

THESE

En vue de l'obtention du : **DOCTORAT**

Structure de Recherche : LIMIARF

Discipline : Sciences de l'Ingénieur

Spécialité : Intelligence Artificielle

Présentée et soutenue le : 21/04/2018 par :

Yosra ACODAD

Algorithmes de réparation et heuristiques d'ordonnancement pour les problèmes de satisfaction de contraintes dynamiques

JURY

Fouzia OMARY	PES, Université Mohammed V, Faculté des Sciences de Rabat	Président
El Houssine BOUYAKHF	PES, Université Mohammed V, Faculté des Sciences de Rabat	Directeur de thèse
Imade BENELALLAM	PH, Institut National de Statistique et d'Economie Appliquée (INSEA)	Examineur
Ali GUEDIRA	Expert en Technologies de l'Information, Energie et Politiques Publiques	Examineur
Redouane EZZAHIR	PH, Université Ibn Zohr, Ecole Nationale des Sciences Appliquées (ENSA) d'Agadir	Rapporteur
Abdeslam KADRANI	PH, Institut National de Statistique et d'Economie Appliquée (INSEA)	Rapporteur

Année Universitaire :2017-2018

Algorithmes de réparation et heuristiques
d'ordonnancement pour les problèmes de
satisfaction de contraintes dynamiques

Yosra Acodad

19 juillet 2018

Remerciement

Les travaux présentés dans ce mémoire de thèse ont été réalisés au sein du laboratoire *LIMIARF* (Laboratoire d'Informatique, Mathématique appliquées, Intelligence Artificielle et Reconnaissance de Formes) de la Faculté des Sciences de Rabat (*FSR*).

Ce travail a été effectué sous la direction du Professeur *El Houssine Bouyakhf* de la *FSR* en Codirection avec le Professeur *Imade Benelallam* de l'Institut National de Statistiques et Économie Appliquée (*INSEA*) de Rabat.

Je remercie très sincèrement M. *El Houssine Bouyakhf* pour son encadrement et ses conseils et tiens à lui exprimer ma reconnaissance et ma profonde gratitude pour m'avoir accueillie au sein du laboratoire *LIMIARF*. Merci pour votre confiance et pour les multiples discussions qui ont permis l'aboutissement de cette thèse.

Je tiens à remercier M. *Imade Benelallam* pour son encadrement, ses précieux conseils et ses pertinentes remarques.

J'exprime toute ma gratitude à Mme *Fouzia Omary*, Professeur à la *FSR*, qui m'a fait l'honneur de présider le jury de ma soutenance.

Je tiens à remercier mon rapporteur de thèse, M. *Redouane Ezzahir*, Professeur Habilité à l'École Nationale des Sciences Appliquées (*ENSA*) d'Agadir, pour avoir accepté d'évaluer mon travail et de lire mon manuscrit pour apporter ses précieux conseils.

De même, je remercie mon second rapporteur, M. *Abdeslam Kadrani*, Professeur Habilité à l'*INSEA* de Rabat, pour son évaluation de mon travail et ses pertinentes remarques.

Je remercie également mon examinateur M. *Ali Guedira*, Expert en Technologies de l'Information, Énergie et Politiques, pour l'intérêt qu'il a porté à mon travail.

Je tiens à remercier mes très chers parents pour leur confiance et encouragement durant toutes ces années d'études, ce qui m'a permise de développer ce goût pour la science. Merci également au reste de ma famille et surtout à mon mari et collègue pour son soutien.

Je remercie tous les membres du laboratoire, et surtout mes collègues de l'équipe *AI*. Ce fut une expérience très intéressante de travailler avec vous durant ces années.

Que chacun trouve dans mon travail l'expression de ma haute considération.

Résumé

Dans le monde réel, une large gamme de problèmes pratiques sont des problèmes dynamiques, suite aux changements imprévisibles dans l'environnement ou dans la structure du problème lui-même. Ces changements sont causés par l'aspect dynamique du problème, le manque de connaissance complète ou les fausses actions.

Dans cette thèse, nous nous intéressons au traitement des problèmes dynamiques par le paradigme de programmation par contraintes *CP* (*Constraint Programming*), notamment les problèmes de satisfaction de contraintes dynamiques *DCSPs* (*Dynamic Constraint Satisfaction Problems*) qui désignent l'ensemble de ces problèmes qui changent au fil du temps. L'objectif de notre travail est de proposer essentiellement de nouvelles méthodes pour réparer les *DCSPs*, ainsi que des heuristiques qui orientent la recherche dans le but de réduire l'effort de la réparation.

Notre thèse est constituée de trois axes principaux :

Le premier axe consiste à concrétiser notre travail, en proposant une modélisation d'une des applications d'allocation de ressources suivant le paradigme *CP*, à savoir le problème d'affectation des stages hospitaliers, sous forme de problème de satisfaction de contraintes *CSP* (*Constraint Satisfaction Problem*), qui peut être étendue sous forme de *DCSP*.

Le deuxième axe est consacré à proposer, en première partie, de nouvelles approches de réparation de *DCSPs*, ainsi que des heuristiques d'ordonnement en deuxième partie.

Les approches proposées, *Extended PDB* (*EPDB*), *Lazy Repairing Backtracking* (*LRB*) et *Improved PDB* (*IPDB*), sont des améliorations de l'une des méthodes les plus connues de réparation de *DCSPs*, à savoir *PDB* (*Partial-order Dynamic Backtracking*).

Ensuite, nous proposons des heuristiques dynamiques de choix de contraintes *DCO* (*Dynamic Constraint Ordering*), dont le but est d'adopter une stratégie d'ordre dont les contraintes sont testées. Aussi, nous proposons une heuristique de choix de variables *PDeg* (*Profound Degree*), qui vise à garder aussi stables que possible les parties les plus critiques du réseau lors de la réparation d'un *DCSP*.

Enfin, le troisième axe concerne les problèmes *DCSPs* Distribués (*Dynamic Distributed Constraint Satisfaction Problems*), pour lesquels nous avons proposé une approche de réparation en temps réel nommée *LiveABT* (*Live Asynchronous BackTracking*).

Abstract

In the real world, a wide range of practical problems are dynamic, due to unpredictable changes in the environment or in the structure of the problem. These changes are caused by the dynamic aspect of the problem, the lack of complete knowledge or the wrong actions.

In this thesis, we are interested in the treatment of dynamic problems by the constraint programming paradigm (*CP*), more precisely by dynamic constraint satisfaction problems *DCSPs* which refers to all of these problems that change over time.

The goal of our work is to propose essentially new methods to repair (*DCSPs*), as well as heuristics that direct the search in order to reduce the effort of repairing.

Our thesis consists of three main axes :

The first axis is to concretize our work by proposing a modeling of one of the resource allocation applications following the *CP* paradigm, namely the problem of assigning hospital internships, in the form of a constraint satisfaction problem (*CSP*), which can be expanded as *DCSP*.

The second axis is dedicated to propose, in the first part, new repair approaches of *DCSPs*, as well as scheduling heuristics in the second part. The proposed approaches, *Extended PDB (EPDB)*, *Lazy Repairing Backtracking (LRB)* and *Improved PDB (IPDB)*, are improvements of one of the most known *DCSPs* repair methods, namely *PDB (Partial-order Dynamic Backtracking)*.

Next, we propose dynamic constraint heuristics *DCO (Dynamic Constraint Ordering)*, whose goal is to adopt an order strategy whose constraints are tested. Also, we propose a variable choice heuristic *PDeg (Profound Degree)*, which aims to keep as stable as possible the most critical parts of the network when repairing a *DCSP*.

Finally, the third axis concerns Distributed *DCSPs* issues, for which we have proposed a real-time repair approach named *LiveABT (Live Asynchronous BackTracking)*.

Table des matières

1	Problèmes de Satisfaction de Contraintes	12
1.1	formalisme des CSPs	13
1.2	Exemples de CSPs	14
1.2.1	Le coloriage de graphe	14
1.2.2	Les n-reines	14
1.2.3	La planification de réunion	15
1.3	Stratégies de résolution de CSPs	16
1.3.1	Algorithmes à exploration systématique	17
1.3.2	Algorithmes rétrospectifs (lookback)	18
1.3.3	Algorithmes prospectifs (lookahead)	21
1.3.4	Heuristiques d'ordonnancement	24
2	Contribution aux CSPs	28
2.1	Description de la problématique	29
2.2	Modélisation du problème en tant que CSP	30
2.3	implémentation du problème	32
3	Les CSPs Dynamiques	33
3.1	formalisme des DCSPs	34
3.2	Méthodes de réparation de DCSPs	34
3.2.1	Travaux connexes	34
3.2.2	Partial-order Dynamic Backtracking	36
4	Contribution aux DCSPs	40
4.1	Extended Partial-order Dynamic Backtracking	41
4.1.1	Motivation	41
4.1.2	EPDB comparé à PDB	42
4.1.3	Exemple d'exécution	43
4.1.4	Description de l'algorithme	45
4.1.5	Preuve de terminaison	45
4.1.6	Étude expérimentale	47
4.1.7	Discussion	52
4.2	Dynamic Constraint Ordering	54

4.2.1	Introduction	54
4.2.2	Travaux effectués	54
4.2.3	Dynamic Constraint Ordering	56
4.2.4	Résultats expérimentaux	61
4.2.5	Discussion	64
4.3	Profound degree	65
4.3.1	Motivation	65
4.3.2	Exemple d'exécution	65
4.3.3	Description de l'heuristique	67
4.3.4	Étude expérimentale	68
4.3.5	Discussion	72
4.4	Improved Partial-order Dynamic Backtracking	75
4.4.1	Motivation	75
4.4.2	Description de la structure de données proposée	78
4.4.3	Preuve de terminaison	79
4.4.4	Étude expérimentale	80
4.4.5	Expérimentations sur le problème de planification de réunions dynamiques <i>DMSP</i>	84
4.4.6	Discussion	87
4.5	Lazy Repairing Backtracking	89
4.5.1	Motivation	89
4.5.2	LRB comparé à EPDB	90
4.5.3	Exemple d'exécution	91
4.5.4	Description de l'algorithme	93
4.5.5	preuve de terminaison	95
4.5.6	Étude expérimentale	95
4.5.7	Discussion	100
5	Les CSPs Dynamiques et Distribués	104
5.1	Introduction	105
5.2	formalisme des CSPs Dynamiques et distribués	106
5.3	Méthodes de réparation de DisCSPs dynamiques	107
5.3.1	Asynchronous Backtracking	107
5.3.2	Dynamic Asynchronous Backtracking	107
6	Contribution aux DDisCSPs	110
6.1	description du LiveABT	111
6.2	LiveABT comparé à DynABT	111
6.3	exemple de simulation	112
6.3.1	Perturbation 1 : L'ajout de contrainte	112
6.3.2	Perturbation 2 : La suppression de contrainte	113
6.4	Exécution	115
6.5	Terminaison et complétude	118
6.6	Expérimentations	118

6.6.1	Expérimentations sur les problèmes aléatoires	119
6.6.2	Expérimentations sur les problèmes de coloriage de graphe	120
6.6.3	Expérimentations sur les problèmes de planification de réunions	122
6.6.4	Discussion	125
7	Conclusion et perspectives	126

Introduction générale

Les contraintes apparaissent spontanément dans la vie réelle. Nombreuses sont les situations, confrontées par l'être humain, qui peuvent être traduites par des contraintes. En fait, les contraintes formalisent d'une manière claire et naturelle les dépendances qui existent dans le monde physique.

Les contraintes sont un moyen très générique pour représenter les régularités qui gouvernent notre monde, que ce soit au niveau informationnel, physique, biologique ou social. En général, elles fixent les règles de ce qui est possible ou non.

Les contraintes ont quelques propriétés intéressantes [11] qui leur sont caractéristiques. Elles sont déclaratives : elles spécifient une relation qui doit être vérifiée, sans pour autant définir la manière d'atteindre ce but. Elles sont également additives, puisque l'ordre de leur définition ne compte pas, et seule la conjonction finale des contraintes posées est importante. De plus, elles sont non-directionnelles.

La Programmation Par Contraintes (*CP* pour *Constraint Programming*) [42], est une branche interdisciplinaire de l'informatique qui fait appel à des techniques issues de différents champs de connaissance, notamment la logique, les langages de programmation, la recherche opérationnelle et surtout l'intelligence artificielle (*AI* pour *Artificial Intelligence*). Plus précisément, la *CP* c'est un paradigme de programmation, qui se base sur les contraintes, visant à offrir des langages de modélisation déclaratifs de haut niveau et des algorithmes de résolution efficaces et génériques et permettant ainsi de résoudre des problèmes *NP complets* tels que les problèmes de planification et d'ordonnancement.

Résoudre un problème dans un environnement de *CP* revient à passer par deux étapes distinctes. La première étape consiste à déclarer et définir toutes les variables (les inconnues) du problème à résoudre ainsi que les contraintes qui gouvernent la manière dont ces variables peuvent être affectées les unes vis-à-vis des autres. Cette première étape est appelée l'étape de modélisation. La deuxième étape consiste à chercher la solution du problème modélisé lors de la première étape. Cette recherche se fait soit en lançant une procédure de recherche prédéfinie dans l'environnement ou en définissant une nouvelle procédure de recherche en s'appuyant sur les différents algorithmes proposés par l'environnement lui même.

Du fait de sa simplicité et son efficacité, la *CP* a connu lors de ces vingt dernières années un grand succès, que ce soit dans le monde académique ou industriel. D'une part, depuis son émergence, les chercheurs n'ont pas cessé d'améliorer le niveau d'expressivité des langages de modélisation et l'efficacité des algorithmes de résolution. D'autre part, les industriels ont trouvé dans la *CP* un moyen simple pour exprimer et résoudre des problèmes réels et pratiques qui leur permet d'améliorer leurs services et d'augmenter leur productivité. Comme exemples de ces applications pratiques nous pouvons citer : la confection des emplois du temps dans les établissements ; la planification des trains ; l'ordonnancement des tâches de fabrication dans les usines ; la planification du trafic aérien ; l'ordonnancement d'actions ; la gestion de ressources dans les administrations ; la configuration et le dimensionnement des systèmes ; la conception des circuits intégrés ; etc.

La notion de Problème de Satisfaction de Contraintes (*CSP* pour *Constraint Satisfaction Problem*) désigne l'ensemble de ces problèmes, définis par des contraintes, et consistant à chercher une solution qui les respectent.

Le formalisme et les méthodes *CSP* sont de plus en plus utilisés pour représenter et traiter des problèmes d'intelligence artificielle et de recherche opérationnelle, mais en réalité, la plupart de ces problèmes sont des problèmes dynamiques (du fait de l'environnement, de l'utilisateur ou de tout autre agent). Ainsi, les *CSPs* dynamiques (*DCSPs* pour *Dynamic CSPs*) furent introduits pour répondre à un réel besoin dans ce carrefour.

En fait, la plupart des problèmes de planification, de conception et de reconnaissance de situations sont des problèmes de nature dynamique.

Dans des problèmes de planification, et surtout concernant l'ordonnancement d'actions ou de gestion de ressources 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 conception, et plus précisément dans la configuration ou de dimensionnement de systèmes, les spécifications du système à concevoir peuvent varier du fait de l'utilisateur qui désire connaître les implications de spécifications différentes ou du fait d'entités responsables de la conception d'autres parties ou aspects du système (conception distribuée). Un concepteur doit faire des restrictions pour compléter la définition de son problème, mais doit aussi pouvoir retirer des contraintes quand il n'y a plus de solution.

Dans des problèmes de reconnaissance de situations, surtout en diagnostic de pannes, 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.

Le traitement d'un *DCSP* doit prendre en compte les besoins de l'utilisateur, concernant l'efficacité de la recherche et la stabilité des solutions successives, d'une part, et d'autre part, le problème technique.

Concernant les besoins de l'utilisateur, le besoin de mécanismes de recherche efficaces n'est pas nouveau dans le cadre des *CSP*, mais il peut être plus fort dans le cadre de problèmes dynamiques où le temps disponible pour produire une nouvelle solution peut être limité. Aussi, la stabilité des solutions successives peut être importante, au cas où un certain travail (commandes, préparation, organisation...) a pu être engagé sur la base de la solution précédente.

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 les solutions et/ou contraintes ainsi que l'intelligence acquise durant les recherches précédentes, quelque soit la nature du changement (ajouts et/ou retraits de contraintes).

Ainsi, nos travaux de thèse portent essentiellement sur le développement de nouvelles approches de réparation et heuristiques d'ordonnancement, dans le but de réduire l'effort déployé pour réparer les *CSPs* dynamiques.

Chapitre 1

Problèmes de Satisfaction de Contraintes

1.1 formalisme des CSPs

Définition 1 (CSP). *Un Problème de Satisfaction de Contraintes (CSP) est un tuple (X, D, C) , tels que :*

- X est l'ensemble des n variables $\{x_1, x_2, \dots, x_n\}$.
- D est l'ensemble des domaines $\{D_1, D_2, \dots, D_n\}$ de ces variables, tel que D_i contient les valeurs possibles que x_i peut avoir.
- C est l'ensemble des m contraintes $\{c_1, c_2, \dots, c_m\}$ entre des variables appartenant à des sous-ensembles de X . chaque $c_i \in C$ exprime une relation qui définit les combinaisons des affectations possibles (ou interdites) pour les variables $\text{vars}(c_i)$ dans la portée de la contrainte.

Rappelons que les contraintes peuvent être définies aussi bien en extension qu'en intention.

Une contrainte en extension est définie par une liste de tuples : Ce sont les paires qui violent la relation qui définit la nature de cette contrainte.

Une contrainte en intention, par contre, est décrite par un prédicat qui est évalué pour déterminer si le tuple en question satisfait la contrainte ou pas.

Définition 2 (Arité d'une contrainte). *On désigne par $\text{var}(c_i)$ l'ensemble des variables mises en jeu par la contrainte c_i . Si $\text{var}(c_i)$ est un singleton, on dit que c_i est une contrainte unaire. Si $\text{var}(c_i)$ est une paire alors c_i est une contrainte binaire. Si $\text{var}(c_i)$ est un ensemble à n éléments alors c_i est une contrainte n – aire. On dit aussi que n est l'arité de c_i .*

On désignera par c_{ij} la contrainte binaire portant sur les variables x_i et x_j . Un CSP est binaire si toutes ses contraintes sont d'arité 2. Les CSP binaires jouent un rôle particulier car n'importe quel CSP peut être transformé en un CSP binaire équivalent [6]. C'est pour cela que la plupart des recherches sur les méthodes pour la résolution de CSPs discrets (CSPs où les domaines sont finis) traitent les CSP binaires.

Un CSP binaire peut être représenté par un graphe de contraintes dans lequel chaque noeud représente une variable, et chaque arc correspond à une contrainte entre 2 variables.

Dans cette thèse, nous nous focalisons sur les CSPs binaires.

Définition 3 (affectation). *Une affectation élémentaire est un couple variable-valeur (x_i, v) tel que v appartient à D_i . Une affectation est un ensemble d'affectations élémentaires. Si une affectation porte sur toutes les variables du CSP alors on parle d'affectation complète. Dans le cas contraire, on parle d'affectation partielle.*

Définition 4 (Solution). *Une solution d'un CSP P est une affectation complète S telles que toutes les contraintes de P sont satisfaites par S .*

Si P admet au moins une solution alors P est dit consistant ou satisfiable, sinon, P est inconsistant, insatisfiable ou sur-contraint.

Définition 5 (Espace de recherche). *L'ensemble de toutes les combinaisons de $D_1 \times D_2 \times \dots \times D_n$ est appelé espace de solutions ou espace de recherche dans la mesure où la solution doit être cherchée dans cet espace.*

L'espace de recherche d'un *CSP* est souvent représenté par une arborescence, ainsi, on appelle les méthodes qui le parcourent des méthodes de recherche arborescente.

1.2 Exemples de CSPs

Les exemples suivants sont d'un intérêt particulier, car ils sont des prototypes pour une grande classe de problèmes pratiques.

1.2.1 Le coloriage de graphe

Le problème de coloriage de graphe est utilisé pour modéliser certains types de problèmes d'ordonnancement et de gestion de ressources. Il consiste à colorier un graphe non-orienté, comprenant n noeuds, avec 3 couleurs. Chaque noeud doit être colorié avec une des trois couleurs disponibles de telle sorte que deux noeuds voisins n'aient pas la même couleur.

La figure 1.1 montre le problème de coloriage d'une carte de 8 régions avec 3 couleurs (*noir, blanc, gris*).

Les contraintes sont toutes du même type : ne pas colorier avec la même couleur les régions voisines.

Ce problème peut être modélisé comme problème de satisfaction de contraintes *CSP* (X, D, C) comme suit :

- $X = \{x_1, x_2, \dots, x_8\}$ qui représentent les différentes régions à colorier.
- $D = \{D_1, D_2, \dots, D_8\} / D_1 = D_2 = \dots = D_8 = \{\text{noir}, \text{blanc}, \text{gris}\}$
- $C = \{x_i \neq x_j / x_i \text{ et } x_j \text{ sont 2 variables de } X \text{ correspondant à des régions voisines}\}.$



FIGURE 1.1 – graphe colorié

1.2.2 Les n-reines

Ce problème consiste à placer sur un échiquier, de $n \times n$ cases, n reines qui ne s'attaquent pas (suivant les règles classiques des échecs). Il s'agit d'un

problème équivalent à la recherche d'un ensemble intérieurement stable (qui sera maximum) dans un graphe de $n \times n$ noeuds (un noeud pour chaque case), chaque arête correspondant à une diagonale, une verticale ou une horizontale. Il peut être envisagé pour d'autres types de pièces.

Ce problème peut être modélisé, en tant que *CSP*, de plus qu'une seule manière, et dans la mesure où l'on sait dès le départ qu'il y aura une reine et une seule sur chaque colonne de l'échiquier, la modélisation la plus efficace consiste à associer une variable x_i à chaque colonne i de telle sorte que x_i désigne le numéro de ligne sur laquelle placer la reine de la colonne i .



FIGURE 1.2 – Les 4 reines

Ainsi, pour le *CSP* de 4 reines dans la figure 1.2, la modélisation est la suivante :

- $X = \{x_1, x_2, x_3, x_4\}$
- $D = \{D_1, D_2, D_3, D_4\}$ tels que : $D_1 = D_2 = D_3 = D_4 = \{1, 2, 3, 4\}$
- $C = \{C_{lig}, C_{dm}, C_{dd}\}$ tels que :
 - $C_{lig} = \{x_i \neq x_j \mid i, j \in \{1, 2, 3, 4\} \text{ et } i \neq j\}$: les reines doivent être sur des lignes différentes.
 - $C_{dm} = \{x_i + i \neq x_j + j \mid i, j \in \{1, 2, 3, 4\} \text{ et } i \neq j\}$: les reines doivent être sur des diagonales montantes différentes.
 - $C_{dd} = \{x_i - i \neq x_j - j \mid i, j \in \{1, 2, 3, 4\} \text{ et } i \neq j\}$: les reines doivent être sur des diagonales descendantes différentes.

1.2.3 La planification de réunion

Le problème de planification de réunion (*MSP* pour *Meeting Scheduling Problem*) [52] est un processus de prise de décision qui consiste à organiser des réunions entre des agents, en déterminant les lieux et les horaires, tout en respectant leurs différents calendriers. Il a été défini sous plusieurs versions et avec différents paramètres, tels la durée des réunions [79], les préférences des agents [66], etc. Dans ce problème, un ensemble d'agents souhaitent se rencontrer. Ils cherchent un rendez-vous possible qui satisfasse les contraintes privées de chacun d'eux, tout en prenant en considération les contraintes de temps d'arriver entre les différentes réunions du même agent. La définition générale de ce problème est comme suit :

- Un ensemble A de m agents

- Un ensemble R de n réunions
- La durée de chaque réunion r_i est d_i
- A chaque réunion r_i , un ensemble d'agents $r_i.a$ doit participer
- Ainsi, chaque agent a_k a une série de réunions auxquelles il doit assister
- Chaque réunion r_i est associée à un emplacement $r_i.e$
- Les horaires prévus pour les réunions doivent permettre aux agents participants de se déplacer entre les réunions.

La possibilité de déplacement d'un agent, entre deux réunions, requiert une contrainte nommée *arrival-time*. La contrainte *arrival-time* entre deux réunions r_i et r_j , où au moins un même agent a_k doit participer, est définie comme suit :

$$|\text{temps}(r_i) - \text{temps}(r_j)| - d_i > \text{dplacement}(r_i.e, r_j.e) \quad (1.1)$$

Par conséquent, le problème de planification de réunions peut être représenté comme un *CSP* de la manière suivante :

- $X = \{r_1, \dots, r_n\}$ est l'ensemble de variables, chaque variable représente une réunion.
- $D = \{D_1, \dots, D_n\}$ est l'ensemble des domaines, où D_i est le domaine de la variable r_i . D_i est l'intersection des *time-slots* (les horaires hebdomadaires) à partir des propres calendriers de tous les agents qui doivent assister à r_i , autrement :
 $D_i = \bigcap_{a_j \in r_i.a} \text{calendrier}(a_j)$.
- C est l'ensemble des contraintes *arrival-time*. Une contrainte *arrival-time* existe pour toute paire de réunions (r_i, r_j) si au moins un agent participe aux deux.

