Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes : bilan de quelques approches

Gérard Verfaillie* — **Thomas Schiex****

* *CERT-ONERA/DERI*
2 avenue Edouard Belin, BP 4025, 31055 Toulouse Cedex
** *INRA/SBIA*
Chemin de Borde-Rouge, Auzeville, BP 27, 31326 Castanet-Tolosan Cedex

RÉSUMÉ. *Bien que le formalisme et les méthodes CSP soient de plus en plus utilisés pour représenter et traiter des problèmes de Recherche Opérationnelle ou d'Intelligence Artificielle (ordonnancement d'actions, gestion de ressources, configuration et dimensionnement de systèmes, diagnostic de pannes, reconnaissance de situations...), le fait que la plupart de ces problèmes soient des problèmes dynamiques (du fait de l'environnement, de l'utilisateur ou de tout autre agent dans un système distribué) n'a été que très récemment pris en compte. Après une présentation des origines de la dynamicité et des besoins utilisateur (efficacité et stabilité), ce papier fait un tour d'horizon des différentes approches proposées (mémorisation et réutilisation de solutions ou de contraintes), puis décrit six méthodes bâties au dessus des algorithmes* backtrack, repair, local changes, conflict directed backjumping, nogood recording *et* dynamic backtracking, *dont les résultats expérimentaux, obtenus sur des problèmes dynamiques aléatoires, sont analysés.*

ABSTRACT. *Although the CSP formalism and methods are increasingly used for representing and handling* Operations Research *and* Artificial Intelligence *problems (scheduling, resource management, system configuration, diagnosis, situation recognition...), the fact that most of these problems are dynamic (due to the environment, the user or any other agent in a distributed system) has been only recently taken into account. After presenting the origins of this dynamicity and the user requirements (efficiency and stability), we make a survey of various approaches (solution or constraint recording and reusing), and then describe six methods, which are built from the* backtrack, repair, local changes, conflict directed backjumping, nogood recording *and* dynamic backtracking *algorithms and whose experimental results on randomly generated dynamic problems are assessed.*

MOTS-CLÉS *: Problèmes dynamiques de satisfaction de contraintes, Maintien de solution.*

KEY WORDS *: Dynamic constraint satisfaction problems, Solution maintenance.*

1 Problèmes dynamiques de satisfaction de contraintes

1.1 Origine

De nombreux problèmes de RO ou d'IA (ordonnancement d'actions, gestion de ressources, configuration et dimensionnement de systèmes, diagnostic de pannes, reconnaissance de situations …) s'expriment naturellement sous forme de problèmes de satisfaction de contraintes (*Constraint Satisfaction Problems*, *CSP* [TSA 93]), ce qui ouvre la porte à l'utilisation des diverses méthodes de résolution, développées dans ce cadre (filtrage par cohérence locale, recherche systématique ou non). Mais nombre de ces problèmes sont en réalité dynamiques en ce sens que l'ensemble des variables et des contraintes à considérer évolue du fait de l'*environnement*, de l'*utilisateur* ou plus généralement d'autres *agents* :

– dans des problèmes d'*ordonnancement d'actions* ou de *gestion de ressources* (de planification en général) l'ensemble des actions à réaliser peut varier (ajout ou retrait), ainsi que l'état du système physique permettant de les réaliser (pannes); les effets, les durées et les consommations des actions exécutées peuvent ne pas être conformes à la prévision ;

– dans des problèmes de *configuration* ou de *dimensionnement de systèmes* (de conception en général), les spécifications du système à concevoir peuvent varier du fait de l'utilisateur qui désire connaitre les implications de spécifications différentes ou du fait d'entités responsables de la conception d'autres parties ou aspects du système (conception distribuée) ;

– dans des problèmes de *diagnostic de pannes* et, plus généralement, de *reconnaissance de situations*, l'ensemble des observations à expliquer peut varier suite aux changements d'état ou de mode de fonctionnement des différents composants du système à surveiller.

1.2 Formalisation

La notion de *CSP dynamique* (DCSP) a été proposée dans [DEC 88] pour représenter de telles situations : un DCSP est une séquence de CSP dont chaque élément diffère du précédent par l'ajout et/ou le retrait de certaines contraintes. Il est en effet facile de voir que, si les domaines de valeur des variables (contraintes unaires) sont inclus dans l'ensemble des contraintes, l'ensemble des modifications possibles d'un CSP (ajouts ou retraits de contraintes, ajouts ou retraits de variables, modifications de contraintes ou de domaines) peuvent s'exprimer en termes d'ajout et/ou de retrait de contraintes.

À noter que cette définition est différente de celle qui est proposée sous le même nom dans [MIT 90]. Dans [MIT 90], les changements proviennent de la recherche elle-même : les choix faits au cours de la recherche peuvent ajouter ou retirer des variables et/ou des contraintes ; les conditions d'ajout et de retrait sont explicites (contraintes dites d'*activité*, en sus des contraintes classiques dites de *compatibilité*). Dans [DEC 88] et dans le cadre de ce papier, les changements proviennent, soit de l'intérieur, soit de l'extérieur de la recherche, mais aucune information *a priori* sur eux n'est nécessaire.

1.3 Besoins utilisateur

Du point de vue de l'utilisateur, *efficacité de la recherche* et *stabilité des solutions successives* sont les besoins les plus couramment exprimés :

– le besoin de mécanismes de recherche efficaces n'est pas nouveau dans le cadre CSP, mais il peut être plus fort dans le cadre de problèmes dynamiques (ordonnancement d'actions, gestion de ressources, diagnostic de pannes, reconnaissance de situations…) où le temps disponible pour produire une nouvelle solution peut être limité ;

– la stabilité des solutions successives peut être importante (ordonnancement d'actions, gestion de ressources, configuration et dimensionnement de systèmes), au cas où un certain travail (commandes, préparation, organisation…) a pu être engagé sur la base de la solution précédente.

1.4 Problème technique

Du point de vue technique, l'objectif est d'éviter de recommencer la recherche à zéro après chaque modification du problème. La démarche consiste à mémoriser et à réutiliser des résultats obtenus lors de recherches précédentes, qui peuvent être, soit des *solutions* (permises par l'ensemble des contraintes), soit des *contraintes* (déduites des contraintes explicites du problème). On peut tout de suite noter que le retrait de contrainte préserve les solutions, mais ne préserve pas les contraintes, tandis qu'inversement, l'ajout préserve les contraintes, mais ne préserve pas les solutions. D'où l'objectif de développer des méthodes permettant de réutiliser solutions et/ou contraintes, résultats de recherches précédentes, quelque soit la nature du changement (ajouts et/ou retraits de contraintes).

2 Deux approches

D'un point de vue extrêmement global, deux approches distinctes apparaissent dans la littérature suivant que les objets mémorisés et réutilisés sont des solutions ou des contraintes.

2.1 Mémorisation et réutilisation de solutions

La première approche consiste à mémoriser et à réutiliser des solutions de problèmes précédents, sur la base d'une idée extrêmement simple et naïve : si une solution S a été trouvée pour un problème P, si le problème P' est proche du problème P, une solution du problème P' devrait pouvoir, avec de bonnes chances, être trouvée à proximité de S. L'exploitation de cette idée a donné naissance à diverses méthodes qui, à l'exception de la première (*heuristique de choix de la valeur*), ont en commun une capacité à utiliser n'importe quelle affectation de l'ensemble des variables comme point de départ de la recherche. Dans le cadre de problèmes statiques, cette affectation initiale est générée de façon heuristique. Dans le cadre de problèmes dynamiques, une solution précédente, si elle existe, constitue *a priori* une affectation initiale pertinente et seules posent problème les contraintes ajoutées non satisfaites par la solution précédente.

Heuristique de choix de la valeur Une première méthode, la plus simple à mettre en œuvre, consiste à utiliser la solution du problème précédent comme *heuristique* de choix de la valeur dans le cadre de la recherche courante : choisir en premier la valeur que la variable avait dans la solution du problème précédent. Elle a été proposée en appoint de la méthode dite des *oracles* [HEN 91] (cf. 2.3) et les expérimentations montrent de bons résultats, à la fois en termes d'efficacité et de stabilité, sur des problèmes sous-contraints [1] , soumis à des changements peu importants.

Propagation de valeur Une seconde famille de méthodes repose sur la notion de *propagation de valeur* [FB 90, SAN 93]. Elle suppose des contraintes fonctionnelles : existence, pour chaque contrainte c liant un ensemble V_c de variables et pour chaque partition de V_c en deux sous ensembles Vin_c et $Vout_c$, d'une méthode permettant de calculer la valeur des variables de $Vout_c$ en fonction de celle de Vin_c. À condition qu'aucun cycle ne soit rencontré (double modification de la valeur d'une variable), la recherche de solution peut procéder par application successive de méthodes attachées aux contraintes insatisfaites (propagation des modifications de valeur dans l'hypergraphe de contraintes). Mais le besoin de contraintes fonctionnelles et la difficulté de traitement des cycles limitent sérieusement l'intérêt de l'approche.

Réparation heuristique Une autre méthode est proposée sous le nom de *heuristic repair* dans [MIN 92] autour d'une heuristique de *réparation* ou de *minimisation des conflits*, qui consiste à choisir une variable intervenant dans une contrainte insatisfaite et à choisir, pour cette variable, une valeur qui minimise le nombre de contraintes insatisfaites. Une telle heuristique peut être utilisée aussi bien dans le cadre d'une recherche systématique de type *backtrack* (cf. 3.2 et 4.2), que dans le cadre d'une recherche non systématique de type *gradient* ou *recuit simulé* [GHE 93]. On peut noter la proximité entre ces méthodes et les nombreuses méthodes de recherche locale récemment proposées dans le cadre de la logique propositionnelle [SEL 92, GEN 93], à base de déplacements heuristiques et/ou aléatoires. Les expérimentations montrent de très bons résultats, à la fois en termes d'efficacité et de stabilité, sur des problèmes sous-contraints.

1. Dans ce papier, nous distinguons trois classes de CSP aléatoires : les problèmes sous-contraints, intermédiaires et sur-contraints qui se placent respectivement avant, pendant et après la transition de phase [CHE 91]. Les premiers sont aisément satisfaits ; les troisièmes sont clairement insatisfiables ; quant aux deuxièmes, à la frontière entre cohérence et incohérence, ils sont généralement difficiles à résoudre.

Modifications locales Une dernière méthode est proposée dans [VER 94b] sous le nom de *local changes* (cf. 3.3 et 4.3). Elle s'appuie sur un mécanisme de backtrack particulier qu'on peut qualifier de peu destructif. Suite à l'affectation d'une valeur à une variable v, ce mécanisme ne désaffecte que les variables dont l'affectation est incompatible avec celle de v, ne tenant aucun compte de l'ordre d'affectation. À noter que le mécanisme de recherche est systématique et que l'heuristique de minimisation des conflits précédemment évoquée peut être utilisée avec profit. Les expérimentations montrent de très bons résultats sur des problèmes sous-contraints, à la fois en termes d'efficacité et de stabilité, et sur des problèmes sur-contraints, en termes d'efficacité.

2.2 Mémorisation et réutilisation de contraintes

La seconde approche consiste à mémoriser des contraintes produites dans le cadre de recherches précédentes et à réutiliser celles qui restent valides dans le cadre du problème courant pour couper l'espace de recherche associé. Ces contraintes, conséquences des contraintes explicites du problème, sont produites, soit par les mécanismes de filtrage par cohérence locale, soit par le mécanisme de recherche lui-même.

Maintien d'arc cohérence Le filtrage par cohérence locale est largement utilisé pour produire des contraintes, conséquences des contraintes explicites du problème (unaires avec l'arc-cohérence, binaires avec la chemin-cohérence, d'arité $k-1$ avec la k-cohérence) et réduire ainsi l'espace de recherche. En pratique, le filtrage par arc-cohérence est de loin le plus utilisé. C'est pourquoi les méthodes développées dans le cadre dynamique se sont exclusivement intéressées au maintien d'arc-cohérence.

À noter que l'ajout de contrainte est naturellement pris en compte par les algorithmes de filtrage. Seul le retrait pose problème. L'objectif est, dans ce cas, de limiter les réintroductions de valeurs et la méthode générale comprend trois phases : réintroductions de valeurs, propagation des réintroductions et nouveau filtrage. Les méthodes proposées peuvent être classées en deux groupes selon qu'elles utilisent ou non des justifications de retrait de valeur pour limiter les réintroductions futures.

La première approche, qui mémorise, pour chaque valeur ou ensemble de valeurs retiré, la contrainte justifiant ce retrait, a été développée au dessus de AC3 [PRO 92] et de AC4 [BES 91]. La comparaison avec l'exécution des algorithmes statiques correspondants sur chaque nouveau problème montrent des gains appréciables en termes d'efficacité.

