explication rencontrée est enregistrée. Notons qu'il est également possible de calculer après chaque échec un nogood minimal (sans avoir à maintenir d'explications) avec, par exemple, une technique telle que QuickXplain [16].

Le raisonnement à partir du dernier conflit est un nouveau moyen d'éviter le thrashing tout en étant une technique prospective. En effet, guider la recherche vers la dernière décision d'une sous séquence coupable revient à effectuer une sorte de retour-arrière sur cette décision. Par exemple, nous pouvons montrer que la décision atteinte par un retour-arrière effectué selon la technique de Gaschnig [11] sera au moins atteinte (dans le même contexte) par LC en temps polynomial.

Le tableau 1 nous montre que LC permet d'éviter le phénomène de thrashing sur les deux instances citées ci-dessus. D'un coté, SBT, CBJ et DBT ne peuvent pas éviter le thrashing pour qk-25-25-5-mul puisque l'instance n'est pas résolue en moins de 2 heures (même si on utilise d'autres heuristiques standards). De l'autre, en 1 minute, LC (avec SBT) permet de prouver l'insatisfiabilité de cette instance. La raison est que toutes les variables cavaliers sont singleton arc inconsistantes. Quand une telle variable est atteinte, LC guide la recherche jusqu'à la racine de l'arbre de recherche (cf la proposition 3).

4 Expérimentations

Pour montrer l'intérêt pratique de l'approche décrite dans ce papier, nous avons effectué de nombreuses expérimentations (sur un PC Pentium IV 2,4GHz 512Mo sous Linux). Les performances sont mesurées en termes de nombre de noeuds visités (noeuds) et de temps cpu en secondes (cpu). Notons que lors de nos essais, nous avons utilisé MAC (avec SBT, c'est à dire avec la technique de retour-arrière chronologique), et étudié l'impact de LC avec différentes heuristiques de choix de variables. Nous avons utilisé le solveur abscon (voir `http://www.cril.univ-artois.fr/~lecoutre/`), l'algorithme d'arc consistance AC3.2 [19], et l'heuristique de choix de valeurs $lexico$.

Tout d'abord, nous avons testé plusieurs séries de problèmes. Les résultats obtenus sont indiqués dans le tableau 2. Les instances des séries composées, aléatoires 3-SAT et QWH équilibrées [5] ont été utilisées lors de la première compétition de solveurs CSP [26]. Les instances peuvent être téléchargées à l'adresse `http://cpai.ucc.ie/05/Benchmarks.html`. Chaque instance aléatoire composée possède un fragment principal (ici sous-contraint) et plusieurs fragments auxiliaires, chacun greffé au fragment principal en introduisant des contraintes binaires. Chaque instance aléatoire 3-SAT possède un noyau insatisfiable de petite taille et a été convertie en une instance CSP en utilisant la méthode du “dual encoding”. Chaque instance QWH équilibrée correspond à une instance satisfiable “Quasi-group With Holes”. Finalement, la série des instances surveillance radar a été généré selon le modèle proposé[3] par le Swedish Institute of Computer Science (SICS). Le problème est d'ajuster la force du signal (de 0 à 3) d'un certain nombre de radars couvrant 6 secteurs géographiques sachant que chaque cellule de la

[3] `http://www.ps.uni-sb.de/~walser/radar/radar.html`

	$dom/ddeg$		$dom/wdeg$		bz	
	$\neg LC$	LC	$\neg LC$	LC	$\neg LC$	LC
Instances aléatoires composées (10 instances par série)						
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
Instances aléatoires 3-SAT (100 instances par série)						
ehi-85	1726	2.43	2.21	0.43	1236	1.34
ehi-90	3919	3.17	2.34	0.43	2440	1.37
Instances QWH équilibrées (100 instances par série)						
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)
Instances de surveillance radar (50 instances par série)						
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: Coût moyen (cpu) de MAC sans LC ($\neg LC$) et avec LC sur différentes séries de problèmes

zone géographique doit être couverte par exactement 3 radars, excepté pour certaines cellules qui ne doivent pas être couvertes. Chaque instance est notée rs-i-j où i et j représentent respectivement le nombre de radars et le nombre de cellules ne devant pas être couvertes.

Dans le tableau 2, nous pouvons observer l'impact de LC (en utilisant MAC) sur différentes heuristiques. Notons que le temps limite a été fixé à 1 heure (excepté pour les instances aléatoires 3-SAT, toutes résolues en un temps raisonnable) et que le nombre d'instances expirées (c'est-à-dire le nombre d'instances non résolues en moins d'une heure de temps CPU) est indiqué entre parenthèses. Remarquons également que dans le cas d'instances ayant expiré, le temps CPU indiqué doit être considéré comme une borne inférieure.

Lorsque les heuristiques standards $dom/ddeg$ et bz sont utilisées, il apparaît clairement que LC améliore l'efficacité de MAC en temps CPU ainsi qu'en nombre d'instances résolues, notamment sur les séries composées et surveillance radar. En fait ces instances possèdent une structure telle qu'une recherche mal guidée est sujette au thrashing. L'utilisation de LC évite ce phénomène sans perturber le comportement global des heuristiques. Cependant, quand l'heuristique dirigée par les conflits ($dom/wdeg$) est utilisée, LC ne joue plus un rôle si important car le thrashing est déjà limité par l'heuristique.