1.3 Stratégies de résolution de CSPs

De nombreux efforts ont été déployés pour augmenter l'efficacité des algorithmes de satisfaction des contraintes, d'où les nombreuses méthodes, complètes et incomplètes, de résolution de *CSPs*, ainsi que les heuristiques d'ordonnancement utilisées pour améliorer le comportement de ces méthodes. Nous pouvons classer les méthodes complètes de résolution de *CSPs*, du point de vue de l'exploration, en trois catégories [26] :

- Algorithmes à exploration systématique : L'algorithme le plus courant dans cette catégorie est le *BackTracking* (*BT*). Le *BT* traverse l'espace de recherche en profondeur d'abord [43]. Il démarre avec un ordre pré-établi des variables du *CSP* et des valeurs de chaque variable dans son domaine. L'algorithme affecte des valeurs aux variables, une après l'autre, et teste à chaque fois si l'affectation partielle courante est localement consistante. Lors de son exécution, Le *BT* fait un retour

en arrière quand une affectation partielle ne pourra faire partie d'une solution.

- Algorithmes prospectifs (*lookahead*) : Après chaque affectation de valeur à une variable, les algorithmes prospectifs éliminent des domaines des variables non encore instanciées les valeurs qui conduisent d'évidence à un échec. Parmi ces approches, on peut citer le *Forward Checking (FC)* et le *Maintaining Arc Consistency (MAC)* [13].
- Algorithmes rétrospectifs (*lookback*) : Ces algorithmes sont incorporés dans le processus de recherche lui-même pour réaliser un retour-en-arrière intelligent. Ils utilisent les informations implicites contenues dans les situations d'échec pour sélectionner un meilleur point de retour. Parmi eux, on cite le *BackJumping* [26], le *Graph-based BackJumping*, le *Conflict-directed BackJumping* [59] et le *Dynamic BackTracking* [35].

Dans la même catégorie des méthodes complètes, on trouve d'autres types d'approches intéressantes, comme les algorithmes de filtrage. Ces approches sont utilisées d'abord dans une phase de pré-traitement pour améliorer la performance de l'algorithme de recherche. Une procédure de filtrage augmente la cohérence locale d'un *CSP*. Elle transforme un *CSP* en un autre *CSP* dont l'espace de recherche est réduit par rapport à celui du *CSP* d'origine mais dont l'ensemble de solutions est le même. Il ne perd pas d'information dans la transformation, mais le nouveau *CSP* contient explicitement certaines contraintes qui étaient implicites dans le *CSP* d'origine. Cette explicitation de contraintes évitera à une procédure de recherche de parcourir certaines zones de l'espace de recherche qui mèneraient sûrement à un échec. Les algorithmes de filtrage ou de cohérence locale ne résolvent pas complètement un *CSP* mais éliminent les inconsistances locales, qui autrement auraient pu être découvertes plusieurs fois par les procédures de recherche de solution. Parmi eux, les niveaux de filtrage les plus répandus sont la consistance d'arc [12], la consistance de chemin [56], la *k*-consistance [31]. Les méthodes prospectives s'appuient sur les algorithmes de consistance d'arc, qui est utilisée par de nombreuses méthodes tels *AC3* [46], *AC4* [55], *AC5* [40], *AC6* [12], *AC7* [14], *AC2001* [16], etc. L'algorithme de base et le plus connu est le *AC3* [46].

Dans ce qui suit, on va détailler les méthodes complètes, les plus courantes, de résolution de *CSP*.

1.3.1 Algorithmes à exploration systématique

BackTracking (BT)

La figure 1.3 montre l'exécution de l'algorithme *BackTracking (BT)* chronologique. La fonction s'exécute sous forme d'une procédure récursive. Elle est appelée avec l'affectation vide et affecte les variables une par une. Si

```

Function BT( $(X, D, C), A$ )
1.  if  $A$  not consistent then
2.    return False;
3.  if all variables of  $X$  are assigned then
4.    return True;
5.  else
6.    choose a variable  $x_i \in X \setminus \text{vars}(A)$ ;
7.    for all  $v_i \in D_i$  do
8.      if  $\text{BT}(A \cup (x_i = v_i), (X, D, C)) = \text{True}$  then
9.        return True;
10.   return False;

```

FIGURE 1.3 – Description du *BackTracking*

la fonction détecte que l'affectation courante est non consistante, à cause de la valeur affectée à la dernière variable choisie, elle lui change de valeur (lignes 1, 2 et 7). Si une variable n'a plus de valeur dans son domaine (situation de *dead-end*), elle retourne vers la variable instanciée juste avant pour lui changer de valeur (ligne 10). Si toutes les variables sont affectées, une solution est trouvée et la recherche est alors terminée (lignes 3, 4 et 9). Si l'affectation est consistante mais incomplète, l'approche choisit une variable non encore instanciée et lui affecte une valeur de son domaine et teste à nouveau la consistance avec l'affectation courante (lignes 5 à 9).

Le principe du *BT* peut être résumé par les trois actions suivantes :

- génération d'une affectation partielle consistante.
- extension de l'affectation partielle avec l'affectation d'une nouvelle variable.
- si cette extension ne peut être consistante, on retourne en arrière pour modifier l'affectation de la variable instanciée juste avant.

1.3.2 Algorithmes rétrospectifs (lookback)

Conflict-directed BackJumping (CBJ)

Le concept du *backjump* est l'un des outils utilisés par les algorithmes rétrospectifs. Dans la forme standard du simple retour en arrière (*backtrack*), chaque fois qu'un *dead-end* a lieu, l'algorithme change la valeur de la variable la plus récemment instanciée. Par contre, les algorithmes du *backjump* identifient et font un saut directement vers la variable responsable du *dead-end*. Ainsi, cette variable est ré-instanciée, si c'est possible, sinon, ils remontent vers une variable plus ancienne (par un saut ou un simple *backtrack* suivant l'approche utilisée).

Gaschnig a conçu le *backjump* (*BJ*) [33], qui est le premier algorithme non-chronologique explicite. *BJ* enregistre pour toute variable x_i , la variable x_j la plus récemment instanciée qui est en conflit avec elle. Quand un *dead-end* dans le domaine de x_i a lieu, *BJ* fait un saut directement vers x_j pour lui

changer de valeur. Si aucune valeur n'est possible pour x_j , BJ fait un simple *backtrack* vers la variable instanciée juste avant x_j .

Prosser a proposé le *Conflict-directed BackJumping (CBJ)* [59] qui rectifie le mauvais comportement du BJ quand la variable sur laquelle le saut est fait n'a plus de valeur. La figure 1.4 montre l'exécution de l'algorithme

```

Function CBJ( $(X, D, C), A$ )
1.  if  $A$  complete then
2.    return  $A$ ;
3.  else
4.    choose a variable  $x_i \in X \setminus vars(A)$ ;
5.     $S[i] \leftarrow \emptyset$ ;
6.     $D_i \leftarrow D_i^0$ ;
7.    for all  $v_i \in D_i$  do
8.       $x_i \leftarrow v_i$ ;
9.      if isConsistent( $A \cup (x_i = v_i)$ ) then
10.         $CS \leftarrow CBJ((X, D, C), A \cup (x_i = v_i))$ ;
11.        if  $x_i \notin CS$  then
12.          return  $CS$ ;
13.        else
14.           $S[i] \leftarrow S[i] \cup CS \setminus x_i$ ;
15.      else
16.        delete  $v_i$  from  $D_i$ ;
17.        let  $C_{ij}$  the oldest constraint violated by  $(x_i = v_i)$ ;
18.         $S[i] \leftarrow S[i] \cup x_j$ ;
19.  return  $S[i]$ ;

```

FIGURE 1.4 – Description de *CBJ*

Conflict-directed BackJumping (CBJ). Au lieu d'enregistrer seulement la variable coupable la plus proche, *CBJ* enregistre, pour toute variable x_i , l'ensemble des variables qui ont causé le rejet d'une des valeurs de x_i dans $S[i]$. A chaque fois qu'une variable x_i est choisie pour être instanciée (ligne 3), *CBJ* initialise $S[i]$ par l'ensemble vide. Ainsi, *CBJ* initialise le domaine courant de x_i par son domaine initial (ligne 5). Ensuite, une valeur v_i consistante avec l'affectation actuelle est cherchée pour la variable x_i . Si v_i est inconsistante avec l'affectation actuelle, elle est alors supprimée du domaine de x_i (D_i) (ligne 13), et x_j qui est la variable la plus ancienne causant l'inconsistance de la nouvelle valeur de x_i est alors ajoutée à $S[i]$ (ligne 15). Quand un *dead-end* se produit dans le domaine d'une variable x_i , *CBJ* fait un saut vers la dernière variable instanciée x_j existant dans $S[i]$ (lignes 16, 9 et 10). L'information dans $S[i]$ est remontée vers $S[j]$ (ligne 11). Par conséquent, *CBJ* effectue une forme de *backtrack* intelligent vers la source du conflit permettant à la procédure d'éviter des tests inutiles que peut causer le simple *backtrack* si un *dead-end* se produit au niveau de x_j . Dans ce cas, *CBJ* fait un saut vers la dernière variable instanciée existant dans $S[j]$. Au moment d'un saut de x_i vers x_j , *CBJ* supprime les valeurs de toutes les variables existants entre x_i et x_j , et perd ainsi un effort considérable pour ré-

affecter ces variables, d'où l'idée du *Dynamic Backtracking* (*DBT*) proposé par Ginsberg [35].

Dynamic BackTracking (DBT)

Le *Dynamic Backtracking* [35] (*DBT*) est une amélioration du *CBJ*. Son but est d'éviter d'affecter des variables déjà affectées juste à cause de leur emplacement après la variable causant le conflit.

DBT, comme son nom l'indique, utilise un ordre dynamique. Pour garantir la complétude tout en étant dynamique, l'approche utilise les *nogoods*.

Définition 6 (Nogood). *Un nogood [36] est une expression de la forme :*

$$(X_1 = v_1) \wedge \dots \wedge (X_k = v_k) \Rightarrow X \neq v \quad (1.2)$$

C'est une explication d'échec qui peut être utilisée pour représenter une contrainte comme implication. Un nogood est logiquement équivalent à la contrainte :

$$\neg[(X_1 = v_1) \wedge \dots \wedge (X_k = v_k) \wedge (X = v)] \quad (1.3)$$

qui représente un ensemble d'affectations qui ne peuvent pas être étendues à une solution du problème.

DBT exploite d'une façon efficace la mémorisation des *nogoods*. En fait, à chaque valeur v de chaque variable x est associé un *nogood* Ng_x^v , initialisé au vide, interdisant l'extension d'une affectation avec $x = v$. Ce *nogood* est souvent appelé explication ou justification de rejet de la valeur v . Généralement, il y a plusieurs explications possibles pour le retrait d'une valeur, mais la mémorisation de toutes ces justifications conduirait à une complexité spatiale exponentielle. La figure 1.5 présente l'exécution du *DBT*. Dans cet algorithme, le domaine courant d'une variable correspond à l'ensemble des valeurs qui ont une justification de rejet (un *nogood*) vide. *DBT* réalise la boucle récursive principale qui cherche l'extension d'une affectation A , initialement vide, en affectant des valeurs aux variables futures tant que A n'est pas complète (lignes 3 à 6). La fonction *DBT – Evaluate()* (ligne 7) détermine si la nouvelle affectation est consistante (ligne 18). Dans ce cas, le *nogood* retourné est vide, l'affectation A est étendue et *DBT* est appelée à nouveau (lignes 19, 8 et 9). Sinon, cette fonction retourne un *nogood* expliquant l'échec, en transformant la contrainte violée en un *nogood* grâce à la méthode *transformNogood(c)* (lignes 20 à 22). Si la partie droite de ce *nogood* ($rhs(nogood)$) est la variable courante, le nouveau *nogood* est ainsi enregistré (lignes 10 et 11). Tant que $rhs(nogood)$ est une autre variable que celle courante, ses valeurs interdites par la variable courante sont libérées par la suppression des *nogood* causant leur rejet et le retour en arrière est déclenché à nouveau, ainsi, *DBT* saute récursivement vers une affectation partielle consistante (lignes 12 à 16). Notons que Ng_i^v est le *nogood* associé à la valeur v de x_i .

Function DBT($(X, D, C), A$)

1. **if** all variables of X are assigned **then**
2. *return* A ;
3. **else**
4. choose a variable $x_i \in X \setminus \text{vars}(A)$;
5. **for all** $v \in D_i$ **do**
6. $x_i \leftarrow v$;
7. $\text{nogood} \leftarrow \text{DBT-Evaluate}(A \cup (x_i = v))$;
8. **if** $\text{nogood} = \emptyset$ **then**
9. $\text{nogood} \leftarrow \text{DBT}(A \cup (x_i = v))$;
10. **if** $(x_i = v) = \text{rhs}(\text{nogood})$ **then**
11. store nogood as justification of rejection of v ;
12. **else**
13. **for all** $u \in D_i$ **do**
14. **if** $\text{rhs}(\text{nogood}) \in Ng_i^u$ **then**
15. $Ng_i^u \leftarrow \emptyset$;
16. *return* nogood ;
17. *return* $\cup_{v \in D_i} Ng_i^v$;

Function DBT-Evaluate(A)

18. **if** A consistent **then**
19. *return* $\emptyset$;
20. **else**
21. choose a constraint c violated by A ;
22. *return* $\text{TransformNogood}(c)$;

 FIGURE 1.5 – Description de *DBT*

1.3.3 Algorithmes prospectifs (lookahead)

Forward Checking (FC)

Le principe du *Forward Checking* [38] (*FC*) est de propager l'information concernant les variables affectées vers celles non encore affectées. Ainsi, il est classifié parmi les algorithmes prospectifs (*lookahead*). Comme décrit

Function FC($(X, D, C), A$)

1. **if** all variables of X are assigned **then**
2. *return* *True*;
3. **else**
4. choose a variable $x_i \in X \setminus \text{vars}(A)$;
5. **for all** $v_i \in D_i$ **do**
6. **for all** $x_j \in X \setminus \text{vars}(A)$ **do**
7. $D'_j \leftarrow v_j$ of D_j | $A \cup \{(x_i, v_i), (x_j, v_j)\}$ is consistent
8. **if** D'_j is empty **then**
9. *return* *False*;
10. **if** $\text{FC}(A \cup (x_i, v_i), (X, D', C)) = \text{True}$ **then**
11. *return* *True*;
11. *return* *False*;

 FIGURE 1.6 – Description de *FC*

dans la figure 1.6, FC affecte les variables une par une. Tant qu'il existe des variables non encore affectées, il choisit une de ces variables et lui affecte une valeur de son domaine (lignes 3 à 5). Il procède ainsi par filtrer les domaines des variables futures pour ne garder que les valeurs consistantes avec la valeur courante de la variable actuelle (lignes 6 et 7). Si un de ces domaines est vide, la fonction retourne *Faux* afin de changer la valeur de la variable courante (8 à 10 et 5). FC s'arrête une fois l'affectation consistante est complète (1, 2, 10 et 11), ou une fois la première variable affectée n'a plus de valeur dans son domaine (lignes 8 et 9). On note que pour toute variable x_i , un domaine D'_i est associé. Les suppression des valeurs concernent ce domaine et non pas D_i afin de pouvoir récupérer les anciennes valeurs de x_i après un retour arrière.

Maintaining Arc Consistency (MAC)

Le maintien d'arc consistance (MAC [17] pour *Maintaining Arc Consistency*) s'appuie sur les algorithmes de consistance d'arc.

Définition 7 (Consistance d'arc). *Un réseau de contraintes est arc consistant si et seulement si pour toute valeur v_i , dans le domaine D_i de la variable x_i , il existe dans le domaine D_j , de chaque variable adjacente x_j , une valeur v_j qui soit compatible avec v_i . v_j est dite alors support de v_i dans D_j .*

Si un arc $\{x_i, x_j\}$ (c-à-d une contrainte c_{ij}) n'est pas arc consistant, il peut l'être en supprimant toutes les valeurs des domaines des variables, concernées par la contrainte, pour lesquels aucun support n'existe dans l'autre domaine. Il est évident que ces suppressions maintiennent les solutions du problème, puisque les valeurs supprimées n'apparaissent en aucune solution. Le filtrage par consistance d'arc correspond à une étape préalable à la recherche de solution ; l'espace de recherche est limité, puis la recherche est lancée.

Dans le MAC , l'idée de la consistance d'arc est étendue en alternant le filtrage et l'exploration, ainsi, à chaque étape de la recherche, une nouvelle phase de filtrage est relancée. La propriété de consistance d'arc n'est plus obtenue une seule fois préalablement à la recherche, mais est en quelque sorte maintenue tout au long de la recherche.

Le principe du maintien de consistance d'arc est le suivant : l'affectation d'une variable par une valeur de son domaine est considérée comme la réduction de son domaine à la valeur choisie ; cette réduction est donc propagée pour obtenir la consistance d'arc. Ce processus est répété lors de chaque affectation.

Le MAC permet d'élaguer considérablement l'espace de recherche, et semble présenter un bon compromis entre le sur-coût causé par le filtrage et la réduction de l'espace de recherche. Parmi les algorithmes connus et efficaces

utilisant ce principe, on cite *MAC-3* qui utilise le filtrage par *AC3*, ou encore mieux *MAC-2001* qui utilise *AC2001*.

L'exécution de l'algorithme *AC3* est illustrée dans la figure 1.7. *AC3* cherche

```

Function AC3((X,D,C))
1.   $Q \leftarrow \{(x_i, x_j) / \exists C_{ij} \in C\}$ ;
2.  while  $Q \neq \emptyset$  do
3.     $Q \leftarrow Q \setminus (x_i, x_j)$ ;
4.    if REVISE(( $x_i, x_j$ ), (X, D, C)) then
5.       $Q \leftarrow Q \cup \{(x_k, x_i) / \exists C_{ik} \in C \text{ et } x_k \neq x_i \text{ and } x_k \neq x_j\}$ ;
6.  return (X, D, C);
Function REVISE((X, D, C), ( $x_i, x_j$ ))
7.  DELETE  $\leftarrow$  False;
8.  for all  $v_i \in D_i$  do
9.    if no  $v_j$  of  $D_j$  satisfies binary constraints between  $x_i$  and  $x_j$  then
10.      delete  $v_i$  from  $D_i$ ;
11.      DELETE  $\leftarrow$  True;
12. return DELETE;

```

FIGURE 1.7 – Description de *AC3*

un support pour chaque valeur sur chaque contrainte afin de déterminer celles qui ne sont pas arc consistantes, et qui ne peuvent alors apparaître dans une solution. Pour cela, il emploie une liste Q des paires de variables (x_i, x_j) concernées, c-à-d celles liées par une contrainte (ligne 1). Pour chaque paire de variables $(x_i, x_j) \in Q$ (lignes 2 et 3), *AC3* utilise la procédure *Revise*(x_i, x_j) (lignes 4 et 7 à 12) pour déterminer l'existence d'au moins un support pour chaque valeur de D_i sur la contrainte C_{ij} . Si une valeur n'ayant pas de support est supprimée, *AC3* estime l'ensemble des valeurs qui peuvent ne plus être arc inconsistantes à cause de cette suppression. En fait, si une valeur (x_i, v_i) est supprimée parce qu'elle n'a pas de support sur C_{ij} , il se peut que (x_i, v_i) soit l'unique support de certaines de ces valeurs (x_k, v_k) et que la suppression de (x_i, v_i) les rende arc inconsistantes. Ainsi, pour chaque contrainte $C_{ik} \in C$ avec $k \neq j$, *AC3* réétudie l'arc inconsistance de toutes les valeurs v_k de D_k par rapport à C_{ik} en plaçant la paire de variables (x_k, x_i) dans Q (ligne 5).

La différence entre *AC3* et son extension *AC2001/3.1*, appelé *AC2001* par [16] et *AC3.1* par [44] et proposé indépendamment par leurs auteurs respectifs, est simple mais importante. En fait, lorsqu'un support doit être trouvé pour une valeur, *AC3* commence la recherche à partir du premier tuple tandis que *AC2001/3.1* commence la recherche à partir d'un point de reprise qui correspond au dernier support trouvé pour cette valeur. Plus précisément, *AC2001/3.1* vérifie la propriété d'unidirectionalité positive et négative.

Définition 8 (unidirectionalité). *Dans tout algorithme de consistance d'arc, un test de consistance, noté $C(t)$, est toujours réalisé pour un tuple donné*

et par rapport à un triplet $(C; X; a)$ où C désigne une contrainte, X une variable de $\text{vars}(C)$ et a une valeur de $\text{dom}(X)$.

- **unidirectionalité positive** : $C(t)$ n'est pas testé s'il existe un support t' de $(X; a)$ dans C déjà testé avec succès par rapport à $(C; X; a)$.
- **unidirectionalité négative** : $C(t)$ n'est pas testé s'il a déjà été testé sans succès par rapport à $(C; X; a)$.

Algorithm 1 Init2001()

1. $\forall C \in \text{constraints}, \forall X \in \text{vars}(C), \forall a \in \text{dom}(X)$
2. $\text{last}[C, X, a] \leftarrow \text{nil}$;

Algorithm 2 REVISE2001(in C, X)

3. $\text{nbElements} \leftarrow |\text{dom}(X)|$;
4. **for all** $a \in \text{dom}(X)$ **do**
5. **if** $\text{last}[C, X, a]$ is valid **then** continue;
6. $\text{seekNextSupport}(C, X, a, \text{last}[C, X, a])$;
7. **if** $\text{last}[C, X, a] = \text{nil}$ **then** remove a from $\text{dom}(X)$;
8. **return** $\text{nbElements} \neq |\text{dom}(X)|$;

FIGURE 1.8 – Description de AC2001

Le AC2001, moins naïf que AC3, nécessite l'introduction d'une structure de données qui permet de stocker le dernier support trouvé pour toute valeur du domaine valide de chaque variable, par rapport aux variables voisines. Ainsi, comme illustré dans la figure 1.8, AC2001 supporte la nouvelle structure de données appelée *last*. Cette structure est un tableau qui permet de stocker le dernier support trouvé pour tout triplet $(C; X; a)$ composé d'un arc $(C; X)$ et d'une valeur appartenant à $\text{dom}(X)$. Initialement, la structure *last* doit être initialisée au vide (voir l'algorithme 1). La révision (voir l'algorithme 2) consiste à tester pour toute valeur la validité du dernier support trouvé (le vide n'est pas valide) et à rechercher potentiellement un nouveau support. Il faut noter que *seekNextSupport* met à jour le paramètre t avec le plus petit support de $(X; a)$ dans C strictement plus grand que lui ou avec le vide (rappelons qu'un test de consistance est noté $C(t)$).

1.3.4 Heuristiques d'ordonnancement

Les heuristiques d'ordonnancement sont des règles qui déterminent l'ordre suivant lequel on va choisir les variables pour les affecter ainsi que l'ordre suivant lequel on va choisir les valeurs pour les variables. En général, une heuristique d'ordonnancement concerne :

- l'ordre sur les variables ;
- l'ordre sur les valeurs ;

Une bonne heuristique d'ordonnancement peut avoir un grand impact sur la résolution d'un problème de décision et permettre d'accélérer le processus de

recherche de solution (la détection d'échecs en cas de problèmes insolubles) pendant l'exploration de l'espace de recherche.

Ainsi, on évite que de grandes parties de l'arbre de recherche soient explorées inutilement.

De plus, si l'on veut favoriser l'apparition d'une solution, les variables et les valeurs sont ordonnées par l'heuristique de telle façon que, si une solution existe, elle sera associée à un chemin qui se trouve, par convention, le plus à gauche possible dans l'arbre de recherche et sera ainsi atteinte au plus tôt lors d'une exploration en profondeur d'abord.

Plusieurs travaux se sont intéressés aux heuristiques, que ce soit dans un contexte *SAT*, *CSP* ou optimisation combinatoire [70], [44].

Une heuristique peut être statique ou dynamique selon que les ordres de variables et de valeurs sont fixés au préalable ou évoluent pendant la résolution, de telle sorte qu'elle soit affectée par les changements que subit l'espace de recherche. Ci-dessous, nous présentons quelques exemples d'heuristiques.

Les heuristiques de choix de variables

A chaque étape de la construction d'un arbre de recherche, lors de la résolution d'un *CSP*, l'algorithme doit choisir une variable non encore affectée et lui affecter ainsi une valeur de son domaine.

(L'ordre dans lequel les variables sont choisies a un impact crucial sur les performances de la résolution. C'est pour cette raison que plusieurs heuristiques d'ordre de choix de variables ont été développées.

En fait, l'application de différentes heuristiques de choix de variables pour résoudre un même problème peut conduire à des résultats extrêmement variés en terme d'efficacité. Une heuristique idéale consisterait à sélectionner en priorité un *backdoor* [82], c'est à dire un ensemble de variables qui, une fois assignées, rendent le problème facile à résoudre.

Les heuristiques généralement utilisées peuvent être classées en deux catégories. La première correspond aux heuristiques statiques (*SVOs* pour Static Variable Ordering) qui conservent le même ordre de priorité tout au long de la recherche, ordre établi à partir de la structure du problème dans son état initial.