La seconde approche, qui n'utilise pas de justifications, mais la définition des contraintes, pour limiter les réintroductions de valeurs, a été développée au dessus de AC3 [BER 94]. Par rapport à l'approche précédente, elle a l'inconvénient d'une moindre efficacité, mais l'avantage de ne requérir la mémorisation d'aucune justification.

Mémorisation et réutilisation de nogooods Une autre méthode, issue de l'approche *learning while searching* [DEC 90], est proposée dans [SCH 93, SCH 94b] sous le nom de *nogood recording* (cf. 3.6 et 4.5). Elle s'appuie sur la mémorisation de contraintes produites au cours d'une recherche classique de type *backtrack*. Ces contraintes, appelées *nogoods*, sont des affectations partielles des variables dont il a été établi qu'elles ne participent à aucune solution. Dans le cadre de problèmes statiques, ces *nogoods* peuvent être utilisés pour couper l'espace de recherche. Dans le cadre de problèmes dynamiques, le fait qu'ils soient créés avec leur justification (un sous-ensemble des contraintes explicites du problème dont ils sont une conséquence) permet de réutiliser ceux dont la justification est inclue dans l'ensemble des contraintes explicites du problème courant et de couper ainsi l'espace de recherche courant. Aussi bien dans le cadre statique que dynamique, les expérimentations montrent d'excellents résultats en termes d'efficacité sur des problèmes aléatoires intermédiaires et sur-contraints.

2.3 Mémorisation et réutilisation de solutions et de contraintes

Les deux approches peuvent évidemment être combinées.

Oracles La méthode dite des *oracles*, proposée dans [HEN 91], s'appuie sur une recherche classique de type *backtrack*. Elle repose sur la mémorisation, pour chaque problème résolu, de l'ensemble des contraintes et de la première solution trouvée. Cette solution, appelée *oracle* peut être vue, soit comme l'expression compacte d'une contrainte (toute affectation des variables la précédant suivant l'ordre de recherche n'est pas solution du problème et de tout problème plus contraint), soit comme une heuristique utilisable dans le cadre de problèmes proches. Ces deux points de vue sont combinés dans une méthode qui a l'avantage de la simplicité, mais l'inconvénient d'imposer un ordre de recherche fixe (ordre d'affectation des variables et de choix des valeurs).

Mémorisation et réutilisation de solutions et d'éliminations de valeurs Une autre méthode est proposée dans [VER 94a] sur la base de l'algorithme appelé *dynamic backtracking* [GIN 93] (cf 3.7 et 4.6). L'algorithme initial, développé dans le cadre des CSP statiques, utilise un mécanisme de *backtrack* particulier, qu'on peut aussi qualifier de peu destructif. Ce mécanisme utilise la notion d'explication (ou justification) d'élimination de valeur : ensemble de variables précédemment affectées responsables de cette élimination. En cas d'élimination de l'ensemble des valeurs du domaine d'une variable v, seule la dernière variable apparaissant dans les explications correspondantes est désaffectée. Dans le cadre des CSP dynamiques, ce mécanisme permet la réutilisation de solution et le fait d'introduire les contraintes du problème dans les explications des éliminations de valeurs (en plus des variables précédemment affectées) permet la réutilisation de ces éliminations, dans une démarche identique à celle du *nogood recording*. Les résultats obtenus sont ceux des méthodes à base de réutilisation de solution sur les problèmes sous-contraints (efficacité et stabilité) et ceux des méthodes à base de réutilisation de contraintes sur les problèmes aléatoires intermédiaires et sur-contraints (efficacité).

3 Des problèmes statiques …

Dans la suite de ce papier, six méthodes, utilisant les algorithmes *backtrack*, *repair*, *local changes*, *conflict directed backjumping*, *nogood recording* et *dynamic backtracking*, sont décrites de façon plus approfondie. La compréhension de leur fonctionnement dans le cadre de problèmes statiques étant un préalable à la compréhension de leur fonctionnement dans le cadre de problèmes dynamiques, nous consacrons cette section à ce premier point et la section suivante au second point.

Notons que ces algorithmes réalisent tous une recherche systématique : ils sont corrects et complets, ce qui signifie qu'ils trouvent une solution si et seulement s'il en existe une. Ils sont décrits en tant qu'algorithmes de recherche d'une seule solution, mais au moins trois d'entre eux (*backtrack*, *conflict directed backjumping* et *nogood recording*) peuvent être facilement transformés en algorithmes de recherche de toutes les solutions. Ils sont décrits sans aucun mécanisme de filtrage, mais tous peuvent être améliorés grâce à l'emploi de n'importe quel mécanisme de ce type (*forward-checking*, *partial*, *full* ou *real full look-ahead* …). Ils sont enfin décrits indépendamment de tout mécanisme de choix (variable à affecter ou désaffecter, valeur à donner à la variable, contrainte à vérifier) et autorisent n'importe quel ensemble d'heuristiques.

Chaque algorithme est décrit informellement sous forme d'un système de transitions : conditions générales satisfaites dans tout état, conditions initiales, transitions (affectation et désaffectation) et condition d'arrêt. Les fonctions essentielles sont données sous la forme d'un pseudo-code de type fonctionnel.

3.1 Backtrack

L'algorithme *backtrack* [GOL 65] est suffisamment connu et utilisé pour que nous n'en faisions pas une présentation détaillée. Son intérêt ici est surtout de constituer une base de référence pour la description des suivants. Rappelons seulement qu'avec cet algorithme :

– dans tout état, l'ensemble des variables est divisé en variables affectées (affectation courante) et variables non affectées ;

– dans l'état de départ, aucune variable n'est affectée (affectation vide) ;

– une transition consiste à étendre l'affectation courante en choisissant une variable v non affectée et en lui affectant une valeur val choisie parmi les valeurs non encore explorées ;

– si l'affectation courante devient de ce fait incohérente, un retour est effectué sur le choix fait pour v ; si toutes les valeurs de v ont été explorées, un retour est effectué sur le choix immédiatement précédent ;

– il y a arrêt si une affectation cohérente de l'ensemble des variables est produite (solution) ou s'il est constaté que l'affectation vide ne peut pas être étendue en une affectation cohérente de l'ensemble des variables (inexistence d'une solution).

Les fonctions de plus haut niveau sont décrites en figure 1. Celles de plus bas niveau répondent aux spécifications suivantes :

– VARIABLES(p) retourne l'ensemble des variables du problème p ;

– VARIABLE-CHOICE(V) retourne une variable de l'ensemble de variables V ;

– DOMAIN(v) retourne l'ensemble des valeurs possibles pour la variable v ;

– VALUE-CHOICE(d) retourne une valeur choisie dans l'ensemble de valeurs d ;

– CONSTRAINTS(v) retourne l'ensemble des contraintes qui pèsent sur la variable v ;

– UNSATISFIED-CONSTRAINT(C, V) retourne $true$ si une contrainte de l'ensemble de contraintes C est insatisfaite du fait de l'affectation de l'ensemble de variables V et retourne $false$ sinon ;

– ASSIGN(v, val) affecte la valeur val à la variable v ;

– UNASSIGN(v) désaffecte la variable v.

3.2 Repair

La différence essentielle entre l'algorithme *backtrack* et l'algorithme *repair*, décrit dans [MIN 92], est que le premier part d'une affectation vide, tandis que le second part d'un affectation complète éventuellement incohérente. Avec l'algorithme *repair* :

– dans tout état, l'ensemble des variables est affecté, mais divisé en variables fixées et variables non fixées ; une variable fixée est une variable dont l'affectation ne peut être modifiée que par *backtrack* ;

– dans l'état de départ, aucune variable n'est fixée ;

– une transition consiste à étendre la partie fixée de l'affectation courante en choisissant une contrainte c insatisfaite par l'affectation courante et une variable v non fixée liée par cette contrainte, en affectant à v une valeur val choisie parmi les valeurs non encore explorées et en fixant v ;

– si la partie fixée de l'affectation courante devient de ce fait incohérente ou s'il est constaté qu'elle ne peut être étendue en une affectation cohérente de l'ensemble des variables, un retour est effectué sur le choix fait pour v ; si toutes les valeurs de v ont été explorées, un retour est effectué sur le choix immédiatement précédent ;

– il y a arrêt si toutes les contraintes sont satisfaites par l'affectation courante (solution) ou s'il est constaté qu'un ensemble vide de variables fixées ne peut pas être étendu en une affectation cohérente de l'ensemble des variables (inexistence d'une solution) ; la première condition implique qu'une solution peut être trouvée

```
BT(p)
    return BT-VARIABLES(∅, VARIABLES(p))

BT-VARIABLES(V1, V2)
; V1 est l'ensemble des variables affectées
; V2 est l'ensemble des variables non affectées
    if V2 = ∅
    then return success
    else let be v = VARIABLE-CHOICE(V2)
         return BT-VARIABLE(V1, V2, v, DOMAIN(v))

BT-VARIABLE(V1, V2, v, d)
    if d = ∅
    then return failure
    else let be val = VALUE-CHOICE(d)
         if BT-VALUE(V1, V2, v, val) = success
         then return success
         else return BT-VARIABLE(V1, V2, v, d − {val})

BT-VALUE(V1, V2, v, val)
    ASSIGN(v, val)
    if UNSATISFIED-CONSTRAINT(CONSTRAINTS(v), V1 ∪ {v})
    then UNASSIGN(v)
         return failure
    else if BT-VARIABLES(V1 ∪ {v}, V2 − {v}) = failure
         then UNASSIGN(v)
              return failure
         else return success
```

Figure 1. Backtrack

à une profondeur comprise entre 0 et n, n désignant le nombre de variables du problème, alors qu'avec l'algorithme *backtrack*, une solution ne peut être trouvée qu'à une profondeur égale à n.

Les fonctions de plus haut niveau sont décrites en figure 2. Celles de plus bas niveau, qui n'ont pas déjà été spécifiées en 3.1, répondent aux spécifications suivantes :

– CONSTRAINTS(p) retourne l'ensemble des contraintes du problème p ;

– INITIALIZE(V) affecte l'ensemble de variables V selon une certaine heuristique ;

– UNSATISFIED-CONSTRAINTS(C, V) retourne l'ensemble des contraintes de C qui sont insatisfaites du fait de l'affectation de l'ensemble de variables V ;

– CONSTRAINT-CHOICE(C) retourne une contrainte choisie dans l'ensemble C de contraintes ;

– FIXED-CONSTRAINT(C, V) retourne *true* si une contrainte de l'ensemble C de contraintes a l'ensemble de ses variables inclus dans l'ensemble de variables V et retourne *false* sinon.

3.3 Local Changes

Avec l'algorithme *local changes* [VER 94b], la résolution d'un CSP ressemble à un jeu d'affectation et de désaffectation, dont le principe de base est le suivant : l'affectation d'une valeur val à une variable v est possible si les variables dont l'affectation est incohérente avec celle de v peuvent être désaffectées et réaffectées sans modifier celle de v. Plus précisément :

– dans tout état, l'ensemble des variables est divisé en trois : variables affectées et fixées, variables affectées et non fixées et variables non affectées ;

– dans l'état de départ, aucune variable n'est affectée, ni fixée ;

– une transition consiste à étendre la partie fixée de l'affectation courante en choisissant une variable v non affectée, en affectant à v une valeur val choisie parmi les valeurs non encore explorées et en fixant v ;

– si la partie fixée de l'affectation courante devient de ce fait incohérente ou si, après avoir désaffecté un sous ensemble des variables affectées et non fixées, tel que l'affectation courante redevienne cohérente, celle-ci ne peut pas être étendue sur les variables qui viennent d'être désaffectées (les variables désaffectées du fait de l'affectation de v ne peuvent être réaffectées après ajout de v à l'ensemble des variables fixées), un retour est effectué sur le choix fait pour v ; si toutes les valeurs de v ont été explorées, un retour est effectué sur le choix immédiatement précédent ;

– si, par contre, l'affectation courante peut être étendue sur les variables qui viennent d'être désaffectées, la variable v devient affectée et non fixée ;

```
RP(p)
    let be V = VARIABLES(p)
    let be C = CONSTRAINTS(p)
    INITIALIZE(V)
    return RP-VARIABLES(∅, V, UNSATISFIED-CONSTRAINTS(C, V))

RP-VARIABLES(V1, V2, C)
; V1 est l'ensemble des variables fixées
; V2 est l'ensemble des variables non fixées
; C est l'ensemble des contraintes insatisfaites
    if C = ∅
    then return success
    else let be c = CONSTRAINT-CHOICE(C)
         let be v = VARIABLE-CHOICE(VARIABLES(c) ∩ V2)
         return RP-VARIABLE(V1, V2, C, v, DOMAIN(v))

RP-VARIABLE(V1, V2, C, v, d)
    if d = ∅
    then return failure
    else let be val = VALUE-CHOICE(d)
         if RP-VALUE(V1, V2, C, v, val) = success
         then return success
         else return RP-VARIABLE(V1, V2, C, v, d − {val})

RP-VALUE(V1, V2, C, v, val)
    let be val′ = VALUE(v)
    ASSIGN(v, val)
    let C′ = CONSTRAINTS(v)
    let C′′ = UNSATISFIED-CONSTRAINTS(C′, V1 ∪ V2)
    if FIXED-CONSTRAINT(C′′, V1 ∪ {v})
    then ASSIGN(v, val′)
         return failure
    else if RP-VARIABLES(V1 ∪ {v}, V2 − {v}, (C − C′) ∪ C′′) = failure
         then ASSIGN(v, val′)
              return failure
         else return success
```

Figure 2. Repair

– il y a arrêt si une affectation cohérente de l'ensemble des variables est produite (solution) ou s'il est constaté qu'un ensemble vide de variables affectées et fixées ne peut pas être étendu en une affectation cohérente de l'ensemble des variables (inexistence d'une solution).