Finalement, nous présentons dans le tableau 3 quelques résultats représentatifs obtenus à partir d'instances de la première compétition de solveurs CSP. Le temps limite a été fixé à 1 heure pour les instances académiques et réelles et 10 minutes pour les instances aléatoires.

En ce qui concerne les instances structurées (i.e. les instances académiques et réelles), une fois encore, l'intégration du raisonnement à partir du dernier conflit apporte beaucoup lorsqu'une heuristique standard est utilisée. En effet, même en utilisant une technique de retours-arrières intelligents telle que CBJ et DBT, la plupart de ces instances ne peuvent être résolues rapidement. A ce propos, il suffit de considérer les résultats obtenus dans [20]. D'un autre coté, lorsqu'on utilise l'heuristique MAC-$dom/wdeg$, LC rend la recherche parfois plus efficace et parfois moins efficace.

En ce qui concerne les instances aléatoires, nous avons constaté que LC dégrade le plus souvent les performances quelle que soit l'heuristique de choix de variables utilisée (parmi les 4 référencées dans cet article). Ceci est illustré avec quelques résultats représentatifs donnés dans le tableau 2.

5 Conclusion

Nous avons présenté une approche originale qui peut être intégrée à n'importe quel algorithme de recherche basé sur une exploration en profondeur d'abord. Le principe est de sélectionner en priorité la variable impliquée dans le dernier conflit (c'est-à-dire la dernière assignation menant directement à un conflit) tant que le réseau ne peut être rendu consistant. Cette façon de raisonner permet d'éviter le thrashing en effectuant d'une certaine manière un retour-arrière sur la variable coupable la plus récente impliquée dans le dernier conflit. Cela peut être effectué sans coût

		$dom/ddeg$		$dom/wdeg$		dom		bz	
		$\neg LC$	LC	$\neg LC$	LC	$\neg LC$	LC	$\neg LC$	LC
Instances académiques									
$Golomb$-11-sat	cpu	438	146	584	149	379	134	439	147
	$noeuds$	$55,864$	$19,204$	$51,055$	$12,841$	$58,993$	$21,858$	$60,352$	$21,262$
$BlackHole$-4-4-0010	cpu	3.89	3.60	3.01	5.26	$> 1h$	36.45	$> 1h$	28.50
	$noeuds$	$6,141$	$5,573$	$6,293$	$9,166$	–	$162K$	–	$106K$
cc-10-10-2	cpu	1238	35.63	3.10	4.11	99.95	8.45	1239	35.50
	$noeuds$	$543K$	$14,656$	$2,983$	$3,526$	$180K$	$9,693$	$543K$	$14,656$
qcp-20-$balanced-$23	cpu	$> 1h$	201	17.11	1.00	26.70	2.62	1.11	3.39
	$noeuds$	–	$141K$	$19,835$	$1,210$	$100K$	$6,606$	431	$3,470$
qk-25-25-5-add	cpu	$> 1h$	57.30	135	63.64	$> 1h$	57.72	$> 1h$	57.35
	$noeuds$	–	$10,052$	$24,502$	$11,310$	–	$10,053$	–	$10,053$
qk-25-25-5-mul	cpu	$> 1h$	66.34	134	68.31	$> 1h$	65.64	$> 1h$	66.23
	$noeuds$	–	$9,922$	$22,598$	$9,908$	–	$9,922$	–	$9,922$
Instances réelles									
$e0ddr1$-10	cpu	$> 1h$	87.47	18.85	30.49	102	1.06	$> 1h$	52.11
	$noeuds$	–	$157K$	$37,515$	$56,412$	$307K$	$1,390$	–	$94,213$
$enddr1$-1	cpu	$> 1h$	1.35	0.72	0.57	0.20	0.20	$> 1h$	0.78
	$noeuds$	–	$2,269$	$1,239$	733	50	50	–	$1,117$
$graph2$-$f25$	cpu	$> 1h$	77.86	29.10	3.53	$> 1h$	344	$> 1h$	72.23
	$noeuds$	–	$54,255$	$31,492$	$3,140$	–	$436K$	–	$51,246$
$graph8$-$f11$	cpu	$> 1h$	5.00	7.54	0.49	$> 1h$	57.73	$> 1h$	49.58
	$noeuds$	–	$1,893$	$5,075$	152	–	$41,646$	–	$22,424$
$scen11$	cpu	95.84	1.65	0.97	0.96	$> 1h$	1104	$> 1h$	2942
	$noeuds$	$31,81$	905	911	936	–	$804K$	–	$1672K$
$scen6$-$w2$	cpu	$> 1h$	0.45	0.51	0.29	$> 1h$	0.51	$> 1h$	0.46
	$noeuds$	–	405	741	272	–	706	–	318
Instances aléatoires									
geo-19	cpu	494	$> 10m$	108	491	$> 10m$	$> 10m$	$> 10m$	$> 10m$
	$noeuds$	$338K$	–	74133	$327K$	–	–	–	–
geo-41	cpu	346	$> 10m$	241	$> 10m$	$> 10m$	$> 10m$	377	$> 10m$
	$noeuds$	$237K$	–	$171K$	–	–	–	$264K$	–
$random$-3-$forced$-8	cpu	180	$> 10m$	266	$> 10m$	$> 10m$	$> 10m$	441	$> 10m$
	$noeuds$	$332K$	–	$415K$	–	–	–	$865K$	–
$random$-3-$forced$-9	cpu	357	$> 10m$	$> 10m$	490	$> 10m$	510	$> 10m$	455
	$noeuds$	$782K$	–	–	$776K$	–	$869K$	–	$846K$
$random$-23-48021	cpu	204	459	123	$> 10m$	204	465	203	461
	$noeuds$	$276K$	$618K$	$168K$	–	$276K$	$618K$	$276K$	$618K$
$random$-24-55021	cpu	202	$> 10m$	47	275	197	$> 10m$	205	$> 10m$
	$noeuds$	$245K$	–	$58,919$	$320K$	$245K$	–	$245K$	–