Les heuristiques statiques les plus courantes sont les suivantes :

- *lexico* : les variables sont ordonnées suivant l'ordre lexicographique ;
- *deg* : les variables sont ordonnées de manière décroissante en fonction de leur degré initial [28], qui est défini comme étant le nombre de voisins du chaque variable ;
- *width* : les variables sont ordonnées de manière à limiter la largeur du graphe de contraintes [32].

La seconde catégorie correspond aux heuristiques dynamiques (*DVOs* pour Dynamic Variable Ordering) qui exploitent différentes informations sur l'état

courant du problème.

Dans cette catégorie, les heuristiques les plus courantes sont :

- *dom* : elle consiste à ordonner les variables de manière croissante suivant la taille courante de leur domaine [38]. Son utilisation est justifié par le principe du *fail-first* ;
- *ddeg* : les variables sont ordonnées de manière décroissante en fonction de leur degré courant (dit dynamique). Le *ddeg* d'une variable [7] est le nombre de contraintes qui la lient avec au moins une variable non encore instanciée ;
- *wdeg* : Les variables sont ordonnées de manière décroissante en fonction de leur degré pondéré [19].

C'est une heuristique dirigée par les conflits. Quand une contrainte cause un vidage de domaine, durant la recherche, son poids est incrémenté par un. Chaque variable a un degré pondéré, qui est la somme des poids sur toutes les contraintes dans lesquelles elle participe.

Ces différentes heuristiques sont également combinées entre elles, on retrouve classiquement l'utilisation de l'heuristique d'ordre sur les variable $dom \oplus ddeg$, qui évalue, en cas d'égalité de *dom*, en utilisant *ddeg*, ou l'utilisation de dom/deg ou encore de $dom/wdeg$.

Dans la plupart des cas, les algorithmes de résolution qui utilisent des heuristiques d'ordre assurent une résolution plus rapide que celle obtenue par des algorithmes sans heuristique d'ordre.

Même si les différentes heuristiques d'ordre sont en général incomparables entre elles, dans de nombreux cas (et notamment sur des problèmes structurés) l'heuristique $dom/wdeg$ fournit de très bons résultats.

Les heuristiques de choix de valeurs

Les heuristiques de choix de valeurs sont utilisées pour choisir une valeur du domaine d'une variable pour l'affecter durant la recherche. En fait, si la valeur sélectionnée a une probabilité plus élevée d'être partie d'une solution, le choix de cette valeur peut faire une différence significative en terme de temps de résolution. Ainsi, ces heuristique ordonnent les valeurs en choisissant d'abord une valeur qui permettra à l'affectation partielle courante d'être étendue à une solution.

Ces heuristiques peuvent être classées, comme pour celles de choix de variables, en deux catégories : les heuristiques dynamiques et celles statiques. Pour la première catégorie, l'heuristique la plus connue est celle du conflit minimal *conf*. En fait, pour juger la participation d'une valeur à une solution, l'heuristique *conf* considère le nombre de valeurs qui ne sont pas en conflit avec la valeur en question. Ainsi, la valeur qui est compatible avec la plupart des valeurs des variables non encore instanciées est testée en premier. Concernant la deuxième catégorie, l'idée est d'attribuer, avant la recherche,

un poids à chaque valeur du domaine de chaque variable, puis réarranger les valeurs dans leurs domaines respectifs dans l'ordre décroissant de leur poids. L'ordre des valeurs n'est calculé qu'une seule fois. En général, de nombreux poids sont possibles. Nous présentons les trois poids suivants [49] :

$$weight[x, a] = \sum_{(x,y) \in C} scount[x, y, a] \quad (1.4)$$

$$weight[x, a] = \sum_{(x,y) \in C} scount[x, y, a] / |D_{ac}(y)| \quad (1.5)$$

$$weight[x, a] = \prod_{(x,y) \in C} scount[x, y, a] \quad (1.6)$$

Notons que $scount[x, y, a]$ est le nombre de supports de $[x, a]$ dans $D_{ac}(y)$, le domaine arc consistant de y .

La première équation sert à calculer, pour chaque valeur a du domaine de x , le nombre de ses supports dans les domaines arc consistants des variables voisines $D_{ac}(y)$. La deuxième sert à calculer le nombre de supports de chaque $[x, a]$ par rapport au nombre maximal de supports possibles. Et la troisième permet de calculer l'espérance que la valeur a de x contribue à une solution.

Chapitre 2

Contribution aux CSPs

Le problème d'allocation de ressources est l'un des prototypes des problèmes réels. Dans [10], nous avons modélisé, sous forme de *CSP*, un problème pratique d'allocation de ressources, qui consiste à répartir les étudiants de la Faculté de Médecine et de Pharmacie de Casablanca (*FMPC*) aux différents services hospitaliers du *CHU Ibn Rochd*.

En fait, les stages hospitaliers sont une composante essentielle et centrale de la formation médicale, afin d'acquérir les différentes compétences cliniques, ils nécessitent un engagement de l'étudiant d'une part, et du superviseur d'autre part.

En fait, le deuxième cycle des études médicales à la *FMPC* se concentre sur l'acquisition de ces compétences cliniques et thérapeutiques, et le problème rencontré dans la gestion des cours hospitaliers consiste à adapter le nombre d'étudiants à la capacité d'accueil, sachant que l'affectation des étudiants aux services hospitaliers doit être soumise à certaines contraintes.

en effet, puisque le formalisme des *CSP* peut être simplement, clairement et correctement adapté à une grande classe de problèmes réels, nous avons pensé à utiliser la *PPC* pour réussir cette répartition des étudiants sur les services hospitaliers.

Dans ce qui suit, nous présentons la composante de la formation hospitalière dans le curriculum de l'étudiant-médecin, après nous proposons notre modélisation de ce problème en tant que *CSP*.

2.1 Description de la problématique

Le problème rencontré est d'allouer les étudiants de la *FMPC* aux hôpitaux de la ville, essentiellement aux services du complexe hospitalier universitaire (*CHU*), tout en satisfaisant les contraintes imposées [22].

Nous notons que, pour résoudre ce problème d'allocation, l'administration de la *FMPC* utilise, avant le début de chaque année, une application de gestion d'affectation des étudiants, qui est modélisée en utilisant le formalisme de la programmation orientée objet (*OOP* pour *Object-oriented programming*) [67] et développée avec la plate-forme *DotNet* [61]. Par exemple, pour une population de 500 étudiants et environ 20 services, avec une capacité moyenne de 55 lits par service, l'application actuelle prend entre 7 et 10 jours pour déclarer une solution. Cependant, le résultat obtenu est toujours incomplet (certains étudiants restent sans service en raison de l'épuisement de toutes les capacités des services), raison pour laquelle l'administration complète manuellement la distribution en violant certaines contraintes.

Voici quelques définitions intéressantes qui seront utiles par la suite :

Définition 9 (Période p). Une période p est une période fixe d'un mois et demi (6 semaines).

Définition 10 (Service complémentaire). Un service complémentaire est un service qui dure une période.

Les services de cardiologie, néphrologie et dermatologie, par exemple, sont des services complémentaires.

Définition 11 (Service fondamental). *Un service fondamental est un service qui occupe 2 périodes. Autrement, c'est un couple $\langle SF_1, SF_2 \rangle$, où chaque composant SF_i , tel que $i \in \{1, 2\}$, dure une seule période.*

Le service de pédiatrie, par exemple, est un service fondamental. Il est composé de pédiatrie₁ et pédiatrie₂, et chaque composant dure une seule période.

Définition 12 (Service obligatoire). *Un service obligatoire est un service par lequel chaque étudiant doit passer.*

Définition 13 (Service optionnel). *Un service optionnel est tout service qui n'est pas obligatoire.*

Un service, qu'il soit obligatoire ou optionnel, n'est pas nécessairement un service fondamental.

Les contraintes du problème peuvent être divisés en deux catégories, la première concerne les contraintes locales qui gèrent la relation de chaque étudiant avec ses propres six périodes, et la deuxième concerne les contraintes globales qui définissent la relation entre les étudiants et les services dans lesquels ils sont attribués.

- **Les contraintes locales** (concernant les étudiants)
 - chaque étudiant doit passer exactement 6 stages durant l'année (6 périodes) ; (C_1)
 - aucun étudiant ne doit revisiter un même service ; (C_2)
 - chaque étudiant doit passer par un nombre fixe de services obligatoires ; (C_3)
 - chaque étudiant ayant un service fondamental s_i (partie d'une paire $\langle s_i, s_j \rangle$) dans une période donnée t_k , tel que $k \in [1, 5]$, doit être affecté pour la prochaine période t_{k+1} au service s_j . (C_4)
 - aucun étudiant ne doit avoir la première partie d'un service fondamental comme dernier stage (6^{me} période) ; (C_5)
- **Les contraintes globales** (concernant les services) :
 - chaque service est caractérisé par sa capacité qui ne doit pas être dépassée ; (C_6)

2.2 Modélisation du problème en tant que CSP

Nous avons choisi de formuler ce problème comme un *CSP* pour les raisons suivantes :

- Le problème de l'affectation des étudiants par des services est trop complexe pour être résolu manuellement ;

- De nombreuses exigences de cette affectation sont naturellement traduites par des contraintes ;
- Les techniques de programmation par contraintes ont montré leur succès pour des applications similaires à celle-ci, comme les problèmes d'allocation des ressources, les problèmes d'ordonnancement, ... etc.

Pour ce faire, nous avons traduit les différentes composantes du problème comme un ensemble $\langle Variables, Domaines, Contraintes \rangle$ de telle sorte que la modélisation puisse décrire exactement le problème. La modélisation peut se faire de plusieurs façons, nous avons proposé celle que nous voyons la plus claire, la plus simple et la plus efficace.

— **Variables :**

Une variable X_i^p du *CSP* est une période p pour un étudiant donné i , tels que : $i \in [1, \text{nombre des étudiants}]$ et $p \in [1, 6]$. (C_1)

nous choisissons les variables comme ci-dessus parce que la solution doit être une affectation complète de toutes les périodes pour tous les étudiants.

— **Domaines :**

Soit $D = \{s_1, s_2, \dots, s_n\}$, l'ensemble des n services du *CHU*, le domaine de toutes les variables X_i^p du *CSP*.

$X_i^p = s_j$ si l'étudiant i est affecté pour sa période p au service s_j .

— **Contraintes :**

— **contraintes locales :** $\forall$ étudiant i :

— C_i^1 :

$\forall k, l \in [1, 6]$ tel que $k \neq l$:

$X_i^k \neq X_i^l$ (C_2)

— C_i^2 :

S est la liste des services obligatoires et $m \in [1, \text{cardinal}(S)]$, alors :

$\sum_{p=1}^6 \text{estDans}(X_i^p, S) = m$,

tel que $\text{estDans}(a, A)$ est une fonction booléenne qui retourne 1 quand $a \in A$. (C_3)

— C_i^3 :

$\forall$ service fondamental $S = \langle s, s' \rangle$ avec deux unités s et s' , $\forall p \in [1, 5]$:

$X_i^p = s \Rightarrow X_i^{p+1} = s'$ (C_4)

— C_i^4 :

$\forall$ service fondamental $S = \langle s, s' \rangle$ avec deux unités s et s' :

$X_i^6 \neq s$ (C_5)

— **contraintes globales :**

— C_s :

$\forall s \in D, \forall p \in [1, 6] : occup_p(s) \leq capacite(s)$ (C_6)
tels que $occup_p(s)$ est le nombre des étudiants affectés au service s durant la période p , et $capacite(s)$ est la capacité du service s qui ne doit pas être dépassée.

2.3 implémentation du problème

Afin de mesurer la faisabilité et la rigueur de notre modélisation proposée, nous l'avons implémentée dans la plate-forme *Choco 2.0* [41], dans une machine *dual core* avec un processeur de *1.72 GHz* et une *RAM* de *2GB*. *Students* est la matrice des services de l'ensemble des périodes pour

```

1. IntegerVariable[][] Students = Choco.makeIntArray(
   "Students",NbStudents,6,services,Options.V_ENUM); //(C1)
2. for (int i=0; i<NbStudents; i++) {
3.   CSPModel model = new CSPModel();
4.   model.addConstraint(Choco.allDifferent(Students[i])); //(C3)
5.   model.addConstraint(Choco.among(m,Students[i],ListOfObligatoires)); //(C2)
6.   for (int j=0; j<NbOfPeriods-1; j++) {
7.     for (int l=0; l<ListOfFondamentals1.size(); l++){
8.       model.addConstraint(Choco.implies(
Choco.eq(Students[i][j],s1),Choco.eq(Students[i][j+1],s2))); //(C4)
9.       if (j==latest_period)
10.        model.addConstraint(Choco.notMember(
Students[i][j],ListOfFondamentals1)); //(C5)
        }
      }
11. model.addConstraint(Choco.occupationMax(k,Capacities.get(k),VarsOfStudents));
   //(C6)
12. CSPSolver solver = new CSPSolver();
13. solver.read(model);
14. solver.solve();

```

FIGURE 2.1 – Pseudo code

tous les étudiants, autrement, chaque $Student[i][j]$ est une période pour un étudiant, qui prend une valeur à partir de *services*. La matrice est créée par la fonction *makeIntArray* de la classe *main* de *Choco* (ligne 1).

La procédure boucle sur l'ensemble des étudiants (line 2), pour définir pour chacun les contraintes locales, et définit un modèle qui contient toutes les spécifications du problème courant (ligne 3). Par la suite, elle remplit le modèle par toutes les contraintes locales (lignes 4 à 10).

Après l'ajout des contraintes globales, concernant les capacités des services (ligne 11), toutes les contraintes sont ajoutées, ainsi, la procédure définit le *CSPSolver* (ligne 12) et lui passe le modèle (ligne 13) avant de démarrer le processus de résolution (ligne 14).

Chapitre 3

Les CSPs Dynamiques

3.1 formalisme des DCSPs

De nombreux problèmes réels interagissent avec leur environnement, ainsi, le système de contraintes les définissant n'est pas figé. On peut citer, comme exemples pour ces problèmes, les problèmes d'emplois du temps, où un nouvel emploi du temps est reconstruit quand un professeur a un empêchement, les problèmes d'ordonnancement en ligne, où il faut ré-ordonnancer lorsqu'une machine tombe en panne, etc. Les évolutions d'un problème dynamique doivent être prises en compte très rapidement pour refléter l'état réel du système. Traiter un système dynamique de contraintes consiste donc à résoudre successivement une suite de problèmes (statiques) résultant des évolutions.

Définition 14 (DCSP). *Soit P un CSP dynamique (DCSP) [27]. P est alors une séquence de CSPs statiques $P_0, \dots, P_\alpha, P_{\alpha+1}, \dots$, tel que chaque P_i diffère du problème précédant par l'ajout ou le retrait de quelques contraintes (tous les changements possibles dont subit un CSP peuvent être exprimés en terme d'ajout ou de suppression de contraintes).*

Si $P_\alpha = (X, D, C_\alpha)$, alors $P_{\alpha+1} = (X, D, C_{\alpha+1})$, où $C_{\alpha+1} = C_\alpha \pm c$, tel que c un ensemble de contraintes.

3.2 Méthodes de réparation de DCSPs

Les méthodes de réparation de DCSPs peuvent être classées en trois grandes catégories :

- Les méthodes heuristiques, qui consistent à établir une stratégie de choix d'ordre optimal, comme dans le cadre des CSPs.
- Les méthodes de réparation locales, qui consistent à se baser sur une affectation consistante précédente (complète ou non) et de la réparer, en utilisant une séquence de modifications locales (modifications d'une seule affectation de variable à la fois) [36, 18].
- Les méthodes qui visent à minimiser la distance entre les solutions successives [29, 39, 8, 60].

Les méthodes des deux premiers groupes s'intéressent à l'amélioration de l'efficacité de la recherche, en terme d'effort dépensé pour trouver la solution finale (problème après perturbation), alors que ceux du troisième groupe visent à maintenir la stabilité du problème.

3.2.1 Travaux connexes

En tant que première contribution en DCSPs, l'algorithme *Local Changes* (LC) [71], qui réutilise la solution d'un CSP précédent pour résoudre le nouveau CSP, a été proposé. Il commence par choisir une variable non affectée

(ou une variable existante dans les contraintes violées) et lui attribuer une valeur. Cependant, l'ensemble des variables qui seront affectées par une nouvelle valeur est plutôt arbitraire. Cela dépend des choix des nouvelles valeurs pour les variables non affectées et il n'y a aucun moyen pour garantir que cela aboutira à un effort minimal. Un peu plus tard, *Solution Stability (SS)* [78] a été proposé comme une stratégie pour générer des solutions qui devraient rester valides après les changements de contraintes dans un *DCSP*. Cette approche repose sur l'hypothèse que les changements successifs d'un *DCSP* sont temporaires. Cependant, dans les cas d'ajout de contraintes, il est assez compliqué de suivre les changements directement. Ainsi, cette approche peut devenir trop coûteuse pour être viable. Quelques années plus tard, une nouvelle version de *LC* [80] a été proposée. Cette approche est concentrée sur la nature du changement dans la performance de la recherche après la modification d'un problème et propose une combinaison entre l'algorithme *LC* [71] et les techniques de propagation des contraintes. Cette approche est basée sur l'heuristique de choix de valeurs *conf* et les structures de données utilisées dans l'implémentation basique du *MAC*. En utilisant cette méthode, la performance avec une heuristique donnée est toujours très imprévisible même après des petits changements dans le problème précédent. Nous notons également que les expérimentations ont été effectuées dans des *CSPs* générés aléatoirement, ce qui n'est pas suffisant pour donner une bonne conclusion. Dans [36], *Partial-order Dynamic Backtracking (PDB)* a été proposé en tant qu'une approche de réparation assez flexible pour réparer les *DCSPs*. Cette flexibilité est due à la flexibilité du choix de l'ordre partiel entre les variables, qui peut conduire à une réparation assez optimale en évitant le changement des parties difficiles du réseau de contraintes.

Dans ce qui suit, nous nous focalisons sur l'amélioration de l'efficacité de la recherche. Nous exploitons la flexibilité du *PDB* [36] pour proposer quelques approches de réparation plus efficaces.

Nous notons que les méthodes utilisées pour résoudre les *CSPs* peuvent également être utilisées pour résoudre les *CSPs* dynamiques, en les traitant exactement comme des *CSPs* statiques. En fait, les techniques les plus efficaces pour résoudre les *CSPs* sont celles basées sur la propagation des contraintes, qui regroupe trois classes d'algorithmes. La première concerne les algorithmes basés sur la consistance d'arc simple (*SAC* pour *singleton arc consistency*), à savoir le *singleton arc consistency (SAC)* [24] et le *neighbourhood SAC (NSAC)* [75]. Le second concerne les algorithmes basés sur la consistance de chemin (*PC* pour *path consistency*) : le *restricted path consistency (RPC)*, le *restricted-restricted path consistency (rRPC3)* [68] et le *Max restricted path consistency (maxRPC)* [25]. Et le dernier groupe concerne les algorithmes basés sur la consistance d'arc (*AC* pour *arc consistency*), à savoir les différentes versions de *MAC* [17]. Récemment, une étude approfondie [77] sur différents types de problèmes a comparé tous ces algorithmes. Elle a conclu que le prétraitement avec des niveaux de cohérence plus

élevés suivi du traitement hybride-AC, c-à-d *MAC*, qui alterne le processus de recherche et la stratégie de consistance d'arc (*AC*), donne les meilleurs résultats dans le traitement des *CSPs*, à l'exception des problèmes *Relop* [77] et des problèmes avec contraintes globales, ce qui n'est pas le cas des problèmes étudiés dans nos expérimentations. Bien évidemment, il existe de nombreux algorithmes *AC*, mais les algorithmes les plus référencés sont *AC-3* (en raison de sa simplicité) et *AC-2001/3.1* (en raison de sa simplicité et de son optimalité) [5]. En outre, la variante *AC-2001/3.1* de *AC-3* a une complexité de temps optimale [76], donc sa réputation et son utilisation dans diverses études comme référence.

3.2.2 Partial-order Dynamic Backtracking

notations

Les stratégies dynamiques d'ordonnancement offrent une flexibilité de construire un nouvel ordre entre les variables durant le processus de la recherche. Parmi les approches les plus connues dans ce sens, nous citons le *Partial-order Dynamic Backtracking (PDB)* [36]. Toutefois, la flexibilité de *PDB* concernant la stratégie de backtrack utilisée, surtout pour les *DCSPs* difficiles, n'est pas assez satisfaisante.

Le *PDB* remplace l'ordre fixe des variables, utilisé dans plusieurs approches de résolution de *CSPs*, par un ordre partiel dynamique, qui est partiellement construit et modifié durant le processus de la recherche. Le *PDB* utilise le concept du *nogood*, introduit dans le *Dynamic Backtracking (DBT)* [35]. Contrairement au *DBT*, quand un *nogood* est produit durant la recherche et qu'aucun ordre n'est imposé entre les variables concernées par ce conflit, *PDB* offre une flexibilité de choix concernant la variable qui constituera la conclusion de ce nouveau *nogood*. La performance de cette approche dépend essentiellement de la qualité de ce choix.

Pour deux variables quelconques, l'ordre n'est pas défini au début, mais il existe un ordre partiel qui est construit au fur et à mesure de la recherche, jusqu'à l'obtention de la solution si elle existe, ou la détection de l'insolvabilité du problème à travers un *nogood* vide. L'ordre partiel entre les variables est représenté par les *nogoods* ainsi qu'une structure d'ordres nommée conditions de sécurité (*safety conditions*).

Définition 15 (*safety condition*). *Un safety condition est défini comme étant une affirmation de la forme : $x < y$, tels x et y sont des variables.*

Si S est un ensemble de *safety conditions*, on dénote par $\leq_s$ la fermeture transitive de $<$, ce qui signifie que S est acyclique si $\leq_s$ est antisymétrique.

$$x <_s y \implies x \leq_s y, y \not\leq_s x \quad (3.1)$$

Autrement : $x <_s y$, s'il existe une séquence (peut être vide) de *safety conditions* :

$$x < z_1 < z_2 < \dots < z_n < y \quad (3.2)$$

Pour un *nogood* donné γ , on dénote par S_γ tous les *safety conditions* $x < y$, tels que x appartient à l'antécédent de γ et y à sa conclusion.

Une paire $\langle \Gamma, S \rangle$, avec Γ un ensemble de *nogoods* et S un ensemble de *safety conditions*, est acceptable si Γ est acceptable, S est acyclique et $S_\Gamma \subset S$.

Définition 16 (Affaiblissement). *Pour tout safety condition S et toute variable x , on définit l'affaiblissement de S à x , noté par $W(S, x)$, l'ensemble de safety conditions obtenus en supprimant de S tous les safety conditions de la forme $z < y$, tel que $x <_s y$, et en ajoutant $z < y$ pour tout y .*

Pour tout ensemble de *safety conditions* S , toute variable x et tout ordre total $<$ consistant avec le *safety condition* dans $W(S, x)$, il existe un ordre total consistant avec S et approprié pour $<$ à travers x .

Description de l'algorithme

Dans le but de réutiliser la dernière solution ainsi que les techniques de raisonnement passées, on démarre la recherche avec un *DCSP* qui contient une affectation complète, qui est normalement la dernière solution trouvée, tout en gardant les informations concernant les *nogoods* *NgList* et les *safety conditions* *SC*, enregistrées durant l'ancien processus de recherche. Ainsi, on met à jour *cstrs*, la liste des contraintes à satisfaire et on teste la consistance de chaque contrainte (ligne 1). Tant que la solution n'est pas trouvée et qu'aucun *nogood* vide n'est généré, on choisit une contrainte quelconque pour tester sa consistance (lignes 4 et 5). Si le choix de contrainte a échoué (toutes les contraintes sont déjà testées et satisfaites), la solution est trouvée et la recherche est alors terminée (lignes 6, 7, 4 et 15). Si une inconsistance est détectée, un *nogood* est généré en utilisant la procédure *GenerateNg(conflictVars)* (lignes 8 et 10). Par la suite, on ajoute le nouveau *nogood* à *NgList* et on appelle la fonction récursive *Repair(Rhs(ng))* qui cherche une nouvelle affectation à la variable *Rhs(ng)* (lignes 11-12). Si la contrainte c est consistante, elle est déplacée automatiquement depuis la liste des contraintes *cstrs* vers la liste des contraintes déjà testées et satisfaites *freeCstrs* (ligne 13 et 14).