Les fonctions de plus haut niveau sont décrites en figure 3. Celles de plus bas niveau, qui n'ont pas déjà été spécifiées en 3.1 ou en 3.2, répondent aux spécifications suivantes :

– CHOICE-VARIABLES-COVERING(V, C) retourne un sous-ensemble V' de l'ensemble de variables V tel que toute contrainte de l'ensemble de contraintes C ait une de ses variables incluse dans V' ; le choix de V' peut être guidé par n'importe quelle heuristique ;

– SAVE-ASSIGNMENTS(V) sauvegarde l'affectation de l'ensemble de variables V ;

– UNASSIGN(V) désaffecte l'ensemble de variables V ;

– RESTORE-ASSIGNMENTS(V) restaure l'affectation de l'ensemble de variables V ;

3.4 Apprentissage au cours de la recherche

Les trois algorithmes qui suivent (*conflict directed backjumping*, *nogood recording* et *dynamic backtracking*) manipulent implicitement ou explicitement ce qu'on peut appeler des *nogoods*, pour reprendre le terme utilisé dans la communauté *TMS* [DOY 79].

Dans le cadre *CSP*, un *nogood* peut être défini comme une affectation partielle (affectation d'un sous-ensemble des variables du problème) qui ne fait partie d'aucune solution du problème ou, de façon équivalente, comme une affectation partielle interdite par les contraintes du problème. On parle aussi d'*affectation partielle globalement incohérente*, mais nous utiliserons le terme de *nogood* qui a l'avantage de la concision.

Définition 1 *Soit A une affectation d'un sous-ensemble V' des variables du problème; A est un nogood ssi il n'existe pas de solution S telle que $S \downarrow_{V'} = A$* [2].

Notons que toute affectation partielle incohérente est un *nogood*, sans que l'inverse soit vrai.

Le filtrage par cohérence locale [FRE 78] est un première méthode de production de *nogoods*. Il utilise la propriété suivante :

Propriété 1 *Soit A une affectation partielle cohérente ; soit v une variable non affectée par A et $domain(v)$ son domaine ; A est un nogood ssi $\forall val \in domain(v)$, $A \cup \{(v, val)\}$ est un nogood.*

2. Soit A une affectation d'un sous ensemble V des variables du problème et V' un sous-ensemble de V ; $A \downarrow_{V'}$ désigne la restriction de A à V'.

```
LC(p)
    return LC-VARIABLES(∅, ∅, VARIABLES(p))

LC-VARIABLES(V1, V2, V3)
; V1 est l'ensemble des variables affectées et fixées
; V2 est l'ensemble des variables affectées et non fixées
; V3 est l'ensemble des variables non affectées
    if V3 = ∅
    then return success
    else let be v = VARIABLE-CHOICE(V3)
         if LC-VARIABLE(V1, V2, v, DOMAIN(v)) = failure
         then return failure
         else return LC-VARIABLES(V1, V2 ∪ {v}, V3 − {v})

LC-VARIABLE(V1, V2, v, d)
    if d = ∅
    then return failure
    else let be val = VALUE-CHOICE(d)
         if LC-VALUE(V1, V2, v, val) = success
         then return success
         else return LC-VARIABLE(V1, V2, v, d − {val})

LC-VALUE(V1, V2, v, val)
    ASSIGN(v, val)
    let C = UNSATISFIED-CONSTRAINTS(CONSTRAINTS(v), V1 ∪ V2 ∪ {v})
    if C = ∅
    then return success
    else if FIXED-CONSTRAINT(C, V1 ∪ {v})
         then UNASSIGN(v)
              return failure
         else let be V3 = CHOICE-VARIABLES-COVERING(V2, C)
              SAVE-ASSIGNMENTS(V2)
              UNASSIGN(V3)
              if LC-VARIABLES(V1 ∪ {v}, V2 − V3, V3) = failure
              then RESTORE-ASSIGNMENTS(V2)
                   UNASSIGN(v)
                   return failure
              else return success
```

Figure 3. Local changes

Le filtrage par k-cohérence se limite à la production de *nogoods* d'arité $k-1$, qui sont utilisés pour créer ou renforcer des contraintes.

Une seconde méthode de production de *nogoods* a été proposée dans [DEC 90], puis améliorée dans [SCH 93, FRO 94]. Elle s'appuie sur une recherche systématique de type *backtrack* et utilise les deux propriétés suivantes (la seconde étant une simple extension de la propriété 1) :

Propriété 2 *Soit A une affectation partielle ; soit c une contrainte ; soit $V(c)$ l'ensemble des variables contraintes par c ; si c est insatisfaite par A, alors $A \downarrow_{V(c)}$ est un nogood.*

$A \downarrow_{V(c)}$ est en effet une affectation partielle incohérente.

Propriété 3 *Soit v une variable ; soient $val_1, \ldots, val_i, \ldots, val_d$ les valeurs possibles de v ; soient $A_1, \ldots, A_i, \ldots, A_d$ des affectations partielles, restrictions d'une affectation partielle cohérente n'impliquant pas v ; si*

$$A_1 \cup \{(v, val_1)\}, \ldots, A_i \cup \{(v, val_i)\}, \ldots, A_d \cup \{(v, val_d)\}$$

sont des nogoods, alors $A_1 \cup \ldots \cup A_i \cup \ldots \cup A_d$ est un nogood.

Expliciter les contraintes dans les *nogoods* conduit à la définition suivante :

Définition 2 *Soit A l'affectation d'un sous-ensemble V' des variables du problème ; soit C' un sous-ensemble des contraintes du problème ; (A, C') est un nogood ssi il n'existe pas de solution S du CSP réduit à C' telle que $S \downarrow_{V'} = A$.*

Les propriétés 2 et 3 et deviennent :

Propriété 4 *Soit A une affectation partielle ; soit c une contrainte ; soit $V(c)$ l'ensemble des variables contraintes par c ; si c est insatisfaite par A, alors $(A \downarrow_{V(c)}, \{c\})$ est un nogood.*

Afin que l'on puisse prendre en compte explicitement les domaines des variables en tant que contraintes unaires éventuellement supprimables, nous utilisons dans la suite la notation $dom(v)$ pour désigner la contrainte unaire correspondant au domaine de v.

Propriété 5 *Soit v une variable ; soient $val_1, \ldots, val_i, \ldots, val_d$ les valeurs possibles de v ; soient $A_1, \ldots, A_i, \ldots, A_d$ des affectations partielles, restrictions d'une affectation partielle cohérente n'impliquant pas v ; soient $C_1, \ldots, C_i, \ldots, C_d$ des sous-ensembles des contraintes du problème ; soit $dom(v)$ la contrainte unaire correspondant au domaine de v ; si*

$$(A_1 \cup \{(v, val_1)\}, C_1), \ldots, (A_i \cup \{(v, val_i)\}, C_i), \ldots, (A_d \cup \{(v, val_d)\}, C_d)$$

sont des nogoods, alors

$$(A_1 \cup \ldots \cup A_i \cup \ldots \cup A_d, C_1 \cup \ldots \cup C_i \cup \ldots \cup C_d \cup \{dom(v)\})$$

est un nogood.

Comme cela a été noté dans [SMI 88, KLE 89], ces différentes propriétés peuvent être vues comme des particularisations et des applications au cadre *CSP* de la règle d'*hyper-résolution* suivante, valide en logique propositionnelle :

$$\{L_1 \vee \neg M_1, \ldots, L_d \vee \neg M_d, M_1 \vee \ldots \vee M_d \vee M\} \models L_1 \vee \ldots \vee L_d \vee M$$

Il suffit pour cela de remplacer chaque L_i par $(\neg C_i \vee \neg A_i)$, chaque M_i par $(v = val_i)$ et M par $dom(v)$, ce qui garantit la validité des propriétés énoncées.

Même si les versions de base des algorithmes tels que *conflict directed backjumping* et *dynamic backtracking* n'utilisent que les propriétés 2 et 3, nous décrivons les trois algorithmes *conflict directed backjumping*, *nogood recording* et *dynamic backtracking* de façon uniforme, sur la base des propriétés 4 et 5. L'explicitation des contraintes dans les nogoods a en effet le double avantage de permettre l'explication des incohérences (ensemble de contraintes responsables) et de faciliter le traitement des problèmes dynamiques.

3.5 Conflict Directed Backjumping

L'algorithme appellé *conflict directed backjumping* [PRO 93] est une amélioration de l'algorithme *backtrack* (cf. 3.1). Cette amélioration porte sur le mécanisme de retour dans le cas où toutes les possibilités d'extension d'une affectation courante sur une variable v ont été explorées. Alors que l'algorithme *backtrack* revient systématiquement sur le choix immédiatement précédent, l'algorithme *conflict directed backjumping*, s'appuyant implicitement sur la notion de *nogood*, revient sur la dernière variable, selon l'ordre d'affectation, dont l'affectation est un élément de l'explication de l'échec constaté sur v.

Dans la version de base, présentée dans [PRO 93], une explication d'échec est un sous-ensemble de l'affectation courante (utilisation de la définition 1 et des propriétés 2 et 3). Dans la version que nous présentons, c'est un sous-ensemble des contraintes (utilisation de la définition 2 et des propriétés 4 et 5). En termes de *backjumping*, ces deux versions sont équivalentes : soit A l'affectation courante ; soit C' un sous-ensemble des contraintes du problème suffisant à expliquer l'échec de l'extension de A en une affectation cohérente de l'ensemble des variables ; soit $V(C')$ l'ensemble des variables liées par les contraintes de C' ; la projection de A sur $V(C')$ est un sous-ensemble de l'affectation courante suffisant à expliquer l'échec. Mais la seconde version est plus riche en termes d'explication proprement dite. L'explication d'un échec global (échec de l'extension de l'affectation vide en une affectation cohérente de l'ensemble des variables) fournit en particulier un sous-ensemble C' de l'ensemble des contraintes responsable de l'incohérence du problème. On peut alors affirmer que tout problème dont l'ensemble des contraintes contient C' est lui aussi incohérent.

Les fonctions de plus haut niveau sont décrites en figure 4. Celles de plus bas niveau, qui n'ont pas déjà été spécifiées en 3.1, répondent aux spécifications suivantes :

– VARIABLES(C) retourne l'ensemble des variables liées par les contraintes de C ;

– NOT-MEMBER(v, V) retourne $true$ si la variable v n'appartient pas à l'ensemble de variables V et $false$ sinon ;

– FIRST-UNSATISFIED-CONSTRAINT(C, V) retourne l'ensemble constitué de la première contrainte de l'ensemble de contraintes C insatisfaite du fait de l'affectation de l'ensemble de variables V, dans le cas où une telle contrainte existe, et retourne l'ensemble vide, dans le cas où une telle contrainte n'existe pas.

À noter qu'alors que les algorithmes précédents (*backtrack*, *repair* et *local changes*) retournent *success* ou *failure*, cet algorithme et les suivants (*nogood recording* et *dynamic backtracking*) retournent, grâce aux propriétés 4 et 5, soit l'ensemble vide synonyme de succès, soit un sous-ensemble de l'ensemble de contraintes responsable de l'échec.

3.6 Nogood Recording

L'algorithme *nogood recording* [SCH 93, SCH 94b] peut être vu comme une simple amélioration de l'algorithme *conflict directed backjumping* (cf. 3.5). Cette amélioration consiste à mémoriser les *nogoods* produits au cours de la recherche et à les réutiliser pour couper l'espace de recherche restant.

Comme dans le cas de l'algorithme *conflict directed backjumping*, la version que nous présentons utilise explicitement la définition 2 et les propriétés 4 et 5. Les *nogoods* produits sont ajoutés aux contraintes sous forme de n-uplets interdits, accompagnés de l'ensemble des contraintes initiales du problème qui les justifient. Au cas où un *nogood* lie des variables qui ne sont pas liées par une contrainte existante, une contrainte doit être créée (modification du graphe de contraintes). Le nombre de *nogoods* produits au cours d'une recherche pouvant croître exponentiellement en fonction de la taille du problème, une limitation pratique, analogue à celle qui est utilisée pour le filtrage par cohérence locale, consiste à limiter l'arité des *nogoods* mémorisés.