Table 3: Coût de MAC sans LC ($\neg LC$) et avec LC sur des instances académiques, réelles et aléatoires

supplémentaire en espace et avec une complexité temporelle dans le pire des cas en O(end^3) si MAC est utilisé. Les résultats de l'expérimentation que nous avons menée montrent clairement l'intérêt de cette approche pour résoudre des instances structurées lorsqu'une heuristique de choix de variables classique telle que dom, bz et $dom/ddeg$ est utilisée. Avec une heuristique de choix de variables adaptative telle que $dom/wdeg$, les résultats sont plus nuancés.

Avec notre approche, l'heuristique de choix de variables est violée tant que l'on n'a pas effectué un retour-arrière sur la variable coupable et qu'une valeur singleton consistante n'a pas été trouvée. Cependant, il existe une alternative consistant à ne pas considérer la valeur singleton consistante comme la prochaine valeur à assigner. Dans ce cas, l'approche devient une technique d'inférence pure qui correspond à maintenir (partiellement) la singleton consistance (par exemple SAC) sur la variable impliquée dans le dernier conflit. Cela revient à une forme originale de shaving proche de la technique récemment introduite de "Quick Shaving" [21], mais qui se démarque de celle-ci par le fait que les valeurs testées appartiennent toutes à la même variable plutôt qu'à une liste pré-enregistrée.

Remerciements

Ce papier a été soutenu par le programme Cocoa de la Région Nord/Pas-de-Calais et par l'IUT de Lens.

References

[1] F. Bacchus. Extending Forward Checking. In *Proceedings of CP'00*, pages 35–51, 2000.

[2] R.J. Bayardo and R.C. Shrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of AAAI'97*, pages 203–208, 1997.

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

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

[5] F. Boussemart, F. Hemery, and C. Lecoutre. Description and representation of the problems selected for the first international constraint satisfaction problem solver competition. In *Proceedings of CPAI'05 workshop held with CP'05*, pages 7–26, 2005.

[6] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22:251–256, 1979.

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

[8] R. Debruyne and C. Bessière. Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417, 1997.

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

[10] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proceedings of CP'01*, pages 77–92, 2001.

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

[12] M. Ginsberg. Dynamic backtracking. *Artificial Intelligence*, 1:25–46, 1993.

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

[14] T. Hulubei and B. O'Sullivan. Search heuristics and heavy-tailed behaviour. In *Proceedings of CP'05*, pages 328–342, 2005.

[15] J. Hwang and D.G. Mitchell. 2-way vs d-way branching for CSP. In *Proceedings of CP'05*, pages 343–357, 2005.

[16] U. Junker. QuickXplain: preferred explanations and relaxations for over-constrained problems. In *Proceedings of AAAI'04*, pages 167–172, 2004.

[17] N. Jussien, R. Debruyne, and P. Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Proceedings of CP'00*, pages 249–261, 2000.

[18] G. Katsirelos and F. Bacchus. Generalized nogoods in csps. In *Proceedings of AAAI'05*, pages 390–396, 2005.

[19] C. Lecoutre, F. Boussemart, and F. Hemery. Exploiting multidirectionality in coarse-grained arc consistency algorithms. In *Proceedings of CP'03*, pages 480–494, 2003.

[20] C. Lecoutre, F. Boussemart, and F. Hemery. Backjump-based techniques versus conflict-directed heuristics. In *Proceedings of ICTAI'04*, pages 549–557, 2004.

[21] O. Lhomme. Quick shaving. In *Proceedings of AAAI'05*, pages 411–415, 2005.

[22] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[23] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

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

[25] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings of CP'94*, pages 10–20, 1994.

[26] M.R.C. van Dongen, editor. *Proceedings of CPAI'05 workshop held with CP'05*, volume II, 2005.

[27] 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*, pages 279–285, 2001.