La procédure *Repair(var)*, qui cherche à trouver une affectation à la variable *var*, teste si le domaine de cette variable est vide ou pas (ligne 17). Si elle n'arrive pas à affecter une valeur à *var* (à cause d'un vidage de domaine), elle génère un *nogood ng*, qui est le *nogood* de résolution de tous les *nogoods* où *var* représente la conclusion, en utilisant la fonction *ResolutionNg(var)* (lignes 17-18). Si le *nogood* résultant est vide, cela exprime qu'aucune solution ne peut être trouvée et la recherche est ainsi arrêtée (lignes 19-21), sinon, la

```

Procedure PDB(DCSP,newCstrs)
1.  DCSP.cstrs.upDate(newCstrs);
2.  solutionFound  $\leftarrow$  False;
3.  emptyNgGenerated  $\leftarrow$  False;
4.  while ( $\neg$ solutionFound  $\&\&$   $\neg$ emptyNgGenerated) do
5.     $c \leftarrow$  getACstr(DCSP.cstrs);
6.    if (isNULL(c)) then
7.      solutionFound  $\leftarrow$  true;
8.    if ( $\neg$ isConsistant(c)) then
9.      conflictVars  $\leftarrow$  getListOfVars(c);
10.     ng  $\leftarrow$  GenerateNg(conflictVars);
11.     DCSP.ngList.add(ng);
12.     Repair(Rhs(ng));
13.   else
14.     move(c, DCSP.cstrs, DCSP.freeCstrs);
15. DCSP.ReturnResult();

```

```

Function Repair(var)
17. if (var.Domain.isEmpty()) then
18.   ng2  $\leftarrow$  ResolutionNg(var);
19.   if (ng2.isEmpty()) then
20.     emptyNgGenerated  $\leftarrow$  True;
21.     Return False;
22.   if ( $\neg$ Repair(Rhs(ng2))) then
23.     Return False;
24. DCSP.SC.add(var);
25. DCSP.ngList.removeWithLhs(var);
26. moveWith(var, DCSP.freeCstrs, DCSP.cstrs);
27. var.ChangeValue();
28. Return True;

```

```

Function GenerateNg(listVar)
29. lower = CSP.getLower(listVar);
30. if (lower  $\neq$  NULL) then
31.   Return getNg(listVar, lower);
32. GenerateANg(listVar);

```

```

Function ResolutionNg(var)
33. listVar = CSP.NgList.getAllVarLhs(var);
34. GenerateNg(listVar);

```

FIGURE 3.1 – Description de *PDB*

fonction *Repair(Rhs(ng2))* est appelée récursivement afin de chercher une nouvelle affectation à la variable *Rhs(ng2)*. Si *Rhs(ng2)* ne peut pas changer de valeur à cause d'un vidage de domaine, un *nogood* vide est généré, et la valeur *False* est retournée pour exprimer l'insolvabilité (lignes 22-23). Sinon, *Repair(var)* supprime tous les *nogoods* contenant *var* dans l'antécédent et préserve les ordres qu'ils imposent dans la liste des *safety conditions SC*, ainsi, elle affecte à *var* une nouvelle valeur appartenant à son domaine valide

(lignes 24-27).

La procédure *GenerateNg(listVar)* est utilisée pour choisir, à partir d'une liste de variables, une variable pour appartenir à la conclusion du *nogood*, tout en respectant l'ordre déjà imposé par la liste *SC* ainsi que celle des *nogoods* (lignes 29-31), si aucun ordre n'est imposé entre ces variables, la fonction *GenerateANg(listVar)* est appelée dans le but de choisir une variable qui représentera la conclusion du *nogood*, et qui exprimera alors une extension de l'ordre existant (ligne 32).

ResolutionNg(var) est appelée quand la variable *var* détecte un vidage de domaine. Dans ce cas, la fonction enregistre dans la structure *listVar* toutes les variables qui appartiennent à la partie gauche de tous les *nogoods* qui interdisent les valeurs de *var* (ligne 33). Ainsi, elle fait appel à la procédure *GenerateNg(listVar)* pour générer le *nogood* de résolution (ligne 34).

Chapitre 4

Contribution aux DCSPs

4.1 Extended Partial-order Dynamic Backtracking

4.1.1 Motivation

Le *PDB* Étendu (*EPDB* pour *Extended Partial-order Dynamic Backtracking*) est une approche de réparation pour environnements dynamiques. Elle permet de traiter les *DCSPs* d'une manière efficace, en se basant sur les heuristiques d'ordonnancement, tout en respectant l'ordre imposé par les *nogoods* et les *Safety Conditions*, stockés durant le processus de la recherche.

En fait, malgré sa flexibilité, *PDB* ne manipule pas efficacement l'ordre partiel des variables, ce qui peut conduire à des changements considérables dans le comportement de la recherche. Ainsi, les problèmes qui sont relativement faciles à réparer pour une approche de recherche donnée, risquent d'être transformés en des problèmes plus difficiles.

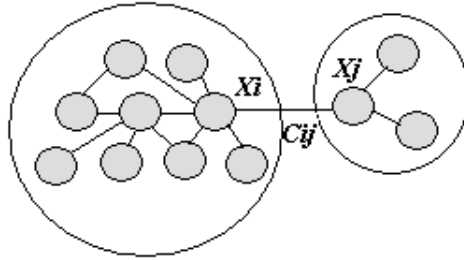


FIGURE 4.1 – réseau de contraintes

Pour schématiser l'inconvénient majeur de *PDB*, prenons comme exemple le réseau de contraintes illustré dans la figure 4.1. Ce réseau comporte deux sous-problèmes différents. L'un est plus complexe et dense que l'autre. Si la contrainte C_{ij} est violée, d'où la génération d'un *nogood*, *PDB* n'a pas de préférences exigées pour l'ordre exprimé par ce *nogood*. Hors, il est plus raisonnable de choisir la variable X_j comme conclusion de ce *nogood*, c-à-d comme variable pour laquelle la valeur va changer. En fait, X_j appartient au sous-problème le plus simple. Ainsi, il est plus facile de réparer la solution en appliquant les changements sur cette partie du réseau. En d'autres termes, si la valeur de X_i change, cela peut conduire à des changements dramatiques dans le processus de la recherche, puisqu'il est plus probable que le changement affecte les voisins de la variable concernée, sachant que ces voisins sont nombreux et ont plusieurs connexions.

Par conséquent on préfère assurer la stabilité des variables appartenant aux sous-problèmes les plus complexes.

Ainsi, l'idée principale de *EPDB* est de mettre à jour la description classique de réparation *DCSP*, en manipulant plus efficacement l'ordre partiel

entre les variables, par l'utilisation des heuristiques d'ordonnancement de variables.

Pour appliquer cette idée, on introduit les heuristiques de choix de variables dans les deux cas suivants :

- lorsqu'un *nogood*, qui justifie la violation d'une contrainte, est généré (fig. 3.1 ligne 10).
- quand un *nogood* de résolution est causé par un vidage de domaine (fig. 3.1 ligne 17).

Une seconde amélioration de *EPDB* consiste à fixer la contrainte à tester jusqu'à sa satisfaction. En fait, une fois une contrainte C_i est testée, satisfaite ou pas, *PDB* passe automatiquement à une autre contrainte C_j (fig. 3.1 ligne 5). C_i sera re-testée après.

4.1.2 EPDB comparé à PDB

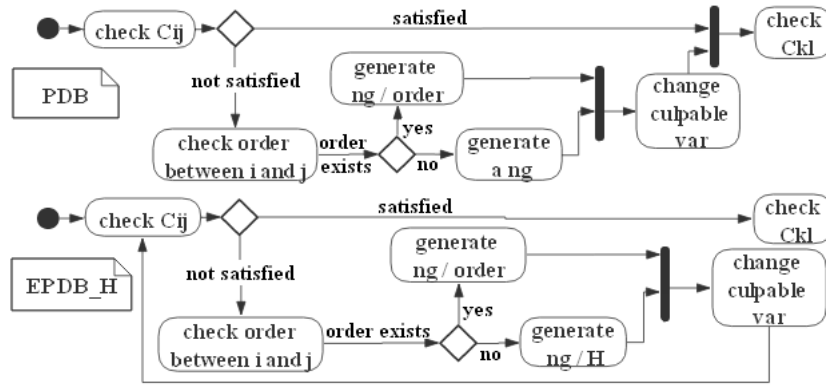


FIGURE 4.2 – Comportement de PDB et EPDB

Comme le montre la figure 4.2, après avoir testé une contrainte C_{ij} , satisfaite ou pas, *PDB* teste une autre contrainte C_{kl} . En fait, si C_{ij} est violée, *PDB* teste si un ordre existe déjà entre les variables concernées i et j . Par la suite, *PDB* génère un *nogood* ng qui respecte l'ordre, s'il existe. Si aucun ordre n'existe entre i et j , *PDB* génère n'importe quel *nogood* ng , qui imposera par la suite un nouvel ordre entre i et j . A ce moment, la conclusion de ng change de valeur et une autre contrainte C_{kl} est testée.

Après le test d'une contrainte C_{ij} , *EPDB* passe à une autre contrainte C_{kl} seulement si C_{ij} est satisfaite. Si ce n'est pas le cas, il teste si un ordre existe déjà entre les variables concernées i et j . Par la suite, *EPDB* génère un *nogood* ng , respectant l'ordre, s'il existe. Si aucun ordre n'existe entre i et j , le *nogood* est généré suivant l'heuristique choisie H . A ce moment là, la conclusion de ng change de valeur et C_{ij} est re-testée.

4.1.3 Exemple d'exécution

L'exemple suivant montre la différence entre *PDB* et *EPDB* utilisant l'heuristique *dom*.

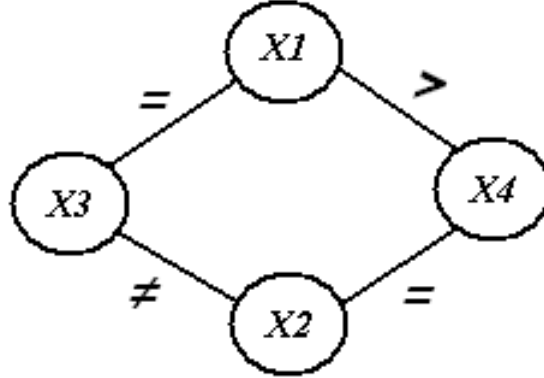


FIGURE 4.3 – Réseau de contraintes, $D_i = \{1, 2\} \forall i \in [1, 4]$

Dans la figure 4.3, le problème original est représenté par une affectation complète. Ce problème comprends les quatre variables $X = \{X_1, X_2, X_3, X_4\}$. tous les domaines contiennent les valeurs $D_i = \{1, 2\}$. Il existe une contrainte d'égalité entre X_1 et X_3 , et entre X_2 et X_4 , une autre contrainte d'inégalité entre X_2 et X_3 , et une contrainte de supériorité entre X_1 et X_4 .

Dans la table 4.1, on assigne la valeur 1 à toutes les variables du réseau puis on lance *PDB* pour réparer l'affectation. Cette affectation initiale viole plus qu'une contrainte. Nous supposons que nous choisissons la contrainte entre X_1 et X_4 . Cette contrainte est violée donc un *nogood* doit être généré. Aucun ordre n'existe entre les deux variables, ainsi, le *nogood* peut être écrit avec soit X_4 ou X_1 dans la conclusion. On suppose que X_4 est sélectionnée comme conclusion du *nogood* (1) et change alors de valeur. Cette affectation viole une variété de contraintes. Nous supposons que nous choisissons celle entre X_2 et X_3 . Nous n'avons toujours pas d'ordre entre X_2 et X_3 , et X_3 est supposée sélectionné comme conclusion du *nogood* (2). La prochaine contrainte vérifiée et violée cette fois est supposée celle entre X_1 et X_3 . Nous avons choisis X_3 comme conclusion du *nogood* (3). Ce *nogood* se résout avec (2) pour produire le *nogood* de résolution (4), puisque X_3 n'a plus de valeur dans son domaine (situation de *dead-end*), X_2 change sa valeur alors (2) est supprimé. Nous ajoutons également (5), la condition de sécurité indiquant que X_2 doit continuer à précéder X_3 . La seule contrainte violée est celle entre X_1 et X_4 . Le *nogood* (1) exige, cette fois ci, que la conclusion du nouveau *nogood* soit X_4 (6). Ce *nogood* se résout avec (1) pour produire (7). Nous supprimons tous les *nogoods* avec X_1 dans la partie gauche, mais nous gardons les ordres (8), (9) et (10). Nous supprimons (5) en raison de l'ajout de

TABLE 4.1 – Execution de *PDB*

X_1	X_2	X_3	X_4	add	remove
1	1	1	1	$X_1 = 1 \rightarrow X_4 \neq 1$ (1)	2
1	1	1	2	$X_2 = 1 \rightarrow X_3 \neq 1$ (2)	
1	1	2	2	$X_1 = 1 \rightarrow X_3 \neq 2$ (3)	
				$X_1 = 1 \rightarrow X_2 \neq 1$ (4)	
				$X_2 < X_3$ (5)	
1	2		2		1,3,4,6
1	2	1	2	$X_1 = 1 \rightarrow X_4 \neq 2$ (6)	
				$X_1 \neq 1$ (7)	
				$X_1 < X_4$ (8)	
				$X_1 < X_3$ (9)	
				$X_1 < X_2$ (10)	5
2	2	1			
2	2	1	1	$X_1 = 2 \rightarrow X_3 \neq 1$ (11)	

TABLE 4.2 – Execution de *EPDB_{dom}*

X_1	X_2	X_3	X_4	add	remove
1	1	1	1	$X_1 = 1 \rightarrow X_4 \neq 1$ (1)	1,2
1	1	1	2	$X_1 = 1 \rightarrow X_4 \neq 2$ (2)	
1	1	1		$X_1 \neq 1$ (3)	
	1	1		$X_1 < X_4$ (4)	
2	1	1	1	$X_1 = 2 \rightarrow X_3 \neq 1$ (5)	
2	1	2	1		

(9), pour la raison de flexibilité. Autrement, nous supprimons les traces des anciens *nogoods* afin que chaque variable n'aie qu'un antécédent dans l'arbre de recherche hiérarchique. Ainsi, comme X_1 est le nouvel antécédent de X_3 , X_2 ne l'est plus, d'où la suppression de (5). À ce stade, nous changeons la valeur de X_3 , puisque la seule contrainte violée est celle entre X_1 et X_3 et que (9) exige que X_1 précède X_3 (11) et la recherche continue.

Dans la table 4.2, *EPDB_{dom}* est exécuté sur le même problème. Toutes les variables sont assignées par la valeur 1. Nous choisissons la même contrainte que dans *PDB*. Le *nogood* est supposé généré comme (1) car aucun ordre n'est imposé entre les deux variables et les domaines des variable ont la même taille. La nouvelle affectation viole une variété de contraintes, mais *EPDB* re-vérifie la contrainte en cours jusqu'à sa consistance. Cette contrainte est toujours violée, mais cette fois ci, (1) impose un ordre entre les deux variables, d'où le *nogood* (2). X_4 n'a plus de valeur dans son domaine (situation de *dead-end*), ainsi, la résolution de (1) et (2) donne le *nogood* de résolution (3). Puisque X_1 va changer de valeur, (1) et (2) sont supprimés en gardant l'ordre qu'ils expriment (4). Supposons que la prochaine contrainte à vérifier

est celle entre X_1 et X_3 . Cette contrainte est violée, et puisque le domaine de X_3 est plus grand que celui de X_1 , X_3 est choisie comme conclusion du nouveau *nogood* (5). On change la valeur de X_3 et on obtient la solution du réseau.

4.1.4 Description de l'algorithme

Dans le but de réutiliser la dernière solution ainsi que les techniques de raisonnement passées, *EPDB* (fig. 4.4) démarre la recherche avec un *DCSP* qui contient une affectation complète, qui est normalement la dernière solution trouvée avant la perturbation, tout en gardant les informations concernant les *nogoods NgList*, enregistrées durant l'ancien processus de recherche. *EPDB* procède comme *PDB* avec les différences majeures suivantes :

- 1^{ère} différence :
 - Dans *PDB* (voir fig. 3.1), tant que la solution n'est pas trouvée et qu'aucun *nogood* vide n'est généré, on passe à la contrainte suivante pour tester sa consistance (lignes 4 et 5).
 - Dans *EPDB* (voir fig. 4.4), on choisit une contrainte, et on la re-teste jusqu'à sa consistance. Autrement dit, on ne choisit une nouvelle contrainte à tester qu'une fois l'ancienne est satisfaite (ligne 13).
- 2^{ème} différence :
 - Dans *PDB* (voir fig. 3.1), si aucun ordre n'est imposé entre les variables en conflit, la fonction *GenerateANg(listVar)* est appelée dans le but de choisir une variable quelconque pour représenter la conclusion du *nogood*. Ce *nogood* est généré quand une contrainte est violée (lignes 10 puis 31), ou quand un vidage de domaine est détecté (lignes 17 puis 33, puis 31).
 - Dans *EPDB* (voir fig. 4.4), si aucun ordre n'est imposé entre les variables en conflit, la fonction *GenerateNgWithHeuristic(listVar, VOH)* est appelée dans le but de choisir, à partir de l'heuristique déterminée *VOH*, la variable qui représentera la conclusion du *nogood*. Ce *nogood* est généré quand une contrainte est violée (lignes 8 puis 32), ou quand un vidage de domaine est détecté (lignes 18 puis 34, puis 32).

4.1.5 Preuve de terminaison

Comme démontré dans [36], l'algorithme *Partial-order Dynamic Backtracking* (*PDB*) termine. En fait, *PDB* respecte les conditions d'ordres relatifs imposés par l'ensemble des *nogoods* d'une part, et par les *Safety*

```

Procedure EPDB(DCSP, newCstrs, VOH)
1.  DCSP.cstrs.update(newCstrs);
2.  solutionFound  $\leftarrow$  False;
3.  emptyNgGenerated  $\leftarrow$  False;
4.   $c \leftarrow$  getACstr(DCSP.cstrs);
5.  while ( $\neg$ solutionFound  $\&\&$   $\neg$ emptyNgGenerated) do
6.    if ( $\neg$ isConsistant(c)) then
7.      conflictVars  $\leftarrow$  getListOfVars(c);
8.      ng  $\leftarrow$  GenerateNg(conflictVars, VOH);
9.      DCSP.ngList.add(ng);
10.     Repair(Rhs(ng), VOH);
11.   else
12.     move(c, DCSP.cstrs, DCSP.freeCstrs);
13.      $c \leftarrow$  getACstr(DCSP.cstrs);
14.     if (isNULL(c)) then
15.       solutionFound  $\leftarrow$  true;
16.   DCSP.ReturnResult();

Function Repair(var, VOH)
17. if (var.Domain.isEmpty()) then
18.   ng2  $\leftarrow$  ResolutionNg(var, VOH);
19.   if (ng2.isEmpty()) then
20.     emptyNgGenerated  $\leftarrow$  True;
21.     Return False;
22.   if ( $\neg$ Repair(Rhs(ng2), VOH)) then
23.     Return False;
24.   DCSP.SC.add(var);
25.   DCSP.ngList.removeWithLhs(var);
26.   moveWith(var, DCSP.freeCstrs, DCSP.cstrs);
27.   var.ChangeValue();
28.   Return True;

Function GenerateNg(listVar, VOH)
29. lower = CSP.getLower(listVar);
30. if (lower  $\neq$  NULL) then
31.   Return getNg(listVar, lower);
32. GenerateNgWithHeuristic(listVar, VOH);

Function ResolutionNg(var, VOH)
33. listVar = CSP.NgList.getAllVarLhs(var);
34. GenerateNg(listVar, VOH);

```

FIGURE 4.4 – Description de l'approche *EPDB*

Conditions qui enregistrent l'ordre qu'imposaient les anciens *nogoods* supprimés d'autre part. Lors d'un échec d'instanciation, l'ordre choisi pour le nouveau *nogood* qui justifie cet échec doit obligatoirement respecter celui imposé par les *safety conditions* ainsi que les *nogoods* existants. Par conséquent, la structure résultante de ces ensembles d'ordres ne peut pas être

cyclique.

EPDB procède de la même manière que *PDB*, avec une différence majeure qui consiste à utiliser une heuristique d'ordonnancement pour générer les nouveaux *nogoods*. Ces heuristiques sont utilisées uniquement dans le cas où aucun ordre n'est imposé entre les variables concernées par ce *nogood*. Ainsi, *EPDB* termine.

4.1.6 Étude expérimentale

Nous avons effectué un ensemble de tests expérimentaux afin de montrer comment l'intégration des heuristiques de choix peut affecter la performance de l'algorithme *PDB*. Nous évaluons les résultats en termes de tests de contraintes (*CC*) et de temps de calcul (*time*). Toutes les expérimentations ont été effectuées sur la plate-forme *Java*. Les heuristiques utilisées en *EPDB* sont : degré (*deg*), taille de domaine (*dom*), degré dynamique (*ddeg*), degré pondéré (*wdeg*), nombre de conflits (*conf*), *dom/deg*, *dom/ddeg* et *dom/wdeg*. Nous évaluons la performance de *EPDB* sur deux problèmes différents : *DCSPs* binaires aléatoires uniformes et problèmes de planification de réunions *MSPs*.

Expérimentations pour les *DCSPs* binaires aléatoires uniformes

Un *DCSP* binaire aléatoire uniforme est une séquence de *CSPs* statiques : $P_0, P_1, \dots, P_{\alpha-1}, P_\alpha$. Le problème P_0 est le *CSP* original qui est sujet à des perturbations, et pour chaque P_i , de nouvelles contraintes sont ajoutées à P_0 . Pour des raisons logiques, dans ces expérimentations, nous supposons que le taux maximal de contraintes ajoutées (c-à-d α/m où α est le nombre maximal de nouvelles contraintes ajoutées et m est le nombre total de contraintes dans le problème final) est équivalent à $\sim 20\%$.

Les *CSP* sont caractérisés par $\langle n, d, p1, p2 \rangle$, où n est le nombre de variables, d le nombre de valeurs par variable, $p1$ la connectivité réseau et $p2$ la dureté des contraintes. La densité $p1$ du réseau de contraintes est défini comme étant le ratio du nombre de contraintes par rapport au nombre maximal de contraintes possibles dans ce réseau. La dureté $p2$ d'une contrainte correspond soit à la proportion de tuples non autorisés, soit à la probabilité que le tuple ne soit pas autorisé.

Des tests ont été effectués sur des problèmes à densité faible et élevée, respectivement $\langle 20, 10, 0.25, 0.6 \rangle$ et $\langle 20, 10, 0.75, 0.25 \rangle$. La dureté des contraintes est choisie de telle sorte que les problèmes se situent dans la transition de phase[23], qui est la région où les problèmes sont les plus difficiles. Pour chaque paire $\langle p1, p2 \rangle$, 20 différentes instances sont traitées par chaque algorithme et les résultats actuels sont les moyennes de ces 20 exécutions. Le taux de contraintes injectées $\%IC = \frac{|nouvelles\ contraintes|}{|contraintes\ totales|}$. Cela indique le taux de perturbation qui sera traité en utilisant à la fois *PDB* et

EPDB. *IC* varie de 1% à 20%, tel que pour chaque perturbation donnée de $m\%$, les contraintes ajoutées sont les mêmes que celles de la perturbation de $m - 1\%$, plus 1% de contraintes supplémentaires. Les 30 problèmes finaux (après la perturbation) sont les mêmes pour chaque taux de perturbation.