Les fonctions de plus haut niveau sont décrites en figure 5. À noter que, par rapport à l'algorithme *conflict directed backjumping* décrit en 3.5, seule la fonction NR-VARIABLE est modifiée, par ajout de la ligne CREATE-NOGOOD$(V_1 \cup \{v\}, C')$:

– CREATE-NOGOOD(V, C) ajoute à la contrainte c liant l'ensemble de variables V le couple constitué :

  – du n-uplet interdit correspondant à la projection de l'affectation des variables de V sur l'ensemble des variables liées par les contraintes de C ;

  – de l'ensemble de contraintes C ;

  au cas où la contrainte c n'existe pas, elle est créée avant ajout du *nogood* ;

– la fonction FIRST-UNSATISFIED-CONSTRAINT doit être modifiée par rapport à la spécification précédente (cf. 3.5) ; FIRST-UNSATISFIED-CONSTRAINT(C, V) retourne ici la justification de l'insatisfaction de la première contrainte de C qui est insatisfaite du fait de l'affectation de l'ensemble de variables V, dans le cas où une telle contrainte existe ; elle retourne l'ensemble vide, dans le cas où une telle contrainte n'existe pas ; si une contrainte c est insatisfaite du fait de sa définition initiale, la justification de cette insatisfaction est l'ensemble constitué

```
CBJ(p)
    return CBJ-VARIABLES(∅, VARIABLES(p))

CBJ-VARIABLES(V1, V2)
; V1 est l'ensemble des variables affectées
; V2 est l'ensemble des variables non affectées
    if V2 = ∅
    then return ∅
    else let be v = VARIABLE-CHOICE(V2)
        return CBJ-VARIABLE(V1, V2, ∅, v, DOMAIN(v))

CBJ-VARIABLE(V1, V2, C, v, d)
    if d = ∅
    then return C ∪ {dom(v)}
    else let be val = VALUE-CHOICE(d)
        let be C' = CBJ-VALUE(V1, V2, v, val)
        if C' = ∅
        then return ∅
        else if NOT-MEMBER(v, VARIABLES(C'))
            then return C'
            else return CBJ-VARIABLE(V1, V2, C ∪ C', v, d − {val})

CBJ-VALUE(V1, V2, v, val)
    ASSIGN(v, val)
    let be C = FIRST-UNSATISFIED-CONSTRAINT(CONSTRAINTS(v), V1 ∪ {v})
    if C ≠ ∅
    then UNASSIGN(v)
        return C
    else let be C' = CBJ-VARIABLES(V1 ∪ {v}, V2 − {v})
        if C' ≠ ∅
        then UNASSIGN(v)
            return C'
        else return ∅
```

Figure 4. Conflict directed backjumping

de la contrainte elle-même ; si elle est insatisfaite du fait d'un *nogood* préalablement produit et mémorisé, la justification est l'ensemble de contraintes associé à ce *nogood*.

3.7 Dynamic Backtracking

L'algorithme *dynamic-backtracking* [GIN 93] propose un autre mécanisme de retour dans le cas où toutes les possibilités d'extension d'une affectation courante sur une variable ont été explorées. Soit V' l'ensemble des variables affectées, ordonné selon l'ordre d'affectation ; soit v la variable sur laquelle l'échec de toute extension a été constaté ; soit v' la dernière variable de V' ; soit v'' la dernière variable de V' dont l'affectation est un élément de l'explication de l'échec constaté sur v. Alors que l'algorithme *backtrack* revient systématiquement sur la variable v', alors que l'algorithme *conflict directed backjumping* revient directement sur la variable v'', mais, ce faisant, désaffecte toutes les variables suivant v'' dans V', l'algorithme *dynamic-backtracking* revient sur la variable v'' sans désaffecter les variables suivant v'' dans V'.

Le mécanisme de retour utilisé dans les algorithmes *backtrack* et *conflict directed backjumping* peut être qualifié de statique, dans la mesure où les variables sont désaffectées dans l'ordre inverse de leur ordre d'affectation. Le mécanisme utilisé dans l'algorithme *dynamic-backtracking* peut, lui, être qualifié de dynamique dans la mesure où les variables ne sont pas systématiquement désaffectées dans l'ordre inverse. À noter que l'espace de recherche résultant n'est plus un arbre, mais un graphe.

Pour ce faire, l'algorithme *dynamic-backtracking* utilise la notion d'*eliminating explanation*, qui est une variante de la notion de *nogood*.

Propriété 6 *Soit v une variable ; soit val une de ses valeurs ; soit A une affectation partielle n'impliquant pas v ; A est une eliminating explanation pour la valeur val de v ssi $A \cup (v, val)$ est un nogood.*

À noter qu'à un *nogood* d'arité n, correspond n *eliminating explanations* qui se différencient uniquement par le choix de la variable apparaissant en conclusion.

Comme nous l'avons fait avec l'algorithme *conflict directed backjumping*, nous modifions la version de base de l'algorithme de façon à faire apparaître les contraintes dans les explications. Dans la version de base, présentée dans [GIN 93], une *eliminating explanation* est un sous ensemble de l'affectation courante (utilisation de la définition 1 et des propriétés 2 et 3). Dans la version que nous présentons, c'est un sous-ensemble de contraintes (utilisation de la définition 2 et des propriétés 4 et 5), ce qui ne modifie pas le mécanisme de retour, mais fournit une information plus riche.

Avec l'algorithme *dynamic backtracking* :

– de façon générale, l'ensemble des variables est divisé en variables affectées (affectation courante) et variables non affectées ;

– dans l'état de départ, aucune variable n'est affectée (affectation vide) ;

```
NR(p)
    return NR-VARIABLES(∅, VARIABLES(p))

NR-VARIABLES(V1, V2)
; V1 est l'ensemble des variables affectées
; V2 est l'ensemble des variables non affectées
    if V2 = ∅
    then return ∅
    else let be v = VARIABLE-CHOICE(V2)
         return NR-VARIABLE(V1, V2, ∅, v, DOMAIN(v))

NR-VARIABLE(V1, V2, C, v, d)
    if d = ∅
    then return C ∪ {dom(v)}
    else let be val = VALUE-CHOICE(d)
         let be C′ = NR-VALUE(V1, V2, v, val)
         if C′ = ∅
         then return ∅
         else if NOT-MEMBER(v, VARIABLES(C′))
              then return C′
              else CREATE-NOGOOD(V1 ∪ {v}, C′)
                   return NR-VARIABLE(V1, V2, C ∪ C′, v, d − {val})

NR-VALUE(V1, V2, v, val)
    ASSIGN(v, val)
    let be C = FIRST-UNSATISFIED-CONSTRAINT(CONSTRAINTS(v), V1 ∪ {v})
    if C ≠ ∅
    then UNASSIGN(v)
         return C
    else let be C′ = NR-VARIABLES(V1 ∪ {v}, V2 − {v})
         if C′ ≠ ∅
         then UNASSIGN(v)
              return C′
         else return ∅
```

Figure 5. Nogood recording

– une transition consiste à étendre l'affectation courante en choisissant une variable v non affectée et en lui affectant une valeur val choisie parmi les valeurs non encore explorées ;

– si l'affectation courante devient de ce fait incohérente (insatisfaction d'une contrainte c), la valeur val est retirée du domaine de v, une *eliminating explanation*, contenant uniquement c, est créée pour cette valeur et un autre choix est tenté pour v.

– si toutes les valeurs de v ont été explorées, si C' est l'union des *eliminating explanations* de l'ensemble des valeurs de v, si $V(C')$ est l'ensemble des variables liées par les contraintes de C' et si v' est la dernière variable affectée appartenant à $V(C')$, si val' est sa valeur courante, alors :

  – la variable v' est désaffectée ;
  – la valeur val' est retirée du domaine de v' ;
  – une *eliminating explanation*, contenant C', est créée pour cette valeur ;
  – toutes les *eliminating explanations* de contenu C'' telles que v' est liée par une contrainte de C'' sont détruites (la désaffectation de v' les rend invalides) ;
  – les valeurs correspondantes sont réintroduites dans les domaines ;
  – un autre choix est enfin tenté pour v' ;

– il y a arrêt si une affectation cohérente de l'ensemble des variables est produite (solution) ou si, toutes les valeurs ayant été explorées pour une variable v et C' étant l'union des *eliminating explanations* de l'ensemble des valeurs de v, il n'y a pas d'intersection entre l'ensemble des variables liées par les contraintes de C' et l'ensemble des variables affectées (retour inutile, incohérence du problème justifiée par C').

Les fonctions de plus haut niveau sont décrites en figure 6. Celles de plus bas niveau, qui n'ont pas déjà été spécifiées en 3.1 ou en 3.5, répondent aux spécifications suivantes :

– LAST-MEMBER(V, V') retourne la dernière variable de l'ensemble ordonné de variables V qui appartienne à l'ensemble de variables V' ;

– ELIMINATING-EXPLANATIONS(v) retourne l'union des *eliminating explanations* des valeurs du domaine de la variable v ;

– CREATE-ELIMINATING-EXPLANATION(v, val, C) retire la valeur val du domaine courant de la variable v et crée pour cette valeur une *eliminating explanation* contenant l'ensemble de contraintes C ;

– REMAINDER(V, v) retourne l'ensemble des variables suivant la variable v dans l'ensemble ordonné de variables V ;

– REMOVE-ELIMINATING-EXPLANATION-VARIABLE(V, v) détruit toutes les *eliminating explanations* des variables de l'ensemble de variables V, contenant un ensemble de contraintes C telles que la variable v est liée par une contrainte de C, puis réintroduit les valeurs correspondantes dans leurs domaines respectifs.

3.8 Quelques combinaisons possibles

L'existence de ces mécanismes ouvre la porte à de multiples combinaisons :

– nous avons déjà signalé que les six algorithmes, qui viennent d'être décrits sans aucun mécanisme de filtrage (avec un simple *backward-checking*), peuvent être améliorés grâce à l'emploi de n'importe quel mécanisme de ce type (*forward-checking*, *partial*, *full* ou *real full look-ahead* …) ; pour le *forward checking* par exemple, il suffit de noter que le filtrage classiquement réalisé depuis les variables affectées sur les variables non affectées doit, en ce qui concerne *repair* et *local changes*, être réalisé depuis les variables fixées sur les variables non fixées ;

– la même démarche est possible avec le *nogood recording* ; l'algorithme que nous avons présenté sous ce nom est en fait une amélioration du *conflict directed backjumping*, par la mémorisation et la réutilisation des *nogoods* produits au cours de la recherche ; la même amélioration est possible à partir de *repair*, de *local changes* ou de *dynamic backtracking* ;

– *repair*, enfin, est une modification du *backtrack*, consistant à réparer une affectation complète plutôt qu'à la construire à partir de zéro ; la même modification est possible à partir de *conflict directed backjumping*, de *nogood recording* ou de *dynamic backtracking* ; cette dernière combinaison (*repair* + *dynamic backtracking*) est d'ailleurs présentée dans [GIN 94].

Repair + *forward-checking* + *conflict directed backjumping* , *repair* + *partial look ahead* + *dynamic backtracking*, *local changes* + *arc-consistency* + *nogood recording* …L'espace des combinaisons est grand. Si on y ajoute toutes les heuristiques possibles pour le choix de la variable, de la valeur ou de la contrainte, il devient immense.

4 …Aux problèmes dynamiques

Il est maintenant possible de décrire le comportement de ces six algorithmes (*backtrack*, *repair*, *local changes*, *conflict directed backjumping*, *nogood recording* et *dynamic backtracking*) dans le cadre de problèmes dynamiques. Alors qu'il s'agissait précédemment de résoudre un problème p, il s'agit maintenant, ayant résolu un problème p, de résoudre un nouveau problème p' dérivé de p par l'ajout d'un ensemble de contraintes C_a et le retrait d'un ensemble de contraintes C_r.

Comme précédemment, dans le cadre de problèmes statiques, chaque algorithme est décrit sous la forme d'un pseudo-code de type fonctionnel. Deux cas sont, chaque fois, considérés : celui où le problème précédent est cohérent (existence d'une solution)

```
DBT(p)
    return DBT-VARIABLES(∅, VARIABLES(p))

DBT-VARIABLES(V1, V2)
; V1 est l'ensemble ordonné des variables affectées
; V2 est l'ensemble des variables non affectées
    if V2 = ∅
    then return ∅
    else let be v = VARIABLE-CHOICE(V2)
         let be C = DBT-VARIABLE(V1, v, DOMAIN(v))
         if C = ∅
         then return DBT-VARIABLES(V1 ∪ {v}, V2 − {v})
         else let be v' = LAST-MEMBER(V1, VARIABLES(C))
              if v' = ∅
              then return C
              else DBT-BACKWARD(V1, V2, v', C)
                   return DBT-VARIABLES(V1 − {v'}, V2 ∪ {v'})

DBT-VARIABLE(V1, v, d)
    if d = ∅
    then let be C = ELIMINATING-EXPLANATIONS(v)
          return C ∪ {dom(v)}
    else let be val = VALUE-CHOICE(d)
         if DBT-VALUE(V1, v, val) = ∅
         then return ∅
         else return DBT-VARIABLE(V1, v, d − {val})

DBT-VALUE(V1, v, val)
    ASSIGN(v, val)
    let be C = FIRST-UNSATISFIED-CONSTRAINT(CONSTRAINTS(v), V1 ∪ {v})
    if C ≠ ∅
    then UNASSIGN(v)
          CREATE-ELIMINATING-EXPLANATION(v, val, C)
          return C
    else return ∅

DBT-BACKWARD(V1, V2, v, C)
    let be V3 = REMAINDER(V1, v)
    REMOVE-ELIMINATING-EXPLANATIONS-VARIABLE(V3 ∪ V2, v)
    CREATE-ELIMINATING-EXPLANATION(v, VALUE(v), C)
    UNASSIGN(v)
```

Figure 6. Dynamic backtracking

et celui où il est incohérent (inexistence d'une solution). Pour chaque algorithme, nous insistons sur sa capacité à réutiliser les résultats obtenus dans le cadre de la recherche précédente.