La figure 4.5 présente l'effort de calcul en termes de tests de contraintes

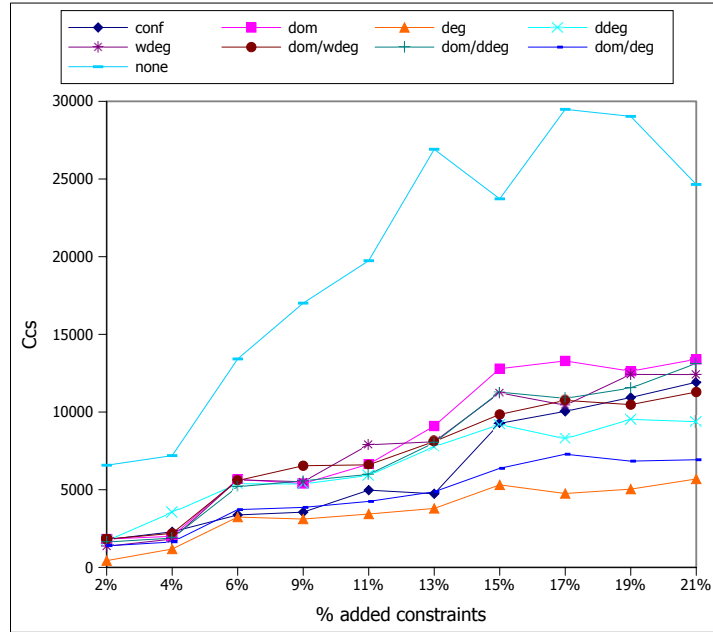
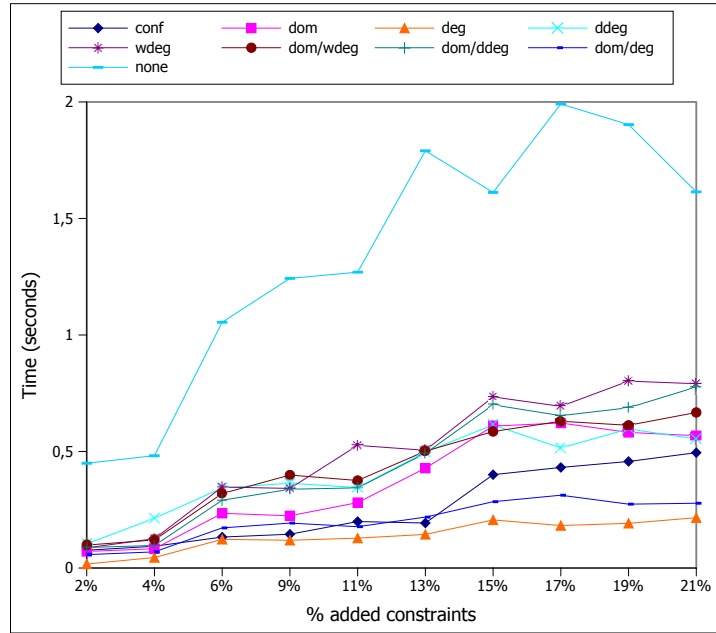
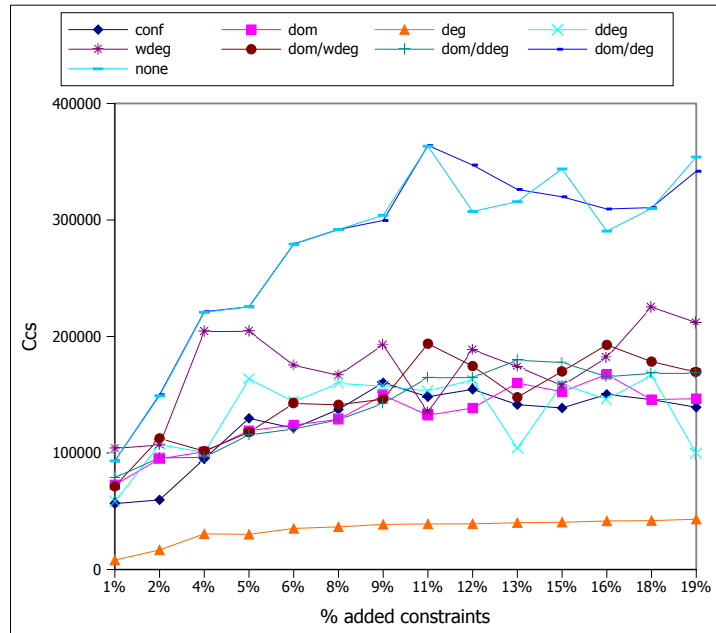


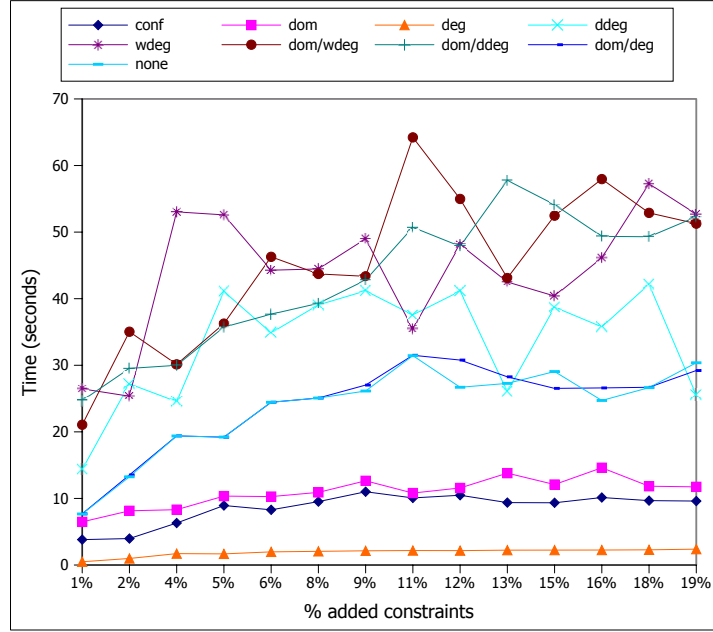
FIGURE 4.5 – Nombre de contraintes testées ($p1 = 0.25$, $p2 = 0.6$)

nécessaires pour réparer les solutions pour des problèmes avec $p1 = 0.25$ et $p2 = 0.6$. *EPDB* se comporte mieux que *PDB* et la plus grande amélioration est observée lors de l'utilisation de l'heuristique *deg*. La figure 4.6 montre le temps de calcul effectué par chaque algorithme pour les mêmes problèmes. *EPDB* combiné avec l'heuristique *deg* nécessite moins de temps pour réparer les solutions. Les figures 4.7 et 4.8 présentent des résultats similaires, en termes de tests de contraintes et de temps de calcul, pour les *DCSP* à densité élevée avec $p1 = 0.75$ et $p2 = 0.25$. Généralement, l'utilisation de l'heuristique *deg* améliore la réparation, surtout en terme de tests de contraintes, indépendamment de la densité du réseau. Pour certaines heuristiques, les performances en terme de temps de calcul sont légèrement différentes par rapport à leurs performances dans des réseaux légers, et leur comportement est pire que celui de *PDB* en raison de leur aspect dynamique.

FIGURE 4.6 – Temps d'exécution ($p_1 = 0.25$, $p_2 = 0.6$)FIGURE 4.7 – Nombre de contraintes testées ($p_1 = 0.75$, $p_2 = 0.25$)

Expérimentations concernant le problème de planification de réunions dynamiques *DMSP*

Ce problème est une séquence de *MSPs* statiques : $P_0, P_1, \dots, P_{\alpha-1}, P_{\alpha}$, tel que le réseau est soumis à des perturbations de contraintes, et chaque P_i

FIGURE 4.8 – Temps d'exécution ($p1 = 0.75$, $p2 = 0.25$)

diffère de celui précédent par l'ajout de quelques contraintes.

Un *MSP* est caractérisé par $\langle k, p, n, d, h, t, a \rangle$, tel que k est le nombre de réunions, p est le nombre de participants, n est le nombre de réunions par participant et d est le nombre de jours. Des intervalles de temps différents sont disponibles pour chaque réunion et h est le nombre d'heures par jour, t est la durée de la réunion et a est le pourcentage de disponibilité pour chaque participant. Dans le modèle utilisé, $p \times n$ est le nombre de variables. Les variables X_{ij} , telles que $i \in p$, $j \in n$ et $X_{ij} \in [timeSlot(1), timeSlot(m)]$, expriment la totalité des réunions par tous les participants. Ainsi, deux types de contraintes existent :

- Comme p est le nombre de participants et n le nombre de réunions par participant, pour chaque deux variables distinctes X_{ij} et X_{kl} , si $i = k$, alors les variables sont connectées par une contrainte appelée *arrival-time*, ce qui signifie que le participant i doit assister à deux réunions distinctes. Par conséquent, $|X_{i,j} - X_{k,l}| \geq \delta$, où δ est le temps nécessaire pour passer de $X_{i,j}$ et $X_{k,l}$, exprimé en créneaux horaires.
- Pour chaque deux participants distincts i et k , $X_{i,j} = X_{k,l}$ signifie que i et k doivent assister à la même réunion.

Nous présentons nos résultats pour la classe $\langle 20, 15, 6, 5, 10, 1, 80 \rangle$ et nous modifions le taux de contraintes d'arrivée ajoutées de $\sim 1\%$ à $\sim 20\%$. Nous avons généré 25 instances différentes réparées par chaque algorithme et les résultats sont une moyenne de ces 25 exécutions. Comme le montrent les

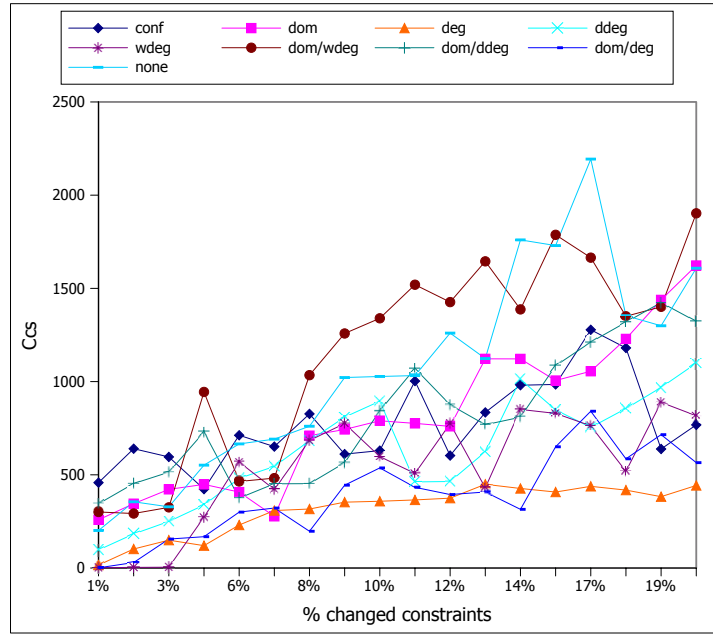


FIGURE 4.9 – Nombre de contraintes testées

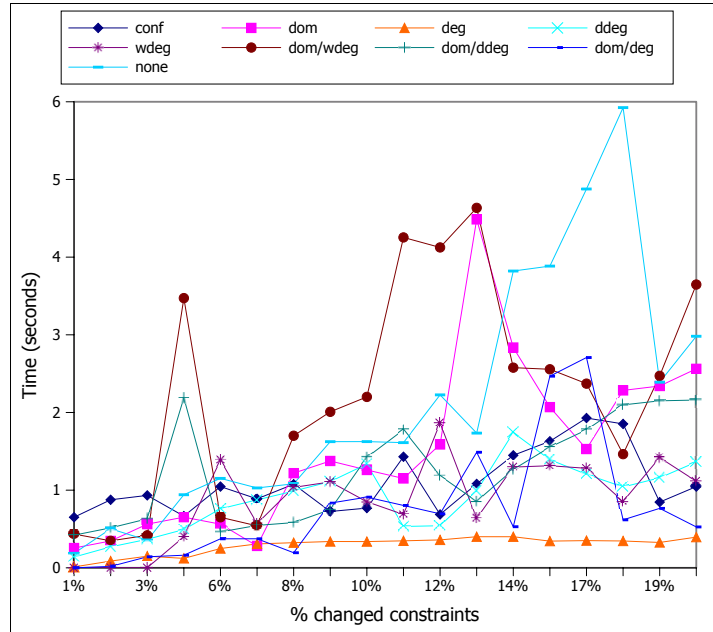


FIGURE 4.10 – Temps d'exécution

figures 4.9 and 4.10, toutes les heuristiques montrent une bonne performance, comparant à l'utilisation du *PDB* standard, à la fois en terme de tests de contraintes et de temps de calcul. La plus grande amélioration est encore

observée lors de l'utilisation de l'heuristique *deg*.

4.1.7 Discussion

Généralement, l'utilisation des heuristiques dans la réparation des environnements dynamiquement modifiés améliore la recherche. Ces heuristiques fonctionnent comme suit : dans une situation de conflit ou de vidage de domaine, un *nogood* est construit de telle sorte que les variables dans la partie gauche, ayant le plus grand ordre de priorité, appartiennent aux sous-problèmes les plus difficiles. Cela peut assurer, autant que possible, que ces variables restent inchangés, et éviter ainsi d'autres situations de conflit lorsque les affectations de ces variables sont modifiées.

L'heuristique la plus pertinente est *deg*, cette heuristique évalue la difficulté des sous problèmes par le nombre de connexions qu'ils partagent avec les autres parties du réseau. Ainsi, lors de la génération d'un *nogood*, les variables ayant le plus grands nombre de voisins sont ceux qui y représentent la partie gauche.

L'heuristique *dom* s'avère plus utile dans les problèmes denses. Ces problèmes sont des *DCSPs* aléatoires uniformes, ainsi, les domaines initiaux des variables sont de tailles égales. Étant donné que ces problèmes se situent dans la transition de phase, la densité élevée permet des éliminations élevées de valeurs, et ainsi des différences plus remarquables entre les tailles des domaines, d'où l'utilité de l'heuristique *dom*.

Dans la réparation, l'heuristique *wdeg* n'est pas assez utile comme elle l'est dans la résolution. Notons que, pour une variable donnée X , le poids au moment t_α est la somme des incréments de poids à chaque moment $t_{\alpha-i}$. Ces incréments se produisent lorsque la variable X est en conflit avec les variables Y qui sont en ordre de priorité inférieure. Cependant, lorsque les changements dans le réseau de contraintes sont importants, il est probable que le réseau a été soumis à plusieurs changements entre $t_{\alpha-i}$ et t_α (c-à-d que les variables Y au moment t_α ont des affectations différentes à celles à l'instant $t_{\alpha-i}$). Par conséquent, *wdeg* ne reflète pas les états actuels du réseau, mais des états passés qui n'existent plus.

Dans les graphes denses, puisque le nombre de *nogoods* devient intéressant, l'heuristique *conf* donne de bonnes performances. Pour assurer une convergence vers une solution dès que possible, les variables subissent à un plus grand nombre de conflits, ce qui implique la réduction de leur taille de domaine, restent inchangées.

L'objectif de *ddeg* est de maintenir la stabilité de l'état des sous-problèmes qui ont plus de contraintes avec les sous-problèmes futurs. Même si cette heuristique améliore la recherche, elle n'est pas suffisante car elle ne tient pas compte des variables de priorité supérieure qui partagent des contraintes avec la variable actuelle, sachant que la notion d'ordre d'affectation des variables est absente dans la réparation. Ainsi, un voisin de priorité supérieure d'une

variable X peut subir à des changements, exactement comme les voisins de priorité inférieure de X , sans que X change de valeur.

4.2 Dynamic Constraint Ordering

4.2.1 Introduction

Plusieurs études et recherches ont développé de nombreuses techniques pour augmenter l'efficacité des algorithmes de satisfaction de contraintes dynamiques (*DCSPs*). Certaines de ces techniques sont basées sur une séquence de décisions à prendre, lors de la résolution d'un algorithme de recherche *CSP*, comme la prochaine variable à assigner et la valeur à choisir en première. Ces décisions sont appelées les heuristiques de sélection de variables et de valeurs. Il a été montré que ces heuristiques peuvent avoir un impact important sur la performance de nombreux problèmes [62]. Cependant, les heuristiques ordonnant les contraintes ont reçu moins d'attention que celles concernant les variables et les valeurs. Dans la littérature, seulement certaines contributions ont été faites et principalement comme une phase de pré-traitement avant de commencer le processus de recherche. Dans ce chapitre, nous allons montrer que les heuristiques de choix de contraintes sont prometteuses et capables de faire des choix pertinents pendant le processus de réparation des *CPSs* dynamiques.

Le problème considéré ici est de trouver une nouvelle solution d'un réseau de contraintes dynamique après chaque modification du problème initial. Le présent travail se concentre sur les scénarios où après qu'un *CSP* est résolu, certaines contraintes ont été changées ou injectées et la solution exige une révision pour considérer toutes ces perturbations. Selon [71], nous considérons que tous les changements possibles concernant un *DCSP* peuvent être exprimés en termes d'ajout ou de suppression de contraintes.

Notre approche proposée, appelée *Dynamic Constraint Ordering (DCO)*, choisit dynamiquement les contraintes qui sont les plus probables à causer des conflits. Ainsi, nous nous appliquons le principe du *fail-first*, pour qu'un événement causant un *backtrack* puisse être exécuté le plus tôt possible. Cette méthode peut guider un algorithme de réparation dynamique vers les sous-problèmes de contraintes les plus durs et aborder les inconsistances pour corriger la solution. Les expérimentations ont confirmé que l'approche proposée est intéressante. Nos résultats montrent clairement l'efficacité des heuristiques combinées avec l'algorithme *PDB* pour le problème de planification de réunions dynamiques *DMSPs*. Les heuristiques ont été testées en termes du temps d'exécution, du nombre de contraintes évaluées et de la stabilité de la solution.

4.2.2 Travaux effectués

Dans la littérature, plusieurs algorithmes ont été développés pour traiter les problèmes de satisfaction de contraintes dynamiques. En particulier, dans [71] une description d'un algorithme complet (*Local Changes*) dédiée à

la maintenance des solutions dans les problèmes dynamiques a été proposée. Ensuite, des extensions différentes ont été proposées pour l'algorithme *LC* pour la gestion des contraintes floues (opérateur d'agrégation maximale), entraînant l'algorithme *FLC* (Flexible Local Changes) [53]. D'autres auteurs ont également essayé de résoudre le même problème en utilisant des approches différentes. Plus précisément, dans [21], ils ont présenté un outil qui est dédié à la planification continue avec l'entrelacement entre la planification et l'exécution et qui utilise une approche de réparation itérative. Le premier reproche de ces contributions est que les auteurs ne sont pas intéressés à étudier la *scalabilité* des algorithmes proposés par rapport aux changements progressifs des problèmes résolus. le second reproche est qu'aucun des auteurs ne s'est intéressé à l'étude et l'utilisation de connaissances heuristiques d'un tel réseau de contraintes dynamique.

D'autre part, les expérimentations et les analyses conduites par plusieurs chercheurs ont montré que l'ordre dans lequel les variables et les valeurs sont choisies pendant la recherche peut avoir un impact considérable sur la complexité de l'espace de recherche exploré. Cependant, nous notons que les heuristiques de choix de contraintes ont reçu moins d'attention par les chercheurs dans la littérature.

Concernant le travail relatif aux heuristiques de choix de contraintes, nous pouvons citer [74], où les auteurs ont montré que les heuristiques d'ordonnement peuvent augmenter l'efficacité de la relaxation en réduisant le nombre de tests de paires de valeurs. Les heuristiques proposées ordonnent la liste des paires de variables (contraintes) de sorte que l'effet majeur consiste à réduire la liste des ajouts. Les auteurs ont conclu que l'ordre par la taille croissante du domaine *relaxed against* est une stratégie générale efficace pour améliorer la performance des algorithmes qui établissent la consistance d'arc complète. Cependant, dans [34], les auteurs ont proposé une nouvelle heuristique de choix de contraintes dans *AC3*, où l'ensemble des choix est composé des arcs dans l'ensemble actuel maintenu par *AC3*. Ils ont utilisé la notion de *constrainedness* (degré de contrainte) comme heuristique pour réduire le nombre de contrôles nécessaires pour établir la consistance d'arc en *AC3*. D'autre part, dans [64] et [63], les auteurs ont présenté une heuristique de pré-traitement appelée *Constraint Ordering Heuristics (COH)* qui étudie la *constrainedness* du problème d'ordonnement et classe les contraintes de sorte que les plus dures soient étudiées d'abord. Ainsi, un algorithme de *backtrack* peut aborder la partie la plus dure d'un problème de recherche et les inconsistances peuvent être trouvées plus tôt.

Comme nous l'avons déjà mentionné, l'ordre de contraintes a reçu moins d'attention que celui concernant celui de variables et de valeurs. Les techniques principalement étudiées sont utilisées comme une approche de pré-traitement pour réduire la redondance dans les réseaux de contraintes ou dans la propagation de contrainte pour réaliser la consistance d'arc. Dans ce qui suit, nous proposons l'utilisation des heuristiques de choix de contraintes au fur

et à mesure de la réparation de *DCSPs*.

4.2.3 Dynamic Constraint Ordering

Un algorithme de recherche concernant la satisfaction de contraintes exige un ordre dans lequel les variables doivent être considérées aussi bien que l'ordre dans lequel les valeurs sont assignées aux variables. Les expérimentations et les analyses de plusieurs chercheurs ont montré que l'ordre dans lequel les variables sont assignées peut avoir un impact substantiel sur la complexité d'un algorithme de recherche. Une fois que la décision est prise pour assigner une variable, plusieurs valeurs peuvent être disponibles. De nouveau, l'ordre dans lequel ces valeurs sont considérées peut avoir un impact considérable sur la performance de trouver la première solution. Cependant, les stratégies ordonnant les contraintes ont reçu moins d'attention que celles ordonnant les variables et les valeurs, ainsi, l'ordre de contraintes n'est pas considéré important, surtout que les *CSPs* statiques n'offrent pas une flexibilité concernant cet ordre, mais c'est que la satisfaction de la conjonction de contraintes qui est intéressante.

Le but de ce chapitre est de proposer une approche heuristique de réparation, stable et robuste, qui permet aux anciennes solutions de *DCSPs* d'être corrigées facilement et étroitement en utilisant l'algorithme *PDB*. Ainsi, un ensemble de contraintes violées, ordonné d'une manière aléatoire, doit être satisfait. Dans cette situation, le choix d'un bon ordre de contraintes peut produire une différence significative dans l'effort global nécessaire pour réparer les solutions. Par conséquent, nous proposons des heuristiques à base de contraintes dynamiques pour réparer les solutions des *DCSPs*, appelé *Dynamic Constraint Ordering DCO*. Ces heuristiques aident *PDB* à prendre une décision moins coûteuse et à choisir la contrainte la plus pertinente à satisfaire en priorité.

Nous notons que *DCO* est adapté essentiellement aux *CSPs* dynamiques, puisque pour les *CSPs* statiques, la recherche se fait en sélectionnant une variable x_i comme variable courante et en lui affectant une valeur. Cependant, avec l'algorithme *PDB* concernant les *CSPs* dynamiques, l'état courant comprend la liste de nouvelles contraintes injectées ou perturbées qui doivent être sélectionnées, testées et satisfaites en vérifiant la consistance de chaque contrainte c_i . Nos heuristiques proposées réordonnent dynamiquement les contraintes qui doivent être satisfaites en priorité. Ainsi, l'ordre dans lequel les contraintes sont prises peut avoir un impact intéressant sur l'effort de recherche.

Définition 17 (DCO). *Soit $(c_1, \dots, c_k)$ une liste de contraintes incohérentes (CL) à satisfaire. A chaque fois que nous devons vérifier la consistance d'une contrainte, nous classifions CL en utilisant une fonction d'ordonnement $ReorderCL$, par rapport à une des heuristiques proposées H.*

Ainsi : $(c_{ord1} < \dots < c_{ordk}) = ReorderCL((c_1, \dots, c_k), H)$ est la liste ordonnée de contraintes.

on désigne par $<$ l'ordre entre les contraintes de CL en ordre croissant ou décroissant.

Comme nous l'avons déjà dit, une contrainte c exprime une relation définissant les combinaisons de valeurs interdites/autorisées pour les variables soumises à la portée de la contrainte. Dans ce chapitre, nous nous concentrons sur les contraintes binaires portant sur deux variables exactement.

Avant de définir les différentes heuristiques de choix de contrainte, nous rappelons d'abord les définitions suivantes :

- $tightness(C_{ij})$: la proportion de tuples interdits par la contrainte C_{ij} . C'est la dureté $p2$ de la contrainte.
- $weight(C_{ij})$: La pondération de la contrainte. C'est le nombre de fois où la propagation de la contrainte C_{ij} conduit à l'échec (conflit).
- $dom(x_i)$: le domaine actuel. C'est le nombre de valeurs restant dans le domaine de x_i .
- $deg(x_i)$: le degré. C'est le nombre de contraintes liant la variable x_i avec d'autres variables, ce qui est égale au nombre de variables liées à x_i .
- $ddeg(x_i)$: le degré dynamique. C'est le nombre de variables, non instanciées, impliquées dans des contraintes avec x_i .

L'ordre des contraintes dans une recherche dynamique est utilisé pour choisir la contrainte à vérifier à chaque pas de la recherche. Cette heuristique devrait être conçue en se basant sur la topologie du graphe de contraintes et le poids des contraintes pour minimiser la longueur ou la profondeur de chaque branche.

Les heuristiques de choix de contraintes proposées sont les suivantes :

- $Ctight(C_{ij}) = max(tightness(C_{ij}))$, sélectionne la contrainte avec la plus grande dureté.
- $Cweight(C_{ij}) = max(weight(C_{ij}))$, choisit la contrainte avec le poids maximum.
- $Cdom(C_{ij}) = min(dom(x_i), dom(x_j))$, sélectionne la contrainte où l'une des variables a un minimum de valeurs dans son domaine.
- $Cdeg(C_{ij}) = max(deg(x_i), deg(x_j))$, choisit la contrainte où l'une des variables possède le degré maximal.
- $Cddeg(C_{ij}) = max(ddeg(x_i), ddeg(x_j))$, choisit la contrainte où l'une des variables a le degré dynamique maximal.

Description

La figure 4.11 présente un pseudo-code de *PDB* utilisant *DCO*. Avant de commencer la recherche, nous mettons à jour l'ensemble des contraintes qui doivent être satisfaites et nous faisons appel à la fonction $ReorderCL(CL, H)$

pour réordonner la liste de contrainte CL en utilisant l'heuristique choisie (lignes 1-3), puis nous vérifions la consistance de chaque contrainte (lignes 7-10).

PDB utilisant DCO procède de la même manière que PDB , en considérant comme liste de contraintes la liste CL ordonnée dynamiquement. Ainsi, chaque fois qu'une contrainte est ajoutée à la liste de contraintes à tester CL (ligne 27), ce qui est précédé par un changement dans les *nogoods*, ce qui peut affecter les valeurs de l'heuristique H choisie, on réordonne l'ensemble de contraintes CL en utilisant cette heuristique (ligne 28). À l'aide de l'exemple

```

Procedure  $PDB\_DCO(CSP, newCstrs, H)$ 
1.  $DCSP.cstrs.update(newCstrs)$ ;
2.  $CL \leftarrow DCSP.cstrs$ ;
3.  $solutionFound \leftarrow False$ ;
4.  $emptyNgGenerated \leftarrow False$ ;
5.  $ReorderCL(C, H)$ ;
6. while ( $\neg solutionFound \ \&\& \ \neg emptyNgGenerated$ ) do
7.    $c \leftarrow get1^{st}Cstr(CL)$ ;
8.   if ( $isNULL(c)$ ) then
9.      $solutionFound \leftarrow true$ ;
10.  if ( $\neg isConsistant(c)$ ) then
11.     $conflictVars \leftarrow getListOfVars(c)$ ;
12.     $ng \leftarrow GenerateNg(conflictVars)$ ;
13.     $DCSP.ngList.add(ng)$ ;
14.     $Repair(Rhs(ng), CL, H)$ ;
15.  else
16.     $move(c, CL, DCSP.freeCstrs)$ ;
17.  $DCSP.ReturnResult()$ ;

Function  $Repair(var, CL, H)$ 
18. if ( $var.Domain.isEmpty()$ ) then
19.    $ng2 \leftarrow ResolutionNg(var)$ ;
20.   if ( $ng2.isEmpty()$ ) then
21.      $emptyNgGenerated \leftarrow True$ ;
22.     Return  $False$ ;
23.   if ( $\neg Repair(Rhs(ng2), CL, H)$ ) then
24.     Return  $False$ ;
25.  $DCSP.SC.add(var)$ ;
26.  $DCSP.ngList.removeWithLhs(var)$ ;
27.  $moveWith(var, DCSP.freeCstrs, CL)$ ;
28.  $ReorderCL(CL, H)$ ;
29.  $var.ChangeValue()$ ;
30. Return  $True$ ;

```

FIGURE 4.11 – Description de l'algorithme PDB_DCO

suiuant, nous montrons la différence entre PDB et PDB_DCO . Supposons que le CSP original (figure 4.12 (a)) comprend :

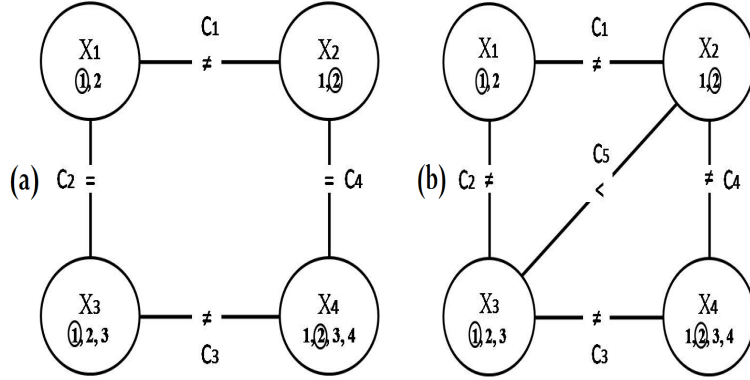


FIGURE 4.12 – (a) Problème original (b) problème perturbé

- Quatre variables : $X = \{X_1, X_2, X_3, X_4\}$
- le Domaine de chaque variable est : $D_1 = D_2 = [1,2]$, $D_3 = [1,2,3]$, $D_4 = [1,2,3,4]$
- Contraintes d'inégalité : $C_1 : X_1 \neq X_2$ et $C_3 : X_3 \neq X_4$
- Contraintes d'égalité : $C_2 : X_1 = X_3$ et $C_4 : X_2 = X_4$.

En utilisant n'importe quel algorithme de recherche, la solution de ce problème est :

$$X_1 = 1, X_2 = 2, X_3 = 1, X_4 = 2$$

Supposons qu'après avoir trouvé cette solution, un changement inattendu (fig. 4.12(b)) affecte le réseau en raison de l'ajout d'une contrainte d'infériorité $C_5 : X_3 < X_2$ et la modification de C_2 et C_4 (inégalité). Dans ce cas, la première solution ne devient plus valable. Dans le tableau 4.3, *PDB* est exécuté. L'affectation initiale viole trois contraintes C_2 , C_4 et C_5 . Nous supposons que la liste de contraintes CL à vérifier est : $[C_2, C_4, C_5]$ (ordre lexicographique). La première contrainte choisie de la liste est C_2 . Cette contrainte est violée, donc un *nogood* est généré et X_3 est sélectionné pour changer sa valeur courante et choisir une autre valeur de son domaine actuel. Cette nouvelle affectation pourrait violer les contraintes liées à X_3 , ainsi, nous ajoutons toutes ces contraintes (C_3 et C_5) à CL pour les vérifier. Après que X_3 choisisse sa valeur, la contrainte C_2 est vérifiée à nouveau. Cette fois, C_2 est consistante et elle est retirée de CL . La prochaine contrainte à vérifier est C_4 qui est violée, alors X_4 modifie sa valeur et sa contrainte liée (C_3) est ajoutée à CL . Puisque la nouvelle valeur de X_4 est compatible, nous vérifions la contrainte suivante C_5 . Comme cette contrainte est inconsistante, X_3 est choisie pour changer sa valeur courante et C_2 et C_3 sont ajoutées à CL . Après que X_3 choisisse sa nouvelle valeur, la contrainte C_5 est retirée de CL puisqu'elle est consistante. La liste CL courante à vérifier est : $[C_3, C_2]$. Les

CL	choix	changement	ajout	suppression
$[C_2, C_4, C_5]$	C_2	$X_1 = 1 \rightarrow X_3 \neq 1$ nvVal : $X_3 = 2$	C_3, C_5	
$[C_2, C_4, C_5, C_3]$	C_2	<i>No change</i>		C_2
$[C_4, C_5, C_3]$	C_4	$X_2 = 2 \rightarrow X_4 \neq 2$ nvVal : $X_4 = 1$	C_3	
$[C_4, C_5, C_3]$	C_4	<i>No change</i>		C_4
$[C_5, C_3]$	C_5	$X_2 = 2 \rightarrow X_3 \neq 2$ nvVal : $X_3 = 3$	C_2, C_3	
$[C_5, C_3, C_2]$	C_5	<i>No change</i>		C_5
$[C_3, C_2]$	C_3	<i>No change</i>		C_3
$[C_2]$	C_2	<i>No change</i>		C_2
$[]$				

TABLE 4.3 – Exécution de PDB

CL	choix	changement	ajout	suppression
$[C_5, C_2, C_4]$	C_5	$X_2 = 2 \rightarrow X_3 \neq 1$ nvVal : $X_3 = 2$	C_2, C_3	
$[C_5, C_2, C_4, C_3]$	C_5	$X_2 = 2 \rightarrow X_3 \neq 2$ nvVal : $X_3 = 3$		
$[C_5, C_2, C_4, C_3]$	C_5	<i>No change</i>		C_5
$[C_2, C_4, C_3]$	C_2	<i>No change</i>		C_2
$[C_4, C_3]$	C_4	$X_2 = 2 \rightarrow X_4 \neq 2$ nvVal : $X_4 = 1$	C_3	
$[C_4, C_3]$	C_4	<i>No change</i>		C_4
$[C_3]$	C_3	<i>No change</i>		C_3
$[]$				

TABLE 4.4 – Exécution de PDB_DCO

contraintes restantes dans CL sont satisfaites, alors la solution est trouvée et la recherche est terminée. Dans le tableau 4.4, PDB_DCO est exécuté sur le même problème et l'heuristique choisie est *Ctight*.

les tuples interdits et la dureté de chaque contrainte sont :

- $C_1 = \{(1,1)(2,2)\} \Rightarrow Ctight(C_1) = 2/4 = 0.50$
- $C_2 = \{(1,1)(2,2)\} \Rightarrow Ctight(C_2) = 2/6 = 0.33$
- $C_3 = \{(1,1)(2,2)(3,3)\} \Rightarrow Ctight(C_3) = 3/12 = 0.25$
- $C_4 = \{(1,1)(2,2)\} \Rightarrow Ctight(C_4) = 2/8 = 0.25$
- $C_5 = \{(1,1)(2,1)(2,2)\} \Rightarrow Ctight(C_5) = 3/6 = 0.50$

Nous commençons la recherche par l'ordonnancement de la liste de contraintes CL en utilisant l'heuristique *Ctigh*. Cette heuristique ordonne les contraintes pour que la contrainte la plus dure soit étudiée en premier. L'ordre des contraintes dans CL sera $[C_5, C_2, C_4]$ et la première contrainte à vérifier est

C_5 . Cette contrainte est violée, donc un *nogood* est généré, X_3 est sélectionnée pour modifier sa valeur courante et les contraintes liées à X_3 (C_2 et C_3) sont ajoutées à CL , puis CL est réordonnée. La liste CL actuelle est : $[C_5, C_2, C_4, C_3]$. La contrainte C_5 est vérifiée à nouveau et elle est toujours violée, alors X_3 choisit une autre valeur de son domaine actuel. Après que X_3 choisisse une nouvelle valeur, la contrainte C_5 est vérifiée à nouveau et cette fois, elle est consistante et est ainsi retirée de CL . La contrainte choisie suivante est C_2 et puisqu'elle est consistante, elle est enlevée de CL et on passe à la contrainte C_4 . Cette contrainte est violée, X_4 est choisie pour modifier sa valeur courante et C_3 est ajoutée à CL . La liste CL après ordonnancement est : $[C_4, C_3]$. Les contraintes restantes dans CL sont satisfaites, alors la solution est trouvée et la recherche est terminée.

Nous remarquons que *PDB* a vérifié plus de contraintes que *PDB_DCO*. Il est vrai que la différence n'est pas grande parce que le problème est petit, mais pour de grands problèmes, la différence sera plus claire, ce que nous verrons dans l'étude expérimentale.

4.2.4 Résultats expérimentaux

Nous avons effectué une série de tests expérimentaux sur des instances aléatoires du problème de planification de réunions dynamiques *DMSP* afin de montrer comment *DCO* peut affecter les performances de l'algorithme *PDB*. Nous évaluons les résultats en termes de nombre de contraintes testées (*CCs*), de temps de calcul (*Time*) et de stabilité de la solution (*Stability*). Nous présentons nos résultats pour la classe $< 20, 5, 15, 5, 10, 1, 90 >$ et nous varions le taux de contraintes injectées (*%IC*) de $\sim 2\%$ à $\sim 24\%$. Rappelons que $\%IC = \frac{|\text{nouvelles contraintes}|}{|\text{contraintes totales}|}$, qui indique le taux de perturbation qui sera traité en utilisant à la fois *PDB* et *PDB_DCO*. Pour chaque perturbation donnée de $m\%$, les contraintes ajoutées sont les mêmes que celles de la perturbation de $m - 1\%$, plus 1% de contraintes supplémentaires. Nous avons généré 25 différentes instances, résolues par chaque algorithme (à savoir *PDB* et *PDB_DCO*) et les résultats sont une moyenne de ces 25 instances. Toutes les expérimentations ont été exécutées sur la plate-forme *Java*.

La figure 4.13 présente l'effort de calcul en termes de nombre de contraintes testées nécessaires à la réparation de solutions. Puisque la sélection de la contrainte à vérifier dans *PDB* est faite aléatoirement, ce dernier apparaît inconstant par rapport à *PDB_DCO*.

En général, l'heuristique la plus pertinente pour les environnements dynamique est *Cweight*, car elle cherche à satisfaire d'abord la contrainte qui conduit le plus à l'échec.

La seconde heuristique qui présente des résultats intéressants est *Cddeg*. Puisque l'heuristique *Cddeg* sélectionne la contrainte avec la variable qui a plus de contraintes avec des variables non assignées, nous pouvons dire que cette heuristique concentre la recherche sur les branches les plus dures du

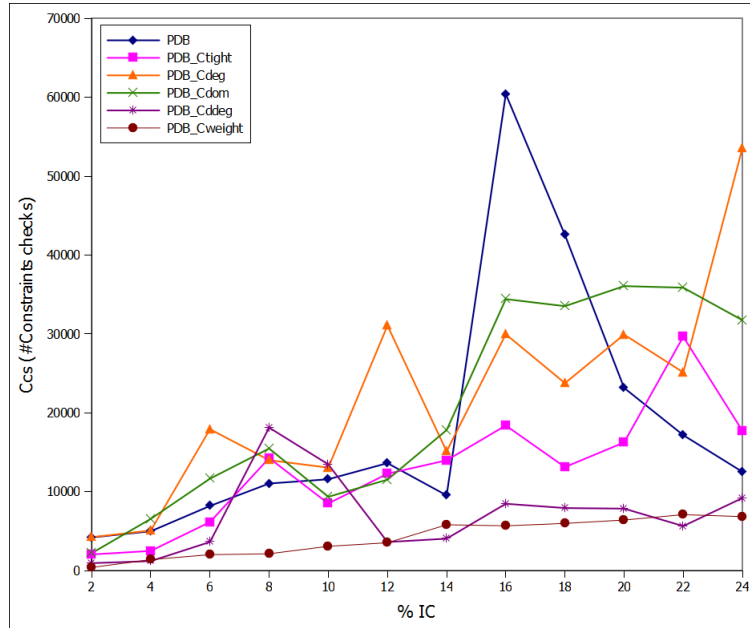


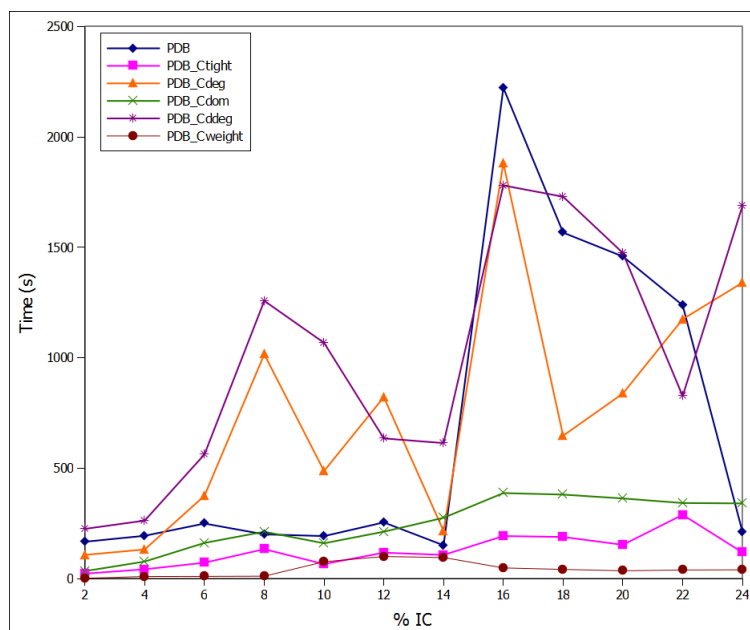
FIGURE 4.13 – Nombre de contraintes testées effectuées par *PDB* et *PDB_DCO*

réseau, dans la mesure où la contrainte choisie est plus probable à engendrer une inconsistance, pour corriger la solution dès que possible.

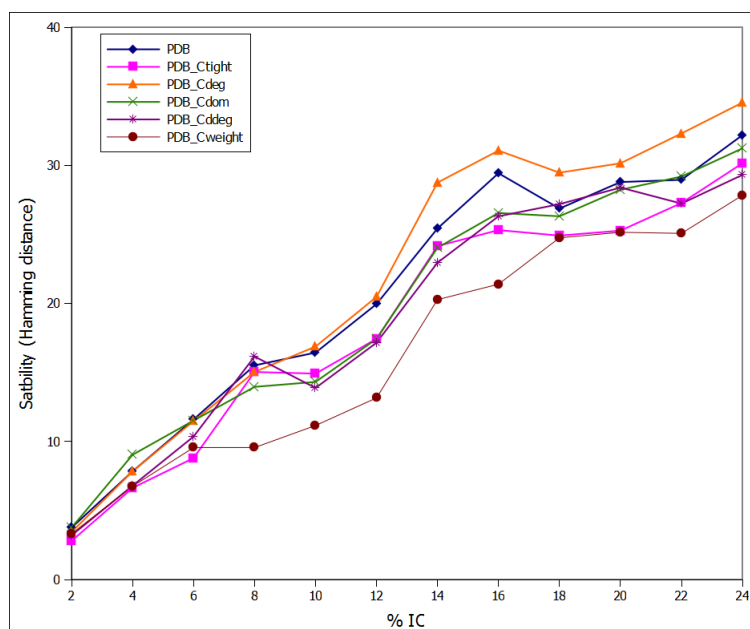
Cependant l'heuristique *Cdeg*, qui choisit la contrainte avec la variable qui a plus de contraintes dans l'ensemble du réseau a un comportement instable. Cette heuristique perturbe les affectations de variables déjà assignées ce qui rend le processus de correction instable (le nombre de *CCs*, qui parfois augmente et parfois diminue).

Comme on le sait, plus la dureté est grande plus le nombre de tuples permis est petit. L'heuristique *Ctight* classe les contraintes pour que la contrainte la plus dure soit vérifiée en premier. Ainsi, l'algorithme *PDB_DCO* aborde la partie la plus difficile du problème, les incohérences sont trouvées plus tôt et le nombre de contraintes testées est considérablement réduite. Nous pouvons dire que cette heuristique, comme *Cdom*, tente de trouver une nouvelle affectation pour la partie de réseau susceptible d'échouer aussi rapidement que possible.

La figure 4.14 montre le temps de calcul effectué par chaque heuristique pour les mêmes problèmes. Le résultat le plus intéressant est que *PDB* avec les heuristiques *Cweight*, *Ctight* et *Cdom* a besoin de moins de temps pour réparer les solutions. Notons que *Cddeg* a besoin de beaucoup de temps pour corriger la solution et c'est à cause du calcul de degré dynamique de chaque contrainte à chaque ordonnancement de *CL*. Concernant *PDB_Cdeg* et le *PDB* standard, ils dépensent plus de temps pour réviser la solution à cause

FIGURE 4.14 – Temps d'exécution effectué par PDB et PDB_DCO

de leur instabilité.

FIGURE 4.15 – Stabilité de solution effectuée par PDB et PDB_DCO

La figure 4.15 représente la stabilité de la solution réalisée par chaque

heuristique. Comme nous l'avons dit, la stabilité de la solution mesure le nombre des variables affectées par des valeurs différentes dans les deux solutions avant et après perturbation. Nous avons évalué la performance des heuristique proposées en utilisant les problèmes *DMSP* avec $n = 5$ qui est le nombre de réunions par participant, $p = 15$ qui est le nombre de participants, ce qui signifie que le nombre de variables est $\text{card}(X) = n \times p = 75$ variables.

Les expériences montrent que *PDB_Cdeg* et *PDB* standard perturbent l'affectation d'un nombre important de variables comparées à *PDB_Cweight* et *PDB_Ctight* qui maintiennent la stabilité de la solution. Par exemple, lorsque le taux de contraintes injectées est égal à 16% (~ 24 contraintes injectées), *Cdeg* change les valeurs de 31 variables ($\sim 41\%$ de la solution), tandis que *Cweight* change seulement les valeurs de 21 variables ($\sim 28\%$ de la solution).

4.2.5 Discussion

Nous avons présenté les *DCO*, des heuristiques de choix de contraintes pour réparer la solution d'un problème dynamique *DCSP*. Cette approche est capable, surtout lors du choix d'une bonne heuristique, de réparer efficacement les *DCSPs* en réordonnant dynamiquement les contraintes suivant leur pertinence. Les expérimentations que nous avons réalisées prouvent que cette approche de réparation dirigée est intéressante et les résultats montrent clairement l'efficacité de ces heuristiques combinées avec *PDB* en termes de temps d'exécution, d'effort de calcul et de stabilité de la solution. En général, les heuristiques les plus performantes pour les environnements dynamique sont *Cweight*, *Ctight* et *Cdom*. La première cherche à fixer la partie de réseau qui conduit souvent à un conflit, ainsi, elle commence par traiter la partie de réseau le plus complexe suivant un critère très raisonnable. La deuxième heuristique cherche à traiter en premier la contrainte avec le plus de tuples interdits, donc celle portant sur des variables avec plus de valeurs interdites, alors avec plus de restrictions dans les domaines. Tandis que la dernière vise à traiter d'abord la partie de réseau qui a moins de chance de trouver une affectation consistante, et qui conduira ainsi le plus probablement à un *backtrack*.

4.3 Deep degree

4.3.1 Motivation

Quand une variable change de valeur, à cause d'une inconsistance, le changement peut affecter non seulement les voisins, mais toutes les variables du réseau de contraintes, avec une certaine probabilité.

Le but du degré profond (*pdeg* pour *profound degree*) [1] est de changer, quand un conflit est détecté, la variable concernée dont le changement peut affecter le réseau le minimum possible. Ainsi, *pdeg* calcule, tout au début de la recherche, l'impact de chaque variable par rapport à sa position dans le réseau.

l'estimation de l'influence de chaque variable considère l'ensemble des contraintes du réseau tout entier, et associe à chaque contrainte un poids, plus ou moins important, relativement à sa distance par rapport à la variable concernée.

pdeg est considéré comme la somme de ces poids.

4.3.2 Exemple d'exécution

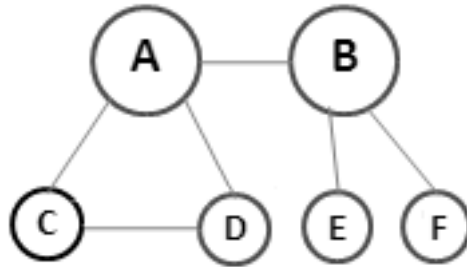


FIGURE 4.16 – réseau de contraintes

Pour expliquer l'idée de *pdeg*, considérons le réseau de contraintes dans fig. 4.16 (on considère des *CSPs* binaires, sachant que le même traitement peut être fait sur ceux *n*-aires [6]) et supposons que chaque variable a une valeur consistante. A cause d'une perturbation du *DCSP*, qui consiste à la modification de la nature de la contrainte C_{AB} , cette contrainte doit être révérifiée. On suppose que C_{AB} est violée, ainsi, une des deux variables A et B doit changer de valeur. Pour simplifier, on assume que toutes les contraintes ont la même dureté $p_2 \in [0, 1]$, qui est définie comme étant le rapport entre les tuples de valeurs interdites par chaque contrainte et l'ensemble des tuples possibles.

Si on change A , les probabilités de violation suivantes, qu'on note $P(xvyz)$, qui est la probabilité que la nouvelle valeur de x viole la contrainte entre y et z , vont être produites :

- $P(A_V AB) = \frac{|FP|-1}{|PP|-1} \simeq \frac{|FP|}{|PP|} = p_2$ (le dernier tuple $\langle A, B \rangle$ violait la courante C_{AB})
- $P(A_V AC) = \frac{|FP|}{|PP|-1} \simeq p_2$ (le dernier tuple $\langle A, C \rangle$ satisfaisait C_{AC})
- $P(A_V AD) = P(A_V AC) \simeq p_2$
- $P(A_V CD) = P((A_V AC \cap \text{changer } C \cap C_V CD) \cup (A_V AD \cap \text{changer } D \cap D_V CD))$

Les deux événements $A_V AC \cap \text{changer } C \cap C_V CD$ et $A_V AD \cap \text{changer } D \cap D_V CD$ sont disjoints, puisque si A viole C_{AC} et le premier événement est vrai, C doit changer, ainsi, $D_V CD$ dans le deuxième événement n'est plus actuel, et vice versa.

$$P(A_V CD) = P(A_V AC \cap \text{changer } C \cap C_V CD) + P(A_V AD \cap \text{changer } D \cap D_V CD)$$

$A_V AC$ et $C_V CD$ (de même pour $A_V AD$ et $D_V CD$) sont disjoints, puisqu'en $CSPs$, une seule contrainte est testée est traitée à la fois, et $A_V AC$ et $C_V CD$ sont successifs avec des valeurs différentes de C , ainsi :

$$P(A_V CD) = P(A_V AC) \times P(\text{changer } C) \times P(C_V CD) + P(A_V AD) \times P(\text{changer } D) \times P(D_V CD)$$

Puisque la violation d'une contrainte engendre le changement d'une seule variable, et puisque la variable changée dépend de l'heuristique utilisée, on remplace la probabilité de changement de chaque variable concernée par une probabilité moyenne $1/2$, ainsi :

$$P(A_V CD) \simeq p_2 \times 1/2 \times p_2 + p_2 \times 1/2 \times p_2 = p_2^2$$

- $P(A_V BE) = P(A_V AB \cap \text{change } B \cap B_V BE) \simeq p_2 \times 1/2 \times p_2 = 1/2 \times p_2^2$
- $P(A_V BF) = P(A_V BE) \simeq 1/2 \times p_2^2$

De la même manière, le changement de B générera les probabilités de violation de contraintes suivantes :

- $P(B_V AB) \simeq P(B_V BE) = P(B_V BF) \simeq p_2$
- $P(B_V AC) = P(B_V AD) \simeq 1/2 \times p_2^2$
- $P(B_V CD) \simeq 1/2 \times p_2^3$

Pour une large classe de problèmes, la dureté des contraintes n'est pas déterminée, puisque les contraintes sont représentées en intention, par conséquent, on remplace p_2 par une dureté moyenne $1/2$. Ainsi, dans l'exemple ci-dessus, sachant que $pdeg$ considère l'ensemble des contraintes du réseau et somme leurs poids dépendants des probabilités des violations :

- $Pdeg(A) = 1/2 \times 3 + 1/2^2 \times 1 + 1/2^3 \times 2 = 2$
- $Pdeg(B) = 1/2 \times 3 + 1/2^3 \times 2 + 1/2^4 \times 1 = 1.8$

Par suite, changer B est assumé être plus bénéfique que changer A , puisque son influence dans le réseau est moins importante.