4.1 Backtrack

L'algorithme *backtrack* ne réutilise que peu de choses de la recherche précédente. Hormis deux cas très particuliers :

- le problème précédent est cohérent et toutes les contraintes ajoutées sont satisfaites par la solution précédente ;
- le problème précédent est incohérent et aucune contrainte n'est retirée ;

la recherche courante repart d'une affectation vide et peut, seulement au cas où le problème précédent est cohérent, utiliser la solution précédente comme heuristique de choix de la valeur.

Le pseudo-code est décrit en figure 7. Les fonctions de bas niveau, qui n'ont pas déjà été spécifiées en 3.1, répondent aux spécifications suivantes :

- RESULT(p) retourne le résultat obtenu sur le problème p : *success* ou *failure* ;
- ADD-CONSTRAINTS(p, C) ajoute les contraintes C au problème p ;
- REMOVE-CONSTRAINTS(p, C) retire les contraintes C du problème p ;
- ALL-SATISFIED-CONSTRAINTS(C, V) retourne *true* si toutes les contraintes de l'ensemble de contraintes C sont satisfaites du fait de l'affectation de l'ensemble de variables V et retourne *false* sinon.

4.2 Repair

Nous avons noté en 3.2, dans le cadre de problèmes statiques, que la différence entre les algorithmes *backtrack* et *repair* est que le premier part d'une affectation vide, alors que le second part d'une affectation complète éventuellement incohérente. Dans le cadre de problèmes dynamiques, ceci induit un comportement différent au cas où le problème précédent est cohérent : la solution précédente peut être utilisée comme point de départ de la recherche courante et non plus seulement comme heuristique de choix de la valeur. Le comportement est par contre identique au cas où le problème précédent est incohérent : sauf dans le cas où aucune contrainte n'est retirée, la recherche courante repart d'une affectation quelconque.

Le pseudo-code est décrit en figure 8. Les fonctions de bas niveau sont conformes aux spécifications données en 3.1, en 3.2 et en 4.1.

```
BT-DYN(p, C_a, C_r)
    let be V = VARIABLES(p)
    let be res = RESULT(p)
    ADD-CONSTRAINTS(p, C_a)
    REMOVE-CONSTRAINTS(p, C_r)
    if res = success
    then if ALL-SATISFIED-CONSTRAINTS(C_a, V)
           then return success
           else UNASSIGN(V)
                  return BT-VARIABLES(∅, V)
    else if C_r = ∅
           then return failure
           else return BT-VARIABLES(∅, V)
```

Figure 7. CSP dynamiques et backtrack

```
RP-DYN(p, C_a, C_r)
    let be V = VARIABLES(p)
    let be C = CONSTRAINTS(p)
    let be res = RESULT(p)
    ADD-CONSTRAINTS(p, C_a)
    REMOVE-CONSTRAINTS(p, C_r)
    if res = success
    then let be C_ua = UNSATISFIED-CONSTRAINTS(C_a, V)
           if C_ua = ∅
           then return success
           else return RP-VARIABLES(∅, V, C_ua)
    else if C_r = ∅
           then return failure
           else let be C_u = UNSATISFIED-CONSTRAINTS(C, V)
                  return RP-VARIABLES(∅, V, C_u)
```

Figure 8. CSP dynamiques et repair

4.3 Local Changes

Avec l'algorithme *local changes*, le résultat de la recherche précédente (une solution au cas où le problème précédent est cohérent, une affectation partielle cohérente dans le cas contraire) est systématiquement réutilisé. La démarche consiste simplement à désaffecter un sous-ensemble des variables affectées, tel que l'affectation courante redevienne cohérente, puis à utiliser l'affectation résultante comme point de départ de la recherche courante.

Le pseudo-code est décrit en figure 9. Les fonctions de bas niveau, qui n'ont pas déjà été spécifiées en 3.2, en 3.3 ou en 4.1 , répondent aux spécifications suivantes :

- ASSIGNED-VARIABLES(p) retourne l'ensemble des variables affectées suite à la résolution du problème p : l'ensemble des variables du problème si celui-ci est cohérent, un sous-ensemble dans le cas contraire ;
- UNASSIGNED-VARIABLES(p) retourne l'ensemble des variables non affectées suite à la résolution du problème p : l'ensemble vide si le problème est incohérent, un sous-ensemble des variables du problème dans le cas contraire.

```
LC-DYN(p, C_a, C_r)
    let be V_1 = ASSIGNED-VARIABLES(p)
    let be V_2 = UNASSIGNED-VARIABLES(p)
    let be res = RESULT(p)
    ADD-CONSTRAINTS(p, C_a)
    REMOVE-CONSTRAINTS(p, C_r)
    let be C_ua = UNSATISFIED-CONSTRAINTS(C_a, V_1)
    let be V_3 = CHOICE-VARIABLES-COVERING(V_1, C_ua)
    if res = success
    then if C_ua = ∅
            then return success
            else UNASSIGN(V_3)
                 return LC-VARIABLES(∅, V_1 − V_3, V_2 ∪ V_3)
    else if C_r = ∅
           then return failure
           else UNASSIGN(V_3)
                return LC-VARIABLES(∅, V_1 − V_3, V_2 ∪ V_3)
```

Figure 9. CSP dynamiques et local changes

4.4 Conflict Directed Backjumping

Nous avons vu en 3.5 que l'algorithme *conflict directed backjumping* est une amélioration de l'algorithme *backtrack*. D'où un comportement très voisin dans le cadre de problèmes dynamiques. La seule différence tient au fait que la version que nous avons présentée de cet algorithme retourne, au cas où le problème est incohérent, un sous-ensemble C' de l'ensemble des contraintes responsable de l'incohérence. Ceci permet d'affirmer, au cas où l'intersection de cet ensemble C' et de l'ensemble des contraintes retirées est vide, que le problème courant est aussi incohérent avec le même ensemble de contraintes responsable.

Le pseudo-code est décrit en figure 10. Les fonctions de bas niveau sont conformes aux spécifications données en 3.1, en 3.5 et en 4.1.

```
CBJ-DYN(p, C_a, C_r)
    let be V = VARIABLES(p)
    let be C = RESULT(p)
    ADD-CONSTRAINTS(p, C_a)
    REMOVE-CONSTRAINTS(p, C_r)
    if C = ∅
    then if ALL-SATISFIED-CONSTRAINTS(C_a, V)
           then return ∅
           else UNASSIGN(V)
                return CBJ-VARIABLES(∅, V)
    else if C ∩ C_r = ∅
           then return C
           else return CBJ-VARIABLES(∅, V)
```

Figure 10. CSP dynamiques et conflict directed backjumping

4.5 Nogood Recording

L'algorithme *nogood recording* est une amélioration de l'algorithme *conflict directed backjumping* (cf. 3.6). D'où un comportement voisin dans le cadre de problèmes dynamiques. La différence essentielle tient à la réutilisation des *nogoods* issus de la recherche précédente. Une telle réutilisation nécessite l'élimination préalable des *nogoods* qui sont devenus invalides du fait des contraintes retirées (ceux dont la justification contient une contrainte retirée).

Le pseudo-code est décrit en figure 11. La seule fonction de bas niveau, qui n'a pas déjà été spécifiée en 3.1, en 3.6 ou en 4.1, répond à la spécification suivante :

– REMOVE-NOGOODS(p, C) détruit tous les *nogoods* du problème p dont la justification contient une contrainte de l'ensemble de contraintes C.

```
NR-DYN(p, C_a, C_r)
    let be V = VARIABLES(p)
    let be C = RESULT(p)
    ADD-CONSTRAINTS(p, C_a)
    REMOVE-CONSTRAINTS(p, C_r)
    REMOVE-NOGOODS(p, C_r)
    if C = ∅
    then if ALL-SATISFIED-CONSTRAINTS(C_a, V)
          then return ∅
          else UNASSIGN(V)
                return NR-VARIABLES(∅, V)
    else if C ∩ C_r = ∅
          then return C
          else return NR-VARIABLES(∅, V)
```

Figure 11. CSP dynamiques et nogood recording

4.6 Dynamic Backtracking

C'est l'algorithme *dynamic backtracking* qui permet la réutilisation la plus fine de la recherche précédente :

– réutilisation de l'ensemble ordonné des variables affectées à l'issue de la recherche précédente : l'ensemble des variables du problème au cas où celui-ci est cohérent, un sous-ensemble dans le cas contraire ;

– réutilisation des *eliminating explanations* mémorisées à l'issue de la recherche précédente.

La démarche consiste à traiter successivement les contraintes ajoutées et retirées :

– pour toute contrainte c_a ajoutée et insatisfaite du fait de l'affectation courante, si $V(c_a)$ est l'ensemble des variables liées par la contrainte c_a et si v est la dernière variable affectée appartenant à $V(c_a)$, si val est sa valeur courante, alors :

  – la variable v est désaffectée ;

  – la valeur val est retirée du domaine de v ;

  – une *eliminating explanation*, contenant uniquement c_a, est créée pour cette valeur ;

  – toutes les *eliminating explanations* de contenu C' telles que v est liée par une contrainte de C' sont détruites (la désaffectation de v les rend invalides) ;

  – les valeurs correspondantes sont réintroduites dans les domaines ;

– pour toute contrainte c_r retirée, toutes les *eliminating explanations* contenant c_r sont détruites (le retrait de c_r les rend invalides).

Le pseudo-code est décrit en figure 12. Les fonctions de bas niveau, qui n'ont pas déjà été spécifiées en 3.2, en 3.7, en 4.1 ou en 4.3 répondent aux spécifications suivantes :

– SATISFIED-CONSTRAINT(c, V) retourne *true* si la contrainte c est satisfaite du fait de l'affectation de l'ensemble de variables V et retourne *false* sinon ;

– VARIABLES(c) retourne l'ensemble des variables liées par la contrainte c ;

– REMOVED-ELIMINATING-EXPLANATION-CONSTRAINT(V, c) détruit toutes les *eliminating explanations* des variables de l'ensemble de variables V, contenant la contrainte c et réintroduit les valeurs correspondantes dans leurs domaines respectifs.

4.7 Synthèse

Du point de vue de la réutilisation de la solution précédente, si elle existe, ou, plus généralement du point de vue de la réutilisation de l'ensemble des variables affectées à l'issue de la recherche précédente, on observe une progression de *backtrack* à *local changes*, en passant par *repair*.

Du point de vue de la réutilisation des contraintes produites et mémorisées au cours de la recherche précédente, on observe une progression identique de *backtrack* à *nogood recording*, en passant par *conflict directed backjumping*.

Notons enfin que *dynamic backtracking* a l'avantage de réutiliser la solution précédente au même niveau que *local changes* et les contraintes précédentes à un niveau proche de *nogood recording*.

5 Expérimentations

Pour évaluer et comparer ces différentes méthodes, des expérimentations systématiques ont été réalisées sur un même ensemble de problèmes dynamiques générés de façon aléatoire. Nous rappelons au lecteur les limitations inhérentes à l'utilisation de ce type de problèmes (possible manque de représentativité).

5.1 Génération

Le modèle de génération aléatoire utilisé est une extension au cas dynamique du modèle proposé dans [HUB 92]. Chaque problème dynamique est généré avec :

– un *nombre de variables* égal à n ;

– une *taille des domaines* égale à d, identique pour chacune des variables ;

– uniquement des *contraintes binaires* ;

```
DBT-DYN(p, C_a, C_r)
  let be V_1 = ASSIGNED-VARIABLES(p)
  let be V_2 = UNASSIGNED-VARIABLES(p)
  let be C = RESULT(p)
  ADD-CONSTRAINTS(p, C_a)
  REMOVE-CONSTRAINTS(p, C_r)
  let be V_3 = DBT-ADDED-CONSTRAINTS(V_1, V_2, C_a, ∅)
  DBT-REMOVED-CONSTRAINTS(V_1 ∪ V_2, C_r)
  if C = ∅
  then if V_3 = ∅
         then return ∅
         else return DBT-VARIABLES(V_1 − V_3, V_2 ∪ V_3)
  else if C ∩ C_r = ∅
        then return C
        else return DBT-VARIABLES(V_1 − V_3, V_2 ∪ V_3)

DBT-ADDED-CONSTRAINTS(V_1, V_2, C_a, V_3)
  if C_a = ∅
  then return V_3
  else let be c_a = CONSTRAINT-CHOICE(C_a)
        let be v = DBT-ADDED-CONSTRAINT(V_1, V_2, c_a)
        if v = ∅
        then return DBT-ADDED-CONSTRAINTS(V_1, V_2, C_a − {c_a}, V_3)
        else return
          DBT-ADDED-CONSTRAINTS(V_1 − {v}, V_2 ∪ {v}, C_a − {c_a}, V_3 ∪ {v})

DBT-ADDED-CONSTRAINT(V_1, V_2, c_a)
  if SATISFIED-CONSTRAINT(c_a, V_1)
  then return ∅
  else let be v = LAST-MEMBER(V_1, VARIABLES(c_a))
        DBT-BACKWARD(V_1, V_2, v, {c_a})
        return v

DBT-REMOVED-CONSTRAINTS(V, C_r)
   for each c_r in C_r
        REMOVED-ELIMINATING-EXPLANATION-CONSTRAINT(V, c_r)
```

Figure 12. CSP dynamiques et dynamic backtracking

– une *connectivité du graphe de contraintes* égale à c ; la connectivité du graphe de contraintes est le rapport entre le nombre nc de contraintes du problème et le nombre maximum $ncmax$ de contraintes binaires entre n variables ($ncmax = n(n-1)/2$; $nc = c * ncmax$) ; nc contraintes sont tirées aléatoirement et avec une probabilité uniforme parmi les $ncmax$ possibles ;

– une *dureté des contraintes* égale à t, identique pour chacune des contraintes ; la dureté d'une contrainte est le rapport entre le nombre nt de couples interdits par la contrainte et le nombre maximum $ntmax$ de couples interdits, correspondant à une contrainte interdisant tout couple de valeurs ($ntmax = d^2$; $nt = t * ntmax$) ; nt couples interdits sont tirés aléatoirement et avec une probabilité uniforme parmi les $ntmax$ possibles ;

– une *taille du changement* égale à tch ; pour chaque changement, le nombre nca de contraintes ajoutées est égal au nombre ncr de contraintes retirées et la taille du changement est égale au rapport entre ces deux nombres et le nombre nc de contraintes du problème ($nca = ncr = tch * nc$) ; en pratique, $ncmax$ contraintes de dureté t ont été générées ; parmi celles-ci, nc ont été tirées pour constituer le problème initial ; parmi celles qui ont été tirées, ncr sont maintenant tirées pour être retirées du problème et, parmi celles qui n'ont pas été initialement tirées, nca sont maintenant tirées pour y être ajoutées ; ces tirages sont aléatoires et avec une probabilité uniforme ; à noter que cette méthode garantit que le nouveau problème ait des caractéristiques globales (n, d, c et t) identiques à celles du problème initial.

Un problème dynamique est donc ici un couple constitué de deux problèmes statiques : un premier problème p_1 et un second problème p_2 résultant de l'ajout et du retrait de contraintes au/du problème p_1.

Les résultats que nous donnons dans ce papier sont obtenus sur un ensemble de problèmes dynamiques générés avec $n = 15$ et $d = 11$ (valeur choisie afin d'obtenir la fameuse transition de phase pour $t = p = 50\%$). Cet ensemble comprend lui-même trois sous-ensembles :

– le premier réunit des problèmes générés avec $c = 30\%$ et $t = 30\%$; ces problèmes dont l'espérance mathématique du nombre de solutions [3] est de l'ordre de 10^{10} sont largement *sous-contraints* et tous cohérents ;

– le second réunit des problèmes générés avec $c = 50\%$ et $t = 50\%$; ces problèmes dont l'espérance mathématique du nombre de solutions est de l'ordre de 1 sont des problèmes *intermédiaires* dont certains sont cohérents et d'autres incohérents ;

– le troisième réunit des problèmes générés avec $c = 70\%$ et $t = 70\%$; ces problèmes dont l'espérance mathématique du nombre de solutions est de l'ordre de 10^{-23} sont largement *sur-contraints* et tous incohérents ;

3. L'espérance mathématique du nombre de solutions d'un problème peut être approchée par la formule : $d^n * (1-t)^{c*n*(n-1)/2}$ [PRO 94], qui fait l'hypothèse grossière que les satisfactions des contraintes par une affectation des variables sont indépendantes l'une de l'autre.

Chacun de ces trois sous-ensembles comprend lui-même six sous-ensembles correspondant à des valeurs de tch égales à 1, 5, 10, 15, 20 et 25% et réunissant chacun 100 problèmes de caractéristiques identiques (n, d, c, t et tch).

5.2 Algorithmes

Les algorithmes expérimentés sont ceux qui ont été décrits en 3 et 4 : versions statiques et dynamiques des algorithmes *backtrack*, *repair*, *local changes*, *conflict directed backjumping*, *nogood recording* et *dynamic backtracking*.

Tous les algorithmes sont expérimentés avec *forward-checking*. Les *heuristiques* utilisées sont :

- pour le choix de la *variable* à affecter, à désaffecter ou à réaffecter : choix de la variable de plus petit domaine ;
- pour le choix de la *valeur* à affecter à une variable :
  - pour les algorithmes *backtrack*, *conflict directed backjumping*, *nogood recording* et *dynamic backtracking*: choix des valeurs dans l'ordre où elles apparaissent dans le domaine, sauf dans le cadre des versions dynamiques, où la valeur que la variable avait dans la solution précédente, si elle existe, est systématiquement choisie en premier ;
  - pour les algorithmes *repair* et *local changes* : choix de la valeur qui minimise le nombre de contraintes insatisfaites (heuristique de minimisation des conflits [MIN 92]).

5.3 Mesures

Pour chaque problème dynamique, c'est-à-dire pour chaque couple (p_1, p_2) de problèmes statiques, et pour chaque algorithme, le problème p_1 est résolu en utilisant la version statique de l'algorithme, puis le problème p_2 est résolu, une première fois en utilisant cette même version statique et une seconde fois en utilisant la version dynamique de l'algorithme et les résultats produits et mémorisés lors de la résolution de p_1.

En termes d'efficacité, les trois mesures classiques sont effectuées sur la résolution de p_2 : *nombre de nœuds*, *nombre de vérifications de contrainte* et *temps cpu*.

En termes de stabilité, la *distance entre solutions successives*, soit le nombre de variables dont l'affectation est différente dans les solutions de p_1 et p_2, est mesurée chaque fois que les deux problèmes sont cohérents.

6 Résultats

Les figures 13 et 14 montrent les résultats obtenus sur les problèmes sous-contraints ($c = t = 30\%$), en termes d'efficacité (figure 13) et de stabilité (figure 14). Les figures 15 et 16 montrent les résultats obtenus en termes d'efficacité sur les problèmes intermédiaires ($c = t = 50\%$, figure 15) et sur les problèmes sur-contraints ($c = t =$

70%, figure 16). La stabilité n'a en effet de sens que pour les problèmes sous-contraints (problèmes successifs cohérents).

Chaque figure comprend trois ensembles de courbes : le premier montre les résultats obtenus par les versions statiques des différents algorithmes, le second les résultats obtenus par les versions dynamiques des mêmes algorithmes et le troisième les rapports entre les seconds et les premiers.

En abscisse apparait la taille du changement (de 1 à 25%), en ordonnée la quantité mesurée (nombre de vérifications de contrainte quand il s'agit d'efficacité, distance entre solutions successives quand il s'agit de stabilité) ou le rapport (en %) entre les résultats obtenus par les versions dynamiques et statiques des différents algorithmes.

Chaque ensemble de courbes comprend six courbes dédiées à chacun des algorithmes : *bt* pour *backtrack*, *rp* pour *repair*, *lc* pour *local changes*, *cbj* pour *conflict directed backjumping*, *nr* pour *nogood recording* et *dbt* pour *dynamic backtracking*.

Chaque point représente la moyenne des résultats obtenus sur 100 problèmes dynamiques. À noter qu'aucun test statistique n'a été réalisé pour évaluer la confiance qu'il est possible d'y accorder.

7 Discussion

7.1 Problèmes sous-contraints

Efficacité De la figure 13, qui s'intéresse à l'efficacité sur les problèmes sous-contraints (nombre de vérifications de contrainte), on peut tirer les leçons suivantes :

– les résultats des versions statiques de *bt*, *cbj*, *nr* et *dbt* sont strictement identiques; ceux des versions statiques de *rp* et *lc* sont moins bons (légérement pour *rp*, nettement pour *lc*);

– les résultats des versions dynamiques de *bt*, *cbj* et *nr* sont strictement identiques; ceux des versions dynamiques de *rp* et *lc* sont meilleurs et proches les uns des autres; ceux de la version dynamique de *dbt* sont les meilleurs;

– les résultats des versions dynamiques des différents algorithmes sont systématiquement meilleurs que ceux des versions statiques; *bt*, *cbj* et *nr* profitent de façon identique de leur version dynamique; *rp* et *lc* en profitent nettement mieux et de façon presque identique, mais c'est *dbt* qui en profite le mieux; même si tous les algorithmes profitent d'autant plus de leur version dynamique que les changements sont de petite taille, *rp*, *lc* et surtout *dbt* continuent à en profiter même avec des changements de grande taille (jusqu'à 25% de contraintes retirées et 25% de contraintes ajoutées).

En termes d'efficacité, sur les problèmes sous-contraints, les résultats de la version dynamique de l'algorithme *dbt* sont donc incontestablement les meilleurs. À noter cependant que ces problèmes sont faciles à résoudre (plusieurs centaines de vérifications de contrainte).

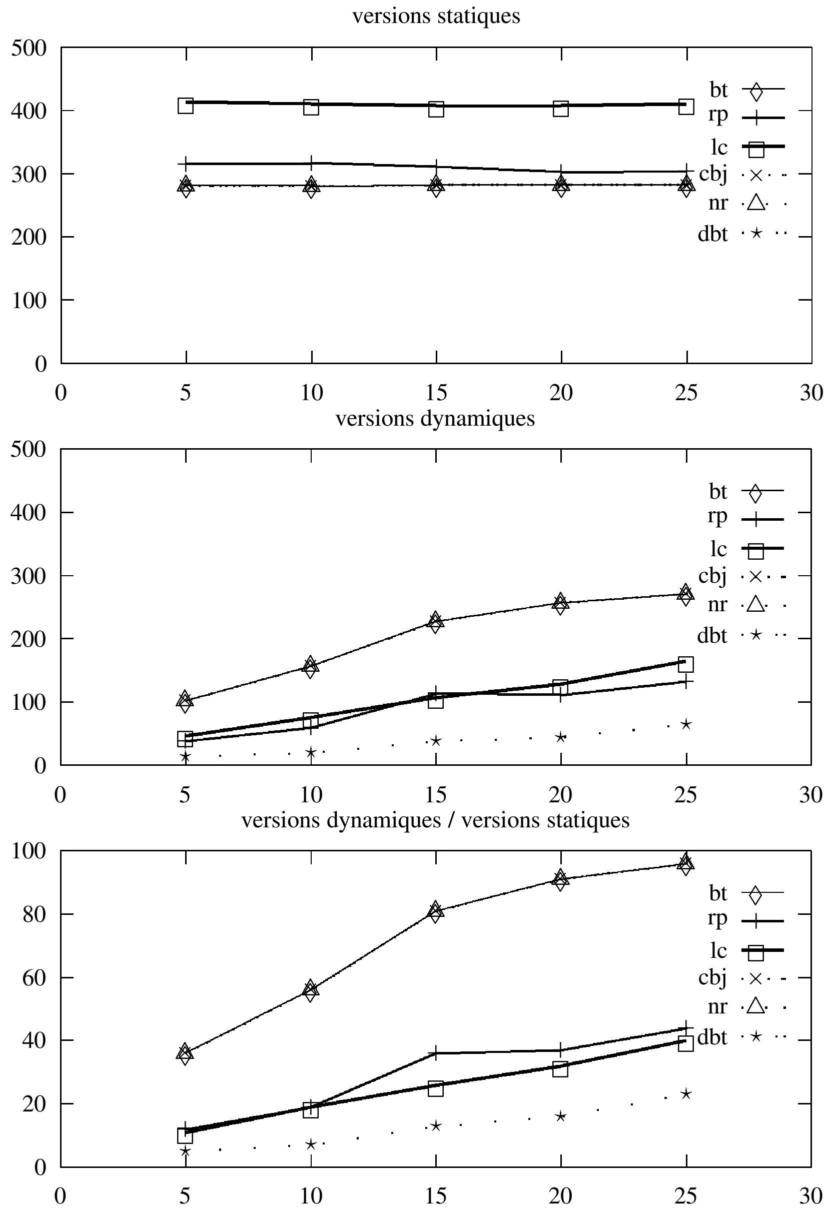

Figure 13. Efficacité sur des problèmes sous-contraints;
nombre de variables : 15, taille des domaines : 11,
connectivité du graphe : 30%, dureté des contraintes : 30%;
abscisse : taille des changements, ordonnée : nombre de vérifications de contrainte.