4.3.3 Description de l'heuristique

Function PDEG(x, CSP)

```

1.   $pdeg \leftarrow 0$ ;
2.   $level \leftarrow 0$ ;
3.   $actual \leftarrow x$ ;
4.   $total \leftarrow x$ ;
5.  while( $actual$ )
6.     $neighbours \leftarrow getAllNeighbours(actual)$ ;
7.     $filter(neighbours, total)$ ;
8.     $actual \leftarrow getDistinct(neighbours)$ ;
9.     $a \leftarrow neighbours.size()$ ;
10.    $b \leftarrow nbConstraints(actual)$ ;
11.    $total.add(actual)$ ;
12.    $pdeg+ = 1/2^{incLevel()} \times a + 1/2^{incLevel()} \times b$ ;
13. Return  $pdeg$ ;

```

FIGURE 4.17 – Description de *PDEG*

Afin d'évaluer l'impact d'une variable donnée x dans le réseau, $pdeg$ [1] procède comme décrit dans la figure 4.17 :

Les valeurs de $pdeg$ et $level$, incrémentée à chaque fois les contraintes à évaluer sont loin de la variable concernée x , sont initialisées par 0 (lignes 1 et 2). Les listes $actual$ et $total$, initialisées par la variable x pour laquelle on calcule le $pdeg$, servent respectivement à enregistrer toutes les variables dans le niveau courant, et toutes les variables déjà considérées dans le calcul (lignes 3 et 4).

Tant que $actual$ contient des variables, c-à-d que quelques variables déjà traitées avec un voisinage non vérifié existent, l'heuristique calcule leur voisinage. Ainsi, les voisinages sont filtrés pour inclure uniquement les nouvelles variables, dans le but de ne considérer une même variable qu'une seule fois (lignes 5-7).

$getAllNeighbours()$ (ligne 6) enregistre un voisin autant de fois que le nombre de contraintes qui le lient avec les variables courantes.

$getDistinct(neighbours)$ extrait les différents voisins, qui sont mis dans $actual$. a enregistre le nombre de voisins, pas encore reconnus, des variables courantes de $actual$ avec redondance (c-à-d le nombre de contraintes entre les variables courantes et les voisins non encore comptabilisés). $nbConstraints(actual)$ compte le nombre de contraintes entre ces voisins, qui est mis dans b (lignes 8-10).

A ce stade là, la fonction ajoute à $total$ les voisins courants et à $pdeg$ leurs poids. Les poids sont calculés comme étant : la dureté moyenne $1/2$ à la puissance $level$, après incrémentation, multiplié par a , additionné à $1/2$ à la puissance $level$, après une seconde incrémentation, multiplié par b (lignes 11 et 12).

L'heuristique sort de la boucle une fois toutes les variables (automatiquement toutes les contraintes) dans le réseau de x sont comptabilisées. Ainsi, la valeur de $pdeg$ est renvoyée comme étant le degré profond de x (ligne 13).

4.3.4 Étude expérimentale

Nous avons effectué une série de tests expérimentaux pour comparer l'intégration de $pdeg$, dans l'algorithme $EPDB$, à d'autres $VOHs$.

Les heuristiques utilisées pour la réparation sont celles qui produisent le meilleur comportement de $EPDB$, à savoir le degré (deg), la taille du domaine (dom), le nombre de conflits ($conf$) et le rapport entre la taille du domaine et le degré pondéré ($dom/wdeg$) [38].

Nous évaluons la performance en termes de tests de contraintes (CCs), temps de calcul ($Time(sc)$) et Hamming distance(HD) [52, 19].

Toutes les expérimentations ont été effectuées sur la plate-forme *Java*.

Expérimentations pour les DCSPs binaires aléatoires uniformes

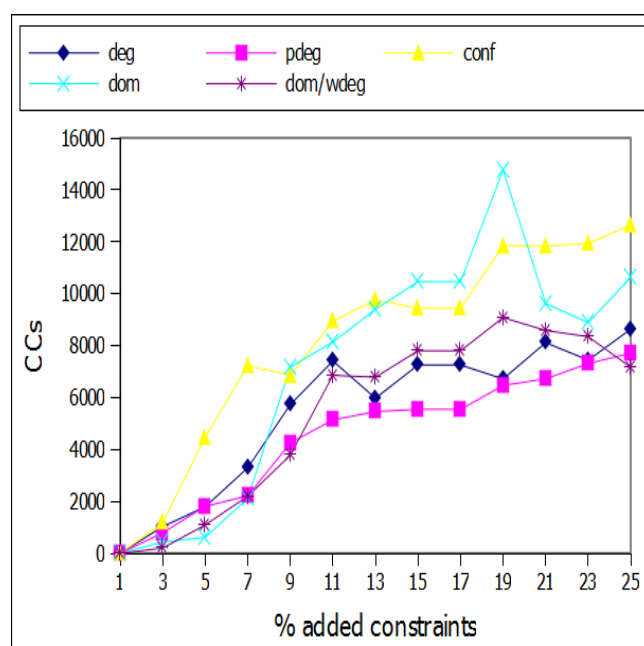
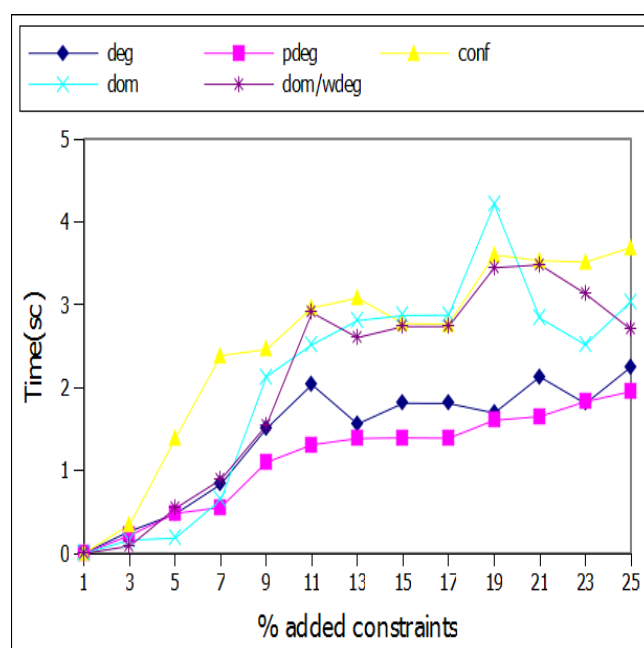
Les tests ont été effectués sur des problèmes de densité faible et élevée, respectivement $< 20, 10, 0.25, 0.6 >$ et $< 20, 10, 0.75, 0.27 >$. Les duretés des contraintes sont sélectionnées, tels que les zones sont aussi complexes que possible (transition de phase) alors que les problèmes ont une solution.

Pour chaque paire $< p1, p2 >$, 20 différentes instances sont traitées par chaque algorithme et les résultats actuels sont les moyennes de ces 20 exécutions. Le taux de contraintes injectées $\%IC = \frac{|nouvelles\ contraintes|}{|contraintes\ totales|}$. Cela indique le taux de perturbation qui sera traité en utilisant $EPDB$. IC varie de 1% à 25%, tel que pour chaque perturbation donnée de $m\%$, les contraintes ajoutées sont les mêmes que celles de la perturbation de $m - 1\%$, plus 1% de contraintes supplémentaires. Les 30 problèmes finaux (après la perturbation) sont les mêmes pour chaque taux de perturbation.

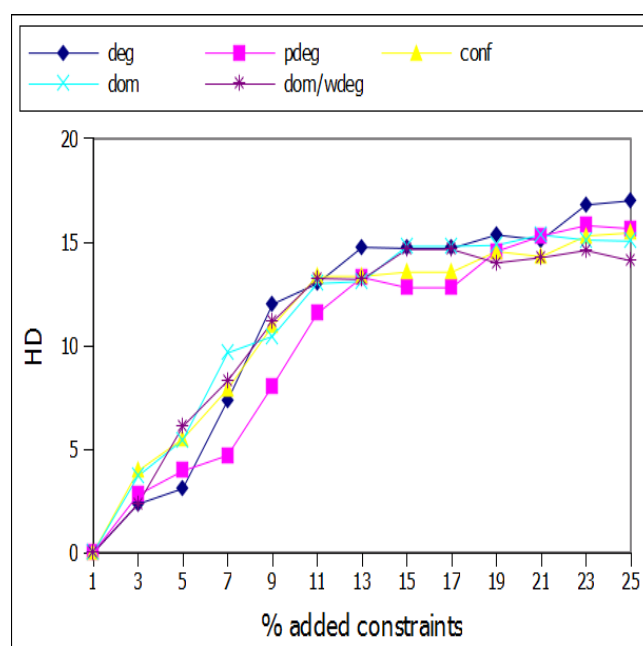
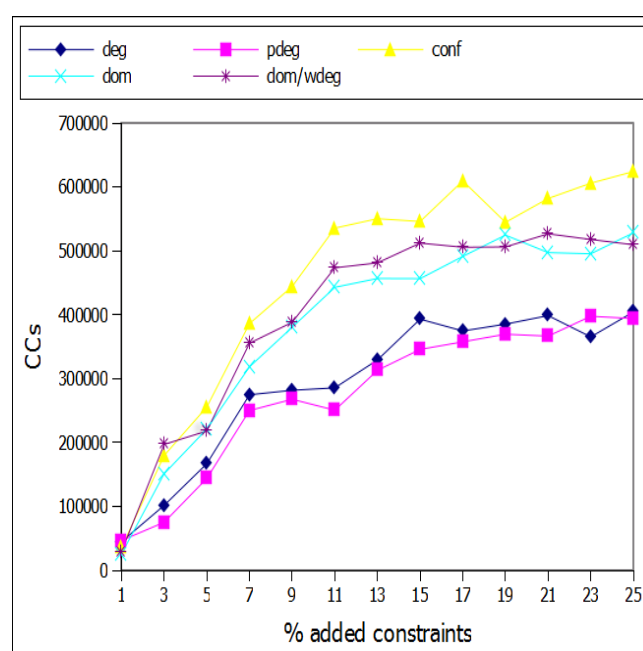
Les figures 4.18, 4.19 et 4.20 présentent l'efficacité des heuristiques appliquées à $EPDB$ pour les régions les plus complexes, c-à-d la transition de phase, de la densité faible choisie. En termes de nombre de contraintes testées CCs , pour des perturbations entre 10% et 24%, $pdeg$ se comporte le mieux, ailleurs, $dom/wdeg$ et dom sont les plus efficaces. Sinon, le deg est le plus proche de $pdeg$, qui semble le plus stable.

Dans le graphique de temps d'exécution, $dom/wdeg$ recule (également pour dom , mais pas de la même manière), en raison des calculs supplémentaires des heuristiques dynamiques dom et $wdeg$, et surtout pour $wdeg$. $pdeg$ est presque toujours le plus performant.

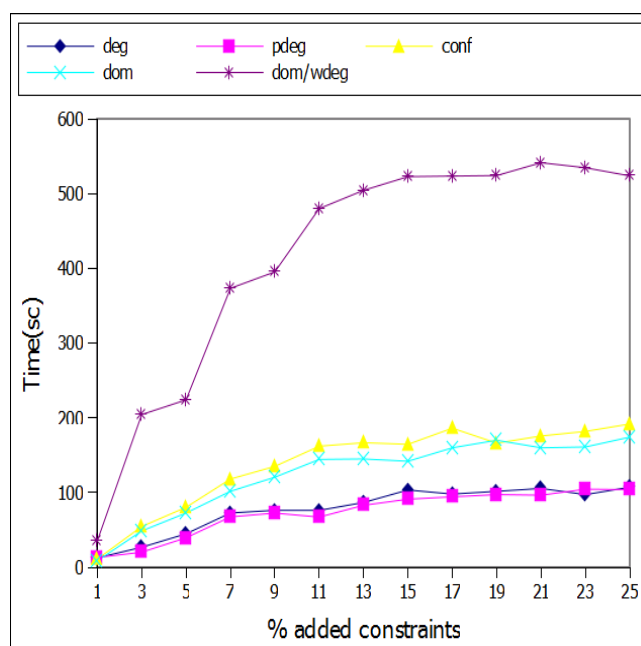
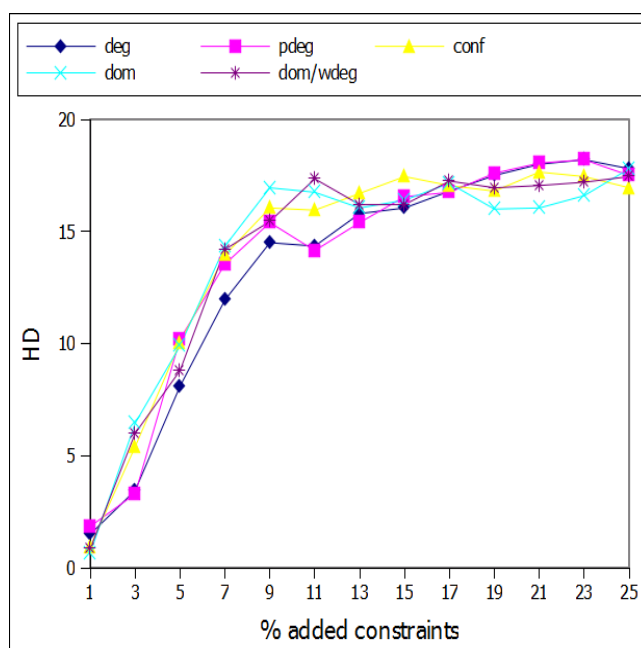
En ce qui concerne la distance de *Hamming* HD , il existe une concurrence entre les heuristiques, et $pdeg$ reste un bon concurrent dans la plupart des régions.

FIGURE 4.18 – Nombre de CCs ($p_1=0.25$, $p_2=0.60$)FIGURE 4.19 – Temps d'exécution ($p_1=0.25$, $p_2=0.60$)

Les figures 4.21, 4.22 et 4.23 montrent l'efficacité des heuristiques pour

FIGURE 4.20 – Distance de Hamming ($p_1=0.25$, $p_2=0.60$)FIGURE 4.21 – Nombre de CCs ($p_1=0.75$, $p_2=0.27$)

les zones les plus complexes de la densité élevée $p_1 = 0,75$. En termes de nombre de CCs et temps d'exécution, $pdeg$ semble se comporter le mieux,

FIGURE 4.22 – Temps d'exécution ($p_1=0.75$, $p_2=0.27$)FIGURE 4.23 – Distance de Hamming ($p_1=0.75$, $p_2=0.27$)

bien qu'il n'est pas assez performant par rapport à *deg*. En ce qui concerne la distance de *Hamming*, aucune heuristique n'est absolument la meilleure.

Expérimentations concernant le problème de planification de réunions dynamiques *DMSP*

Dans ce qui suit, nous présentons nos résultats pour la classe $\langle 20, 5, 15, 5, 10, 1, 70 \rangle$ et nous varions le taux de contraintes modifiées de $\sim 1\%$ à $\sim 25\%$. 20 différentes instances ont été générées et résolues en intégrant chacune des heuristiques dans *EPDB*.

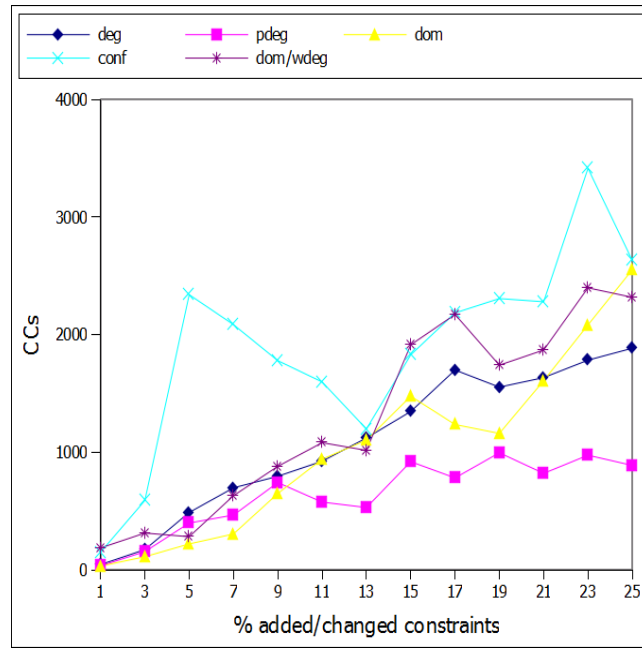


FIGURE 4.24 – Nombre de *CCs*

Les figures 4.24, 4.25 et 4.26 montrent l'efficacité des heuristiques en termes de nombre de *CCs*, temps d'exécution et *HD*. *Pdeg* mène à une excellente amélioration dans la plupart des régions (à partir de 10% de contraintes perturbées en terme de *CCs*, 7% en terme de temps d'exécution et 11% en terme de *HD*).

4.3.5 Discussion

Les heuristiques de calcul de voisinage, à savoir *deg* et *pdeg*, mènent aux meilleurs résultats surtout en termes de nombre de contraintes testées et temps d'exécution.

Pour les densités faibles, *pdeg* se comporte mieux que *deg*, plus remarquablement que pour les densités élevées. En fait, dans les densités élevées, presque toutes les variables sont connectées, c-à-d, pour chaque variable donnée, presque toutes les autres variables du réseau sont des voisins. Par conséquent, calculer *pdeg* d'une variable revient presque à calculer son *deg*.

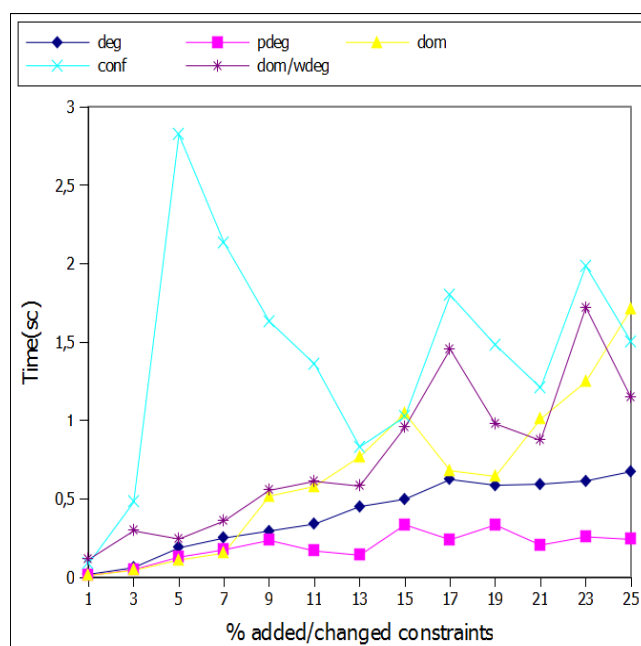


FIGURE 4.25 – Temps d'exécution

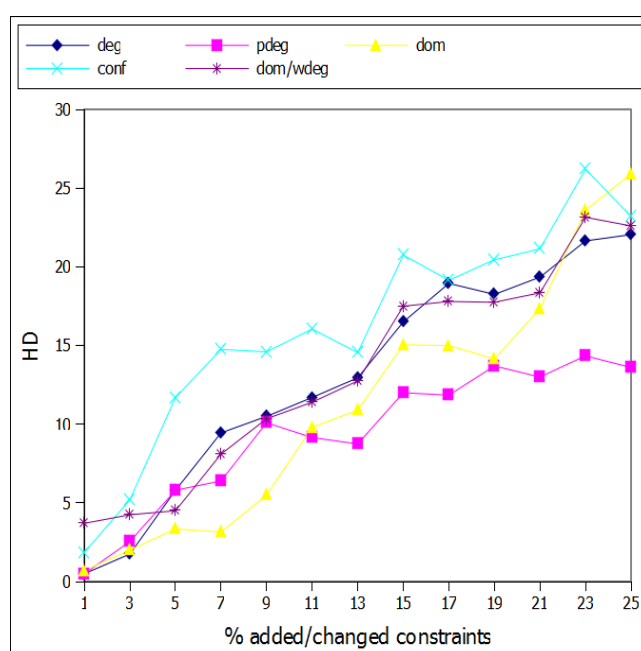


FIGURE 4.26 – Distance de Hamming

pdeg mène, surtout pour les densités faibles, à une distance de *Hamming* (*HD*) minimale. En fait, l'idée de *pdeg* est de changer la valeur de la variable

la moins contraignante du réseau, c'est-à-dire celle qui, lors de la modification de sa valeur, entraînera le moins de changement pour d'autres variables. Et sachant que les variables avant la correction ont des valeurs *SVAs* [19, 52], un changement minimum de variables dans le réseau est un changement minimum de *SVAs*, ce qui engendre la conservation d'un maximum de *SVAs*. Bien que les heuristiques *dom*, *conf* et *dom/wdeg*, dans certaines régions, vérifient moins de contraintes que les heuristiques statiques *deg* et *pdeg*, elles consomment plus de temps de calcul en raison de leur aspect dynamique.

4.4 Improved Partial-order Dynamic Backtracking

4.4.1 Motivation

Comme vu précédemment, *EPDB* est largement flexible par rapport à *PDB* : il exploite la flexibilité de *PDB* pour construire un ordre entre les variables, en utilisant les *VOHs*, afin de converger vers la terminaison en utilisant un nombre minimal de tests de contraintes. L'inconvénient des deux approches est la vérification de l'ordre courant entre les variables concernées avant la génération d'un *nogood*, dans les situations de vidage de domaine ou de conflit. Cette vérification d'ordres, qui sont stockés dans les *nogoods* et les *safety conditions*, est nécessaire pour éviter de boucler, mais elle consomme souvent un temps important, car dans la plupart des cas, les ordres entre les variables ne sont pas directement déduits.

L'exemple suivant explique l'inconvénient du processus de vérification d'ordres à la fois dans *PDB* et *EPDB*. *EPDB* est utilisé pour réparer un *CSP* dynamique. La contrainte courante à tester est C_{12} , qui est violée. Un *nogood* concernant X_1 et X_2 doit être alors généré, ainsi, l'ordre actuel entre X_1 et X_2 est vérifié. Les ordres actuels sont affichés dans tab. 4.5 : En fait,

TABLE 4.5 – ordres existants

safety conditions	nogoods
$X_2 < X_i$ (1)	$X_j = v_j \implies X_k \neq v_k$ (4)
$X_i < X_m$ (2)	$X_k = v_k \implies X_l \neq v_l$ (5)
$X_j < X_k$ (3)	$X_k = v_k \implies X_1 \neq v_1$ (6)
	$X_i = v_i \implies X_j \neq v_j$ (7)

$X_2 < X_1$, puisque $X_2 < X_i$ (1), $X_i < X_j$ (7), $X_j < X_k$ (3) et $X_k < X_1$ (6). Cependant, la déduction de l'ordre entre X_1 et X_2 n'est pas directement effectuée, puisque les variables impliquées dans cet ordre, à savoir X_2 , X_i , X_j , X_k et X_1 , sont impliqués dans la partie gauche d'autres *nogoods* et *safety conditions* stockés avant les ordres concernés. X_i est impliqué dans (2) qui est placé avant (7) et X_k dans (5) qui précède (6), ainsi, (2) et (5) sont également vérifiés.

En outre, la détermination de l'ordre consomme un temps coûteux même lors qu'aucun ordre n'existe entre les variables en conflit. Malheureusement, cela doit être prouvé en vérifiant tous les *nogoods* et *safety conditions* qui les concernent directement (les ordres où ils apparaissent) et indirectement (les ordres où leurs voisins apparaissent directement et indirectement).

Par exemple, supposons encore une fois que *EPDB* est utilisé pour réparer un *DCSP*, telle que la contrainte actuelle à vérifier C_{12} est violée. Pour générer un *nogood*, l'ordre actuel entre X_1 et X_2 est vérifié. Les ordres actuels sont affichés dans tab. 4.6 : L'approche vérifie (1) puis (2) pour tester si X_2 précède X_1 , et (5) puis (4) pour tester si X_1 précède X_2 , avant de conclure

TABLE 4.6 – ordres existants

safety conditions	nogoods
$X_2 < X_i$ (1)	$X_k = v_k \implies X_l \neq v_l$ (4)
$X_i < X_m$ (2)	$X_1 = v_1 \implies X_k \neq v_k$ (5)
$X_j < X_n$ (3)	

qu'aucun ordre n'existe entre les deux variables, et établir un ordre en utilisant la *VOH* spécifiée.