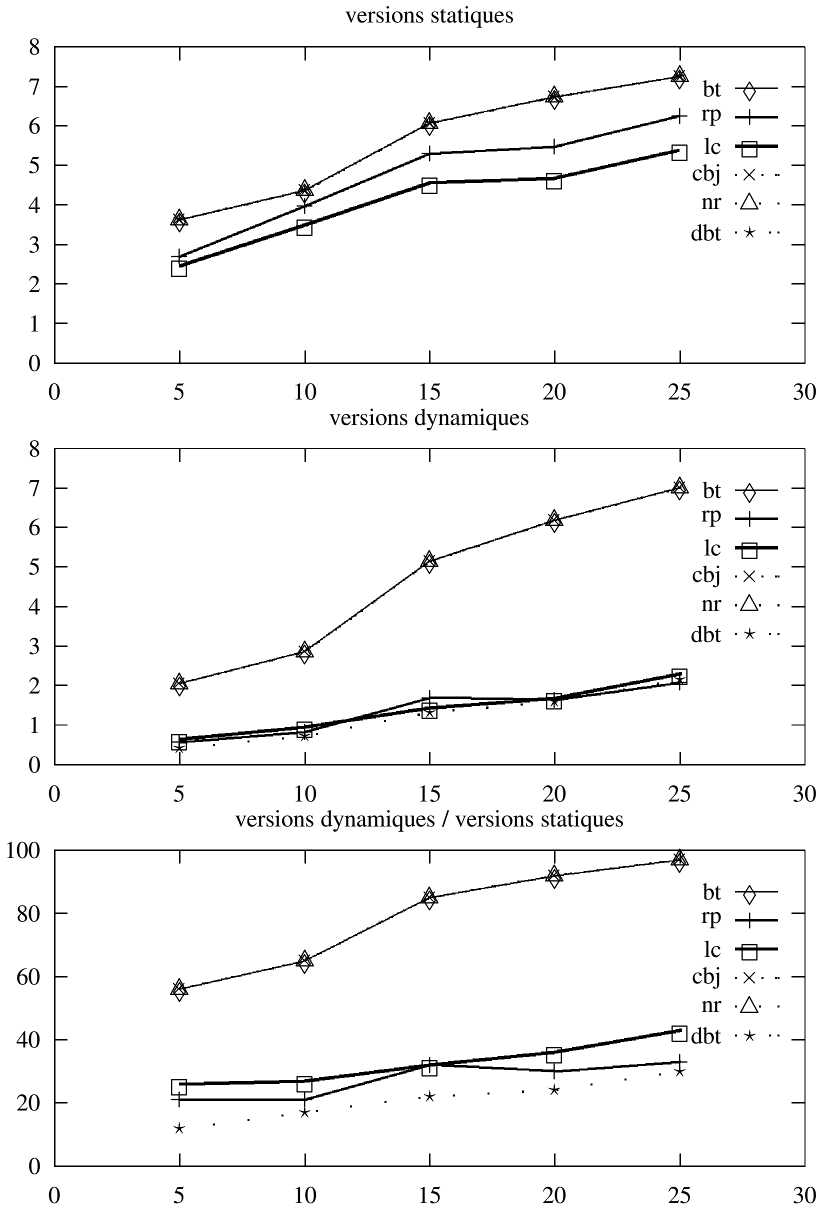

Figure 14. Stabilité sur des problèmes sous-contraints;
nombre de variables : 15, taille des domaines : 11,
connectivité du graphe : 30%, dureté des contraintes : 30%;
abscisse : taille des changements, ordonnée : distance entre solutions successives

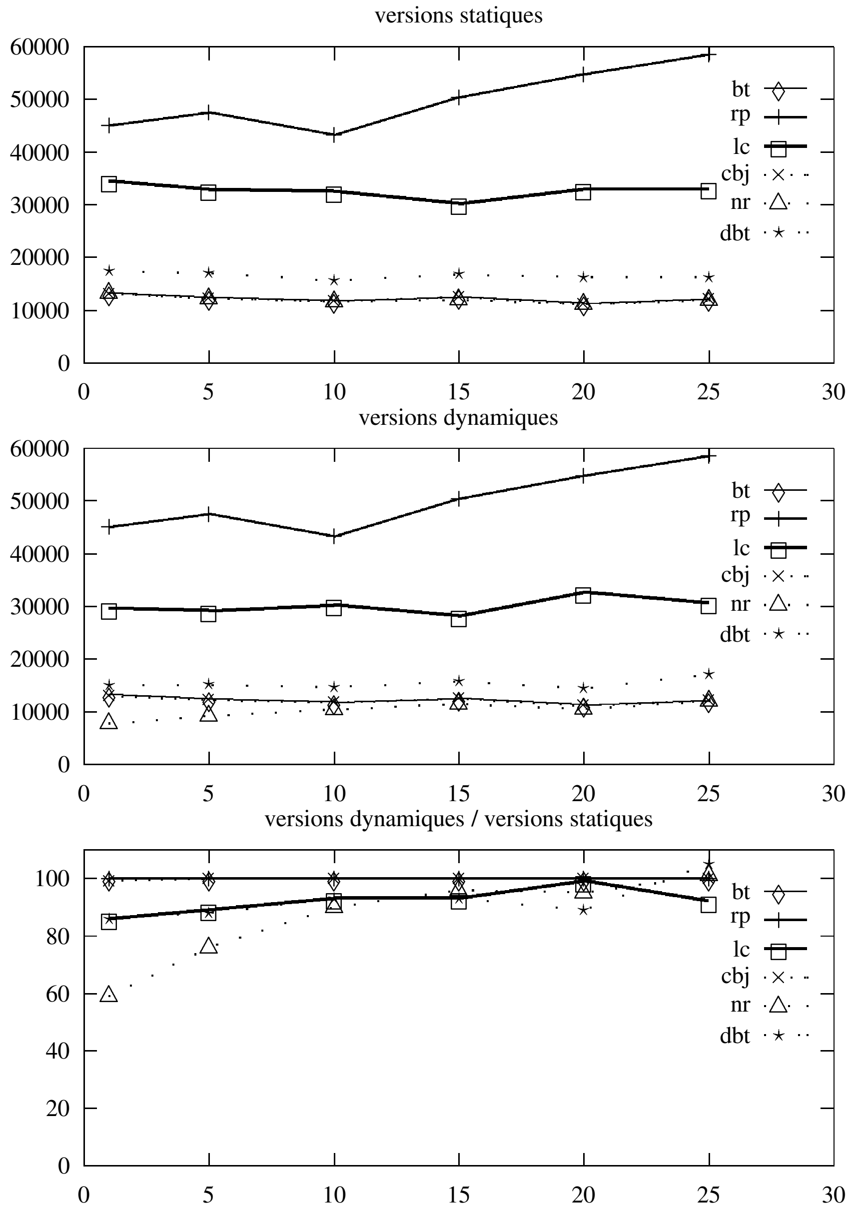

Figure 15. Efficacité sur des problèmes intermédiaires;
nombre de variables : 15, taille des domaines : 11,
connectivité du graphe : 50%, dureté des contraintes : 50%;
abscisse : taille des changements, ordonnée : nombre de vérifications de contrainte.

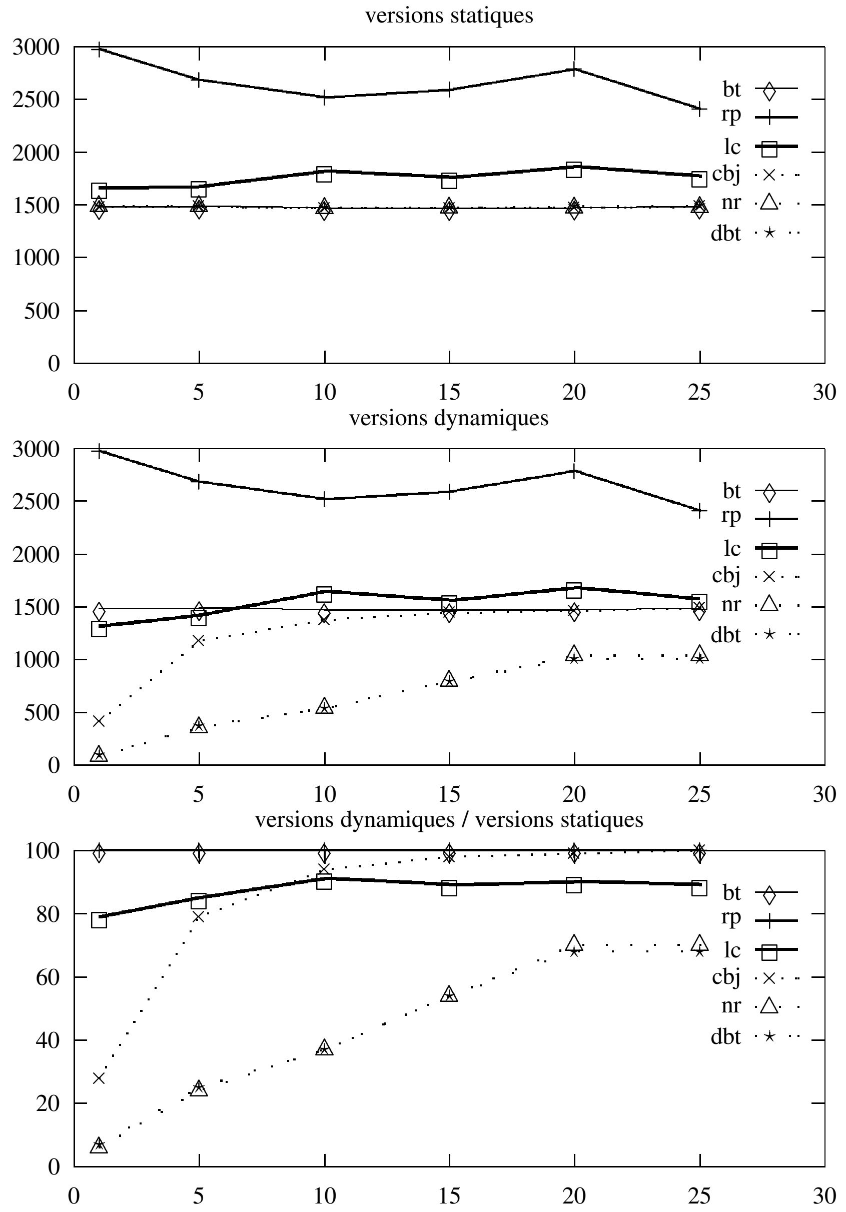

Figure 16. Efficacité sur des problèmes sur-contraints;
nombre de variables : 15, taille des domaines : 11,
connectivité du graphe : 70%, dureté des contraintes : 70%;
abscisse : taille des changements, ordonnée : nombre de vérifications de contrainte.

Stabilité De la figure 14, qui s'intéresse à la stabilité sur les problèmes sous-contraints (distance entre solutions successives), on peut tirer les leçons suivantes :

– les résultats des versions statiques de *bt*, *cbj*, *nr* et *dbt* sont strictement identiques; ceux des versions statiques de *rp* et *lc* sont meilleurs, ce qui peut étonner dans la mesure où les versions statiques n'utilisent, pour la résolution du problème p_2, aucune information provenant de la résolution du problème précédent p_1 ; il semble que le fonctionnement de ces deux algorithmes, allié à l'heuristique de minimisation des conflits, soit source de stabilité;

– les résultats des versions dynamiques de *bt*, *cbj* et *nr* sont strictement identiques; ceux des versions dynamiques de *rp*, *lc* et *dbt* sont nettement meilleurs et très proches les uns des autres;

– les résultats des versions dynamiques des différents algorithmes sont systématiquement meilleurs que ceux des versions statiques; *bt*, *cbj* et *nr* profitent de façon identique de leur version dynamique; *rp*, *lc* et *dbt* en profitent nettement mieux et de façon presque identique; même si tous les algorithmes profitent d'autant plus de leur version dynamique que les changements sont de petite taille, *rp*, *lc* et *dbt* continuent à en profiter même avec des changements de grande taille.

En termes de stabilité, sur les problèmes sous-contraints, les résultats des versions dynamiques des algorithmes *rp*, *lc*, *dbt* sont donc incontestablement les meilleurs.

7.2 Problèmes intermédiaires

De la figure 15, qui s'intéresse à l'efficacité sur les problèmes intermédiaires (nombre de vérifications de contrainte), on peut tirer les leçons suivantes :

– les résultats des versions statiques de *bt*, *cbj* et *nr* sont presque identiques; ceux de la version statique de *dbt* sont légèrement moins bons et ceux des versions statiques de *lc* et *rp* nettement moins bons; ceux de la version statique de *rp* sont les plus mauvais;

– sauf en ce qui concerne la version dynamique de *nr* qui devient la meilleure avec des changements de petite taille (jusqu'à 10%), le tableau précédent n'est pas modifié par les versions dynamiques;

– en fait, seuls *lc*, *nr* et *dbt* profitent de façon significative de leur version dynamique, avec des changements de petite taille; avec *nr* et *dbt* et des changements de grande taille, on peut par contre observer le phénomène inverse : une légère dégradation des résultats en passant de la version dynamique à la version statique.

En termes d'efficacité, sur les problèmes intermédiaires, les résultats de la version dynamique de l'algorithme *nr* sont les meilleurs et ce d'autant plus que les changements sont de petite taille. À noter que ces problèmes sont les plus difficiles à résoudre (plusieurs dizaines de milliers de vérifications de contrainte).

7.3 Problèmes sur-contraints

De la figure 16, qui s'intéresse à l'efficacité sur les problèmes sur-contraints (nombre de vérifications de contrainte), on peut tirer les leçons suivantes :

– les résultats des versions statiques de *bt*, *cbj*, *nr* et *dbt* sont presque identiques; ceux de la version statique de *lc* sont moins bons et ceux de la version statique de *rp* nettement moins bons;

– les résultats des versions dynamiques de *nr* et *dbt* sont presque identiques; ceux des versions dynamiques de *bt*, *cbj* et *lc* sont moins bons et proches les uns des autres, sauf avec des changements de petite taille pour lesquels les résultats de la version dynamique de *cbj* sont proches de ceux des versions dynamiques de *nr* et *dbt*; les résultats de la version dynamique de *rp* sont les plus mauvais;

– sauf en ce qui concerne *bt* et *rp*, dont les résultats des versions statiques et dynamiques sont identiques, les résultats des versions dynamiques sont meilleurs que ceux des versions statiques; mais *lc* profite peu de sa version dynamique; *cbj* en profite avec des changements de petite taille (jusqu'à 10%); ce sont *nr* et *dbt* qui en profitent le mieux et d'autant plus que les changements sont de petite taille;

En termes d'efficacité, sur les problèmes sous-contraints, les résultats des versions dynamiques des algorithmes *nr* et *dbt* sont donc les meilleurs et ce d'autant plus nettement que les changements sont de petite taille. À noter cependant que ces problèmes sont relativement faciles à résoudre (plusieurs milliers de vérifications de contrainte).

7.4 Temps CPU

Même si le *nombre de vérifications de contrainte* a la bonne propriété de mesurer l'efficacité relative des différents algorithmes indépendamment de leur implémentation et de la façon de représenter et de vérifier les contraintes, le *temps cpu* reste la mesure ultime en termes d'efficacité. C'est pourquoi nous indiquons dans le tableau de la figure 17, pour chaque algorithme, le nombre de vérification de contraintes par seconde obtenu avec la version statique, sur des problèmes intermédiaires ($c = t = 50\%$), ainsi que le rapport (en $\%$) entre ce résultat et le meilleur (obtenu avec *bt*). À noter que, pour chaque algorithme, la différence entre la version statique et la version dynamique n'est pas significative.

Ce tableau montre par contre des différences significatives entre algorithmes : les meilleurs résultats sont obtenus par *bt* et *cbj*, suivis de peu par *nr*; les résultats de *dbt* et *rp* sont moins bons et proches l'un de l'autre; les résultats de *lc* sont clairement les moins bons. Ceci modifie, mais ne bouleverse pas les divers classements établis sur la base du nombre de vérifications de contrainte.

8 Conclusion

La première leçon qui peut être tirée de ces expérimentations est qu'aucun algorithme ne se dégage comme étant systématiquement supérieur à tous les autres, ce qui

	nombre	pourcentage par rapport au meilleur
bt	3730	100
rp	2940	79
lc	2574	69
cbj	3723	100
nr	3576	96
dbt	3097	83

Figure 17. Nombre de vérifications de contrainte par seconde

n'est pas nouveau, mais vaut d'être rappelé, dans la mesure où trop de publications concluent hativement et contradictoirement à la supériorité de telle ou telle méthode sur la base d'expérimentations trop limitées.

En ce qui concerne nos expérimentations, dont il faut souligner qu'elles sont, elles aussi, limitées (problèmes de petite taille, problèmes construits avec un modèle de génération aléatoire rudimentaire et donc très homogènes), les résultats varient suivant le *degré de contrainte* (connectivité du graphe de contraintes, dureté des contraintes) des problèmes traités :

– sur les problèmes *sous-contraints*, possédant un grand nombre de solutions, des algorithmes capables de partir d'une solution précédente, tels que les versions dynamiques de *rp*, *lc* et *dbt*, semblent bien adaptés; en termes de stabilité, leurs résultats sont très proches les uns des autres, mais, en termes d'efficacité, la version dynamique de *dbt* est la meilleure;

– sur les problèmes *intermédiaires*, les plus difficiles à résoudre, l'apport des versions dynamiques des différents algorithmes est manifestement plus faible; seule la version dynamique de *nr*, avec des changements de petite taille, semble tirer profit de son mécanisme de production et de réutilisation de *nogoods*;

– sur les problèmes *sur-contraints*, ne possédant aucune solution, le mécanisme de production et de réutilisation de *nogoods* (ou *eliminating explanations*) semble par contre jouer à fond au profit des versions dynamiques de *nr* et *dbt*.

Réutilisation de *solution* sur les problèmes sous-contraints, réutilisation de *contraintes* sur les problèmes intermédiaires et sur-contraints : telle semble être la clé pour obtenir efficacité et stabilité dans le traitement des problèmes dynamiques de satisfaction de contraintes.

Remerciements Ces études ont été réalisés au sein du CERT-ONERA et financées par des contrats CNES et DGA-DRET, que nous tenons à remercier pour la confiance qu'ils nous ont accordée.

Références

[BER 94] BERLANDIER P. et NEVEU B., Maintaining Arc Consistency through Constraint Retraction. *In : Proc. of the 6th IEEE International Conference on Tools with Artificial Intelligence (TAI94)*, New Orleans, LA, 1994.

[BES 91] BESSIÈRE C., Arc-Consistency in Dynamic Constraint Satisfaction Problems. *In : Proc. of AAAI-91*, p. 221–226, Anaheim, CA, 1991.

[CHE 91] CHEESEMAN P., KANEFSKY B. et TAYLOR W., Where the *really* Hard Problems Are. *In : Proc. of IJCAI-91*, p. 294–299, Sidney, Australia, 1991.

[DEC 88] DECHTER R. et DECHTER A., Belief Maintenance in Dynamic Constraint Networks. *In : Proc. of AAAI-88*, p. 37–42, St. Paul, MN, 1988.

[DEC 90] DECHTER R., Enhancement Schemes for Constraint Processing : Backjumping, Learning and Cutset Decomposition. *Artificial Intelligence*, vol. 41, n° 3, 1990, p. 273–312.

[DOY 79] DOYLE J., A Truth Maintenance System. *Artificial Intelligence*, vol. 12, 1979, p. 231–272.

[FB 90] FREEMAN-BENSON B., MALONEY J. et BORNING A., An Incremental Constraint Solver. *Communications of the ACM*, vol. 33, n° 1, 1990, p. 54–63.

[FRE 78] FREUDER E., Synthesizing Constraint Expressions. *Communications of the ACM*, vol. 21, 1978, p. 958–966.

[FRO 94] FROST D. et DECHTER R., Dead-end driven learning. *In : Proc. of AAAI-94*, p. 294–300, Seattle, WA, 1994.

[GEN 93] GENT I. et WALSH T., Towards an Understanding of Hill-Climbing Procedures for SAT. *In : Proc. of AAAI-93*, p. 28–33, Washington, DC, 1993.

[GHE 93] GHEDIRA K., *MASC : une Approche Multi-Agent des Problèmes de Satisfaction de Contraintes*, Toulouse, France, Thèse de doctorat, ENSAE, 1993.

[GIN 93] GINSBERG M., Dynamic Backtracking. *Journal of Artificial Intelligence Research*, vol. 1, 1993, p. 25–46.

[GIN 94] GINSBERG M. et MCALLESTER D., GSAT and Dynamic Backtracking. *In : Proc. of the 4th International Conference on the Principles of Knowledge Representation and Reasoning (KR94)*, p. 226–237, Bonn, Germany, 1994.

[GOL 65] GOLUMB S. et BAUMERT L., Backtrack programming. *Journal of the ACM*, vol. 12, 1965, p. 516–524.

[HEN 91] HENTENRYCK P. V. et PROVOST T. L., Incremental Search in Constraint Logic Programming. *New Generation Computing*, vol. 9, 1991, p. 257–275.

[HUB 92] HUBBE P. et FREUDER E., An Efficient Cross-Product Representation of the Constraint Satisfaction Problem Search Space. *In : Proc. of AAAI-92*, p. 421–427, San Jose, CA, 1992.

[KLE 89] DE KLEER J., A Comparison of ATMS and CSP Techniques. *In : Proc. of IJCAI-89*, p. 290–296, Detroit, MI, 1989.

[MIN 92] MINTON S., JOHNSTON M., PHILIPS A. et LAIRD P., Minimizing Conflicts: a Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence*, vol. 58, 1992, p. 160–205.

[MIT 90] MITTAL S. et FALKENHAINER B., Dynamic Constraint Satisfaction Problems. *In : Proc. of AAAI-90*, p. 25–32, Boston, MA, 1990.

[PRO 92] PROSSER P., CONWAY C. et MULLER C., A Constraint Maintenance System for the Distributed Allocation Problem. *Intelligent Systems Engineering*, vol. 1, n° 1, 1992, p. 76–83.

[PRO 93] PROSSER P., Hybrid Algorithms for the Constraint Satisfaction Problems. *Computational Intelligence*, vol. 9, n° 3, 1993, p. 268–299.

[PRO 94] PROSSER P., Binary Constraint Satisfaction Problems: Some are Harder than Others. *In : Proc. of ECAI-94*, Amsterdam, The Netherlands, 1994.

[SAN 93] SANNELLA M., The SkyBlue Constraint Solver and Its Applications. *In : Proc. of the 1993 Workshop on Principles and Practice of Constraint Programming (PPCP93)*, Newport, RI, 1993.

[SCH 93] SCHIEX T. et VERFAILLIE G., Nogood Recording for Static and Dynamic CSP. *In : Proc. of the* 5^{th} *IEEE International Conference on Tools with Artificial Intelligence (TAI93)*, p. 48–55, Boston, MA, 1993. Une version étendue a été publiée dans [SCH 94a].

[SCH 94a] SCHIEX T. et VERFAILLIE G., Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, vol. 3, n° 2, 1994, p. 187–207.

[SCH 94b] SCHIEX T. et VERFAILLIE G., Stubbornness: a Possible Enhancement for Backjumping and Nogood Recording. *In : Proc. of ECAI-94*, p. 165–169, Amsterdam, The Netherlands, 1994.

[SEL 92] SELMAN B., LEVESQUE H. et MITCHELL D., A New Method for Solving Hard Satisfiability Problems. *In : Proc. of AAAI-92*, p. 440–446, San Jose, CA, 1992.

[SMI 88] SMITH B., Forward-Checking, the ATMS and Search Reduction. *In : Reason Maintenance Systems and their Applications*, éd. par SMITH B. et KELLEHER G., chap. 9, p. 155–168, Ellis Horwood, 1988.

[TSA 93] TSANG E., *Foundations of Constraint Satisfaction*, London, Academic Press Ltd., 1993.

[VER 94a] VERFAILLIE G. et SCHIEX T., Dynamic Backtracking for Dynamic Constraint Satisfaction Problems. *In : Proc. of the ECAI94 Workshop on "Constraint Satisfaction Issues Raised by Practical Applications"*, Amsterdam, The Netherlands, 1994.

[VER 94b] VERFAILLIE G. et SCHIEX T., Solution Reuse in Dynamic Constraint Satisfaction Problems. *In : Proc. of AAAI-94*, p. 307–312, Seattle, WA, 1994.

Gérard Verfaillie *est depuis 1986 ingénieur de recherches au sein du Département d'Informatique du CERT-ONERA. Il s'y intéresse au raisonnement à base de contraintes pour résoudre ou aider à résoudre des problèmes de planification, de conception ou de diagnostic, avec application au domaine spatial (conception et gestion de satellites d'observation…).*

Thomas Schiex *a été ingénieur de recherches au sein du Département d'Automatique du CERT-ONERA de 1991 à fin 1994. Depuis 1994, il est chargé de recherches à l'INRA. Il s'intéresse particulièrement à la résolution de problèmes de satisfaction et d'optimisation sous contraintes dans des domaines discrets, issus plus particulièrement de la biologie.*