L'objectif de notre approche proposée, à savoir *Improved Partial-order Dynamic Backtracking (IPDB)* [2], consiste à déduire directement les ordres entre les variables tout en bénéficiant de la stratégie de réparation de *EPDB*. Par conséquent, *IPDB* procède exactement comme *EPDB*, mais utilise une structure de données additive appelée *successors*. Cette structure permet, comme son nom l'indique, d'avoir l'information concernant tous les successeurs de chaque variable dans le réseau. Ainsi, concernant l'exemple de tab. 4.5, la structure *successors* contient les informations indiquées dans tab. 4.7. Nous supposons que l'ordre de stockage de tous les *nogoods* et *safety conditions* est identique à leur indice. Le premier ordre est $X_2 < X_i$ (1), ainsi, X_i et tous ses successeurs sont ajoutés aux successeurs de X_2 . Ils sont également ajoutés aux successeurs des variables où X_2 est un successeur. Par conséquent, cet ordre cause l'addition, dans *successors*, des

variables suivants $\xrightarrow{(1)} \text{rightarrow}$ jusqu'à $\xrightarrow{(nbr)}$ tel que $nbr \neq 1$. Les informations ajoutées suivantes sont respectivement $X_i < X_m$ (2), $X_j < X_k$ (3) et $X_j = v_j \implies X_k \neq v_k$ (4), qui sont ajoutés de manière similaire à la première. Ensuite, l'information $X_k = v_k \implies X_l \neq v_l$ (5) est ajoutée, X_l est ajouté aux successeurs de X_k . En outre, il est ajouté aux successeurs de X_j , qui a X_k en tant que successeur. Comme X_j a X_k deux fois dans les successeurs, la liste des successeurs de X_k est également ajoutée deux fois. Ce nombre de récurrence est important, afin de ne pas perdre une information d'ordre sur tous les successeurs, lorsqu'une relation entre deux variables est supprimée (c-à-d une *safety condition*) et d'autres relations encore existantes (c-à-d des *nogoods* ou *safety condition*) impliquent les mêmes successeurs. Ensuite, les informations $X_k = v_k \implies X_1 \neq v_1$ (6) sont ajoutées de la même manière à (5). Enfin, puisque $X_i = v_i \implies X_j \neq v_j$ (7), X_j et tous ses successeurs sont ajoutés aux successeurs de X_i . De plus, ils sont ajoutés aux successeurs de X_2 , qui a X_i en tant que successeur. Ainsi, cette fois ci, pour générer le *nogood* concernant X_1 et X_2 , causé par la violation de C_{12} , l'approche vérifie si X_2 est dans *successors*(X_1), ce qui ne l'est pas, puis vérifie si X_1 existe dans *successors*(X_2), ce qui est vrai. Par conséquent, le *nogood* généré est celui avec X_1 dans la conclusion. Ainsi, les ordres actuels sont affichés dans tab. 4.8 :

et les *successors* sont indiqués dans tab. 4.9 :

TABLE 4.7 – Successeurs

Variable	successors
X_1	
X_2	$\xrightarrow{(1)} X_i \xrightarrow{(2)} X_m \xrightarrow{(7)} X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_i	$\xrightarrow{(2)} X_m \xrightarrow{(7)} X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_j	$\xrightarrow{(3)} X_k \xrightarrow{(4)} X_k \xrightarrow{(5)} X_l \rightarrow X_l \xrightarrow{(6)} X_1 \rightarrow X_1$
X_k	$\xrightarrow{(5)} X_l \xrightarrow{(6)} X_1$
X_l	
X_m	

TABLE 4.8 – ordres existants

safety conditions	nogoods
$X_2 < X_i$ (1)	$X_j = v_j \implies X_k \neq v_k$ (4)
$X_i < X_m$ (2)	$X_k = v_k \implies X_l \neq v_l$ (5)
$X_j < X_k$ (3)	$X_k = v_k \implies X_1 \neq v_1$ (6)
	$X_i = v_i \implies X_j \neq v_j$ (7)
	$X_2 = v_2 \implies X_1 \neq v'_1$ (8)

L'autre cas où les *successors* sont manipulés, c'est lors de suppression

TABLE 4.9 – Successeurs

Variable	Successors
X_1	
X_2	$\rightarrow X_i \rightarrow X_m \rightarrow X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1 \xrightarrow{(8)} X_1$
X_i	$\rightarrow X_m \rightarrow X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_j	$\rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_k	$\rightarrow X_l \rightarrow X_1$
X_l	
X_m	

d'ordres. Cette suppression concerne la suppression des anciens ordres (*safety conditions*). Pour étudier ce cas, nous supposons que nous supprimons respectivement toutes les informations de *safety conditions*, ainsi, les *successors* mis à jour sont indiqués dans tab. 4.10. Le premier ordre supprimé est $X_2 < X_i$ (1). Cette suppression provoque le retrait, à partir de *successors*, des variables rayées après $\xrightarrow{\cancel{X}}$ jusqu'à $\xrightarrow{(nbr)}$ tel que $nbr \neq 1$. En fait, X_i et tous ses successeurs sont supprimés des successeurs de X_2 . Les informations suivantes supprimées sont $X_i < X_m$ (2), ce qui entraîne la suppression de X_m des successeurs de X_i . Enfin, les dernières informations supprimées

concernent $X_j < X_k$ (3). X_k et toute sa liste de successeurs est supprimée des successeurs de X_j . En outre, ils sont retirés des successeurs de X_i , qui contient X_j dans les successeurs.

TABLE 4.10 – Successeurs

Variable	Successors
X_1	
X_2	~~(1)~~ $X_i \rightarrow X_m \rightarrow X_j \rightarrow X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1 \rightarrow X_1$
X_i	~~(2)~~ $X_m \rightarrow X_j$ ~~(3)~~ $X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_j	~~(3)~~ $X_k \rightarrow X_k \rightarrow X_l \rightarrow X_l \rightarrow X_1 \rightarrow X_1$
X_k	$\rightarrow X_l \rightarrow X_1$
X_l	
X_m	

4.4.2 Description de la structure de données proposée

Comme nous l'avons indiqué précédemment, *IPDB* [2] est exécuté exactement comme *EPDB*, avec la différence de la façon de tester les ordres, ce qui est nécessaire pour générer les *nogoods*. Ainsi, *IPDB* utilise la structure d'ordres supplémentaire *successors* et vérifie les successeurs des variables concernées par le *nogood* pour tester l'existence de l'ordre entre ces variables (ligne 30 de fig.4.4).

La structure de données *successors* est manipulée dans deux phases :

- quand un *nogood* est généré, lors d'un conflit ou d'une situation de viadage de domaine, la procédure *AddRelation* est appelée pour ajouter les informations concernant les nouveaux ordres établis par ce *nogood* (après les instructions dans les lignes 9 et 35 de fig.4.4).
- quand un *safety condition* est supprimé, la procédure *RemoveRelation* est appelée pour supprimer les informations concernant les ordres causés par ce *safety condition* (ligne 26 de fig.4.4).

La structure utilisée *successors* est une matrice *successors*[n][n], tel que n est le nombre de variables du *DCSP*. *successors*[i][j] stocke l'information concernant le nombre de fois que les variables d'indice j apparaissent dans *successors* des variables d'indice i .

Lorsqu'un *nogood* est ajouté entre X_1 et X_2 , la procédure *AddRelation* (fig. 4.27) est appelée. Elle stocke l'information concernant les successeurs de X_2 (ligne 3) et combien de fois ils apparaissent dans son *successors* (ligne 4). Ensuite, X_2 est ajouté dans *successors* de X_1 (ligne 5). En outre, les successeurs de X_2 sont ajoutés, autant de fois ils apparaissent dans *successors* de X_2 , dans *successors* de X_1 (lignes 6 et 7). X_2 et ses successeurs sont également ajoutés dans *successors* des variables X_j , où X_1 est un successeur. Par

Procedure AddRelation(X_1, X_2)

1. **for each** X_i **in** $DCSP.variables$
2. **if** $successors[X_2][X_i] > 0$ **then**
3. $succeeds.add(X_i)$;
4. $count.add(successors[X_2][X_i])$;
5. increment $successors[X_1][X_2]$;
6. **for** i **from** 0 **to** size of $succeeds$
7. add $count.get(i)$ in $successors[X_1][succeeds.get(i)]$;
8. **for each** X_j **in** $DCSP.variables$
9. **if** $successors[X_j][X_1] > 0$ **then**
10. add $successors[X_j][X_1]$ in $successors[X_j][X_2]$;
11. **for** k **from** 0 **to** size of $succeeds$
12. add $count.get(k) \times successors[X_j][X_1]$ in $successors[X_j][succeeds.get(k)]$;

Procedure RemoveRelation(X_1, X_2)

13. **for each** X_i **in** $DCSP.variables$
14. **if** $successors[X_2][X_i] > 0$ **then**
15. $succeeds.add(X_i)$;
16. $count.add(successors[X_2][X_i])$;
17. decrement $successors[X_1][X_2]$;
18. **for** i **from** 0 **to** size of $succeeds$
19. subtract $count.get(i)$ from $successors[X_1][succeeds.get(i)]$;
20. **for each** X_j **in** $DCSP.variables$
21. **if** $successors[X_j][X_1] > 0$ **then**
22. subtract $successors[X_j][X_1]$ from $successors[X_j][X_2]$;
23. **for** k **from** 0 **to** size of $succeeds$
24. subtract $count.get(k) \times successors[X_j][X_1]$ from $successors[X_j][succeeds.get(k)]$;

FIGURE 4.27 – manipulation de *successors*

conséquent, X_2 est ajouté dans *successors* de X_j autant de fois X_1 apparaît dans *successors* de X_j (ligne 10), et ses successeurs sont ajoutés, pour chaque apparition de X_1 dans *successors* de X_j , autant de fois ils apparaissent dans *successors* de X_2 (lignes 11 et 12).

Lorsqu'une *safety condition* entre X_1 et X_2 est supprimée, la procédure *RemoveRelation* (fig. 4.27) est appelée. Cette procédure se déroule exactement comme *AddRelation*, en supprimant les variables des *successors*, autant de fois elles sont ajoutées par *AddRelation*.

4.4.3 Preuve de terminaison

Comme démontré plus haut, *EPDB* termine. *Improved PDB* se comporte exactement comme *EPDB*, la différence concerne la façon de déterminer les ordres. En fait, dans *EPDB*, les ordres sont déterminés à partir des *nogoods* et des *safety conditions* enregistrés, tandis que dans *IPDB*, les ordres sont déterminés à partir de la structure de données *successors*, qui reflète exactement l'ensemble des ordres enregistrés dans les *nogoods* et

les *safety conditions*. Par conséquent, *IPDB* termine également.

4.4.4 Étude expérimentale

Dans cette section, nous évaluons *IPDB* par rapport à *EPDB*, pour montrer l'amélioration de notre approche proposée par rapport à *EPDB*. Nous comparons également *IPDB* à une approche efficace pour résoudre les *CSPs*, à savoir le *Maintaining Arc Consistency MAC-2001* [17], afin d'identifier le taux maximal de perturbation qui tolère la réparation, c-à-d à partir de quel taux de perturbation, l'utilisation d'un bon algorithme de résolution est plus efficace pour traiter les *DCSPs*.

Nous avons intégré divers *VOHs* à la fois dans *EPDB* et *IPDB*, à savoir *deg*, *dom*, *conf*, *ddeg*, *wdeg*, *dom/deg*, *dom/ddeg*, *dom/wdeg* et *lexico*. Mais comme *ddeg*, *wdeg*, *dom/ddeg*, *dom/wdeg* et *lexico* ont un comportement très modeste, nous montrons uniquement les résultats concernant les autres *VOHs*. *MAC-2001* est combiné à l'heuristique *dom/deg* comme dans [48]. Nous notons que la stratégie de cohérence d'arc peut également être améliorée en utilisant *dom/wdeg* [19], mais comme montré dans [50], cette heuristique est efficace essentiellement en combinaison avec *AC-3*. En tant que critères d'évaluation, nous mesurons le temps d'exécution (*time*) par millisecondes et le nombre de tests de contraintes (*CCs*). Les expérimentations sont effectuées sur la plate-forme *Java*. Ils concernent un *PC Pentium IV* (d'un processeur de 2.8 *GHz* et de 3.24 *Go* de *RAM*).

Expérimentations sur les *DCSPs* binaires aléatoires uniformes

Expérimentations concernant la réparation : Les expérimentations suivantes concernent des *DCSPs* binaires aléatoires et uniformes, les tests sont effectués sur des problèmes à densité faible et élevée, respectivement $\langle 20, 10, 0.25, 0.60 \rangle$ et $\langle 20, 10, 0.75, 0.25 \rangle$.

Les paramètres sont choisis de telle sorte que les instances du problème sont situées à la phase de transition [23] tout en étant satisfaisantes. Les problèmes sont relativement plus difficiles à résoudre dans la région de transition de phase, qui est la région entre une région sous contrainte où toutes les instances sont presque sûrement satisfaisantes et une région trop contrainte où toutes les instances sont presque sûrement insatisfaisantes.

Pour chaque paire $\langle p1, p2 \rangle$, 30 différentes instances sont traitées par chaque algorithme et les résultats actuels sont les moyennes de ces 30 exécutions. Le taux de contraintes injectées $\%IC = \frac{|\text{nouvelles contraintes}|}{|\text{contraintes totales}|}$. Cela indique le taux de perturbation qui sera traité en utilisant à la fois *IPDB* et *EPDB*. *IC* varie de 1% à 100%, tel que pour chaque perturbation donnée de $m\%$, les contraintes ajoutées sont les mêmes que celles de la perturbation de $m - 1\%$, plus 1% de contraintes supplémentaires. Les 30 problèmes finaux

(après la perturbation) sont les mêmes pour chaque taux de perturbation. Pour les approches *DCSP*, à savoir *IPDB* et *EPDB*, l'effort calculé est celui de la réparation de la solution passée, c-à-d qui existait avant la perturbation, alors que l'effort calculé pour *MAC-2001* est celui pour résoudre tout le problème.

En ce qui concerne le nombre de *CCs* (1^{re} partie des tabs. 4.11 et 4.12), utiliser *IPDB* avec une *VOH* donnée revient à utiliser *EPDB* avec la même *VOH*, car les deux approches se comportent de la même manière. Mais en ce qui concerne le temps d'exécution (2^{me} partie des tabs. 4.11 et 4.12), *IPDB* excède remarquablement *EPDB*, quelle que soit la *VOH* utilisée.

L'utilisation de *IPDB* pour réparer des problèmes qui font l'objet d'une petite perturbation est plus efficace que son utilisation pour réparer des problèmes fortement perturbés. Cependant, à partir d'une certaine perturbation, plus ou moins intéressante selon la *VOH* utilisée, l'effort utilisé par *IPDB* pour réparer les problèmes est presque le même. Cet effort n'est pas stable, mais il est sujet à des chutes brusques.

Pour la faible densité (tab. 4.11), *IPDB_{deg}* dépasse *MAC-2001* concernant le nombre de *CCs* et le temps d'exécution. *IPDB_{dom}* est également un bon concurrent de *MAC-2001*, en particulier en ce qui concerne le temps d'exécution, jusqu'à une perturbation moyenne. L'intégration d'autres *VOHs* dans *IPDB* est bénéfique uniquement pour les petites perturbations.

Pour la densité élevée (tab. 4.12), *IPDB_{deg}* dépasse souvent *MAC-2001*, pas aussi bien que pour la densité faible, concernant le temps d'exécution. L'utilisation d'autres *VOHs*, sauf le *dom* dans certaines situations, n'est efficace que pour les petites perturbations.

Expérimentations concernant la résolution : Les expérimentations précédentes concernent des problèmes satisfaisants dans des zones difficiles de densité faible et élevée. Ces expériences montrent que *IPDB* en utilisant certaines *VOHs*, surtout *deg*, est un bon concurrent de *MAC-2001* même pour résoudre (100% de *IC*).

Afin de montrer où *MAC-2001* est plus efficace que *IPDB*, nous exécutons les deux approches sur des problèmes aléatoires, pas nécessairement satisfaisants, avec les mêmes paramètres précédents n , d et $p1$. La dureté de contraintes varie de $p2 = 0, 1$ à $p2 = 0, 9$. 30 différentes instances sont traitées par chaque algorithme et les résultats présentés sont les moyennes de ces 30 exécutions.

Les figures 4.28 et 4.29 montrent que *IPDB*, quelle que soit la *VOH* utilisée, est souvent plus efficace que *MAC-2001* pour traiter les problèmes satisfaisants, c-à-d $< p1 = 0.25, p2 \in [0.1, 0.60[>$ et $< p1 = 0.75, p2 \in [0.1, 0.25[>$, alors que *MAC-2001* est efficace pour détecter la non-satisfaction des problèmes dans $< p1 = 0.25, p2 \in]0.60, 0.9] >$ et $< p1 = 0.75, p2 \in]0.25, 0.9] >$. Dans les zones complexes, c-à-d les phases de transition $< p1 = 0.25, p2 =$

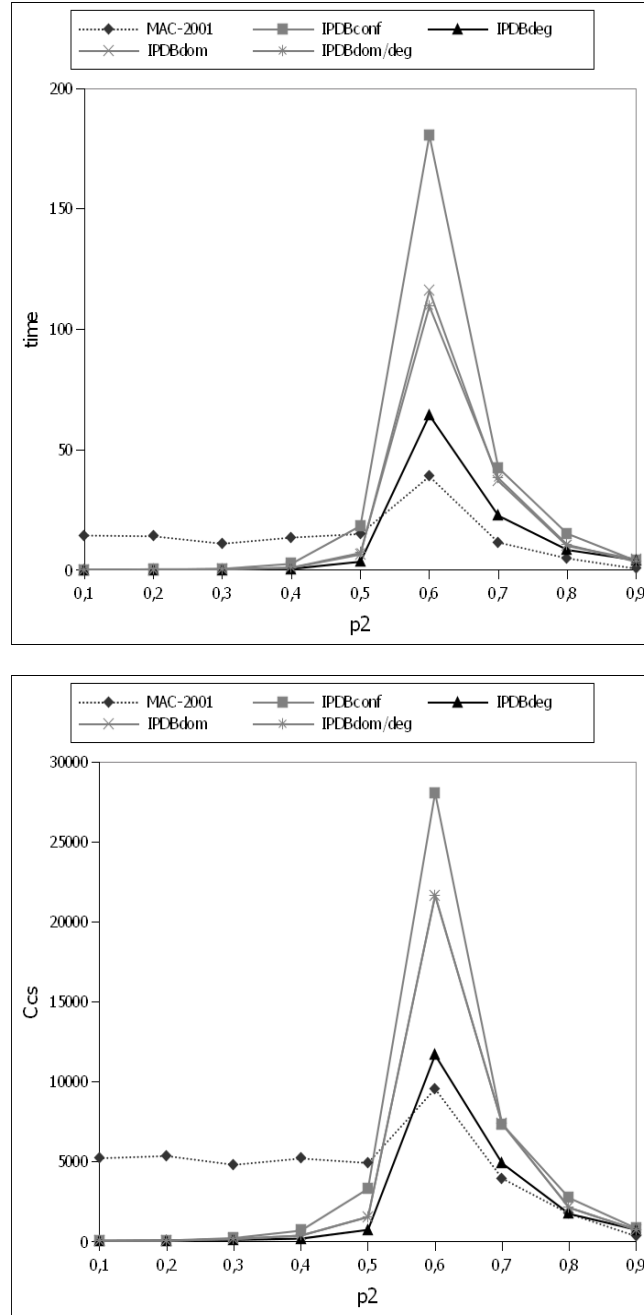
TABLE 4.11 – Nombre de *CCs* et temps d'exécution en (*ms*) ($p1 = 0.25$, $p2 = 0.60$)

%IC	MAC	IPDB				EPDB				
		<i>conf</i>	<i>deg</i>	<i>dom</i>	<i>dom/deg</i>	<i>conf</i>	<i>deg</i>	<i>dom</i>	<i>dom/deg</i>	
1	4992	0	0	0	0	0	0	0	0	CCs
4		467	315	412	172	467	315	412	172	
7		3327	2862	1094	2945	3327	2862	1094	2945	
10		4685	3430	4012	3742	4685	3430	4012	3742	
13		5933	3581	2901	5420	5933	3581	2901	5420	
16		5812	1783	2670	4437	5812	1783	2670	4437	
19		4691	1967	3084	5485	4691	1967	3084	5485	
22		7824	3257	3836	6028	7824	3257	3836	6028	
25		8748	2959	4465	6184	8748	2959	4465	6184	
31		7226	3559	6841	14209	7226	3559	6841	14209	
37		11823	3072	6198	6452	11823	3072	6198	6452	
43		11670	3097	2723	9433	11670	3097	2723	9433	
49		10958	3213	6708	9693	10958	3213	6708	9693	
55		17304	4720	5771	13064	17304	4720	5771	13064	
61		16519	4209	8068	17922	16519	4209	8068	17922	
67		13945	2930	7914	10276	13945	2930	7914	10276	
73		10243	3038	6012	12592	10243	3038	6012	12592	
79		13867	4063	6770	10650	13867	4063	6770	10650	
85		5969	3170	9630	10926	5969	3170	9630	10926	
91		8619	3333	9722	10846	8619	3333	9722	10846	
97		7756	3729	4605	7996	7756	3729	4605	7996	
100		10557	3516	7660	7660	10557	3516	7660	7660	
1	18	0	0	0	0	0	0	0	0	time
4		1	1	1	0	40	19	34	10	
7		10	6	3	7	265	165	51	193	
10		12	7	9	9	361	190	239	256	
13		16	8	7	13	479	200	197	377	
16		16	4	6	11	491	83	182	292	
19		12	4	8	13	375	94	235	368	
22		20	7	9	15	638	158	226	407	
25		22	6	11	14	712	142	282	387	
31		19	7	16	34	591	160	421	1029	
37		32	6	15	15	941	149	446	429	
43		31	7	6	21	926	143	190	544	
49		29	7	15	23	889	148	381	644	
55		46	10	14	30	1321	213	388	904	
61		42	9	18	42	1218	187	467	1235	
67		37	6	18	25	1139	134	499	705	
73		27	7	14	30	789	129	400	871	
79		35	9	15	25	1040	174	410	781	
85		16	7	22	27	428	134	615	834	
91		22	7	22	25	630	146	632	705	
97		20	8	11	20	578	166	255	558	
100		28	7	17	17	815	156	482	483	

TABLE 4.12 – Nombre de *CC*s et temps d'exécution en (*ms*) ($p1 = 0.75, p2 = 0.25$)

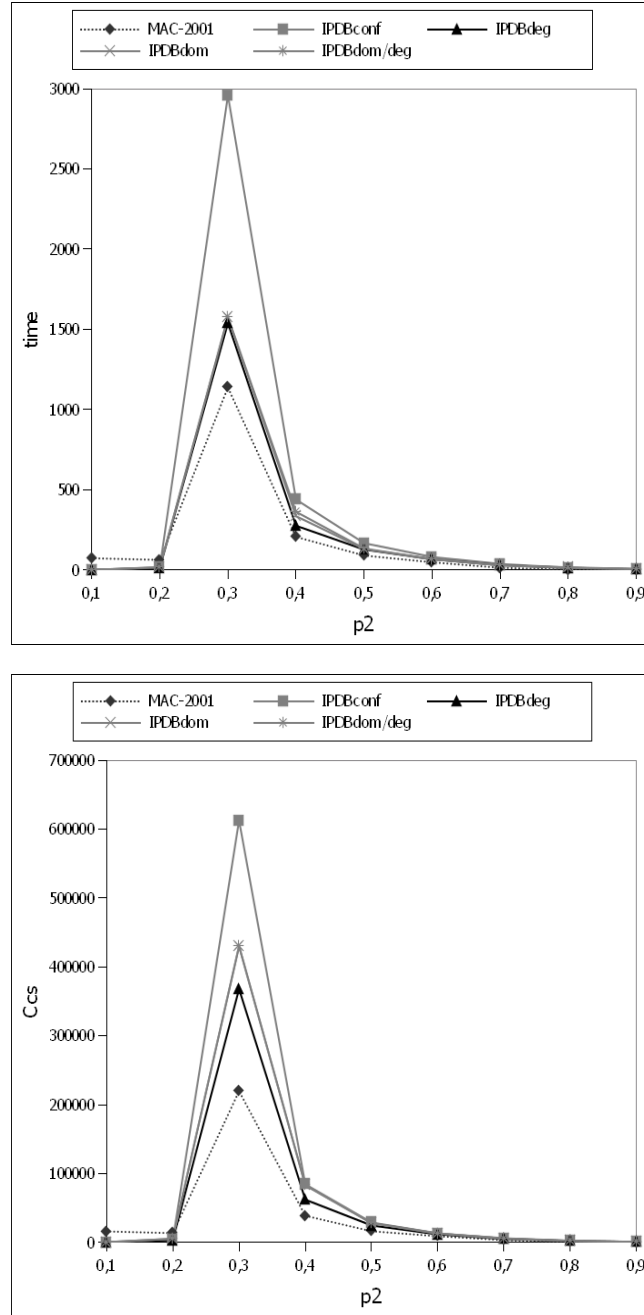
%IC	MAC	IPDB				EPDB				
		conf	deg	dom	dom/deg	conf	deg	dom	dom/deg	
1	102735	30525	34660	22967	60452	30525	34660	22967	60452	CCs
4		161656	58131	125013	330288	161656	58131	125013	330288	
7		200448	97458	155499	287374	200448	97458	155499	287374	
10		211187	81009	126958	365003	211187	81009	126958	365003	
13		281738	94322	231430	475341	281738	94322	231430	475341	
16	102735	367999	93922	184632	344709	367999	93922	184632	344709	
19		321162	116117	149307	386289	321162	116117	149307	386289	
22		319757	125375	136200	544936	319757	125375	136200	544936	
25		249426	149562	135713	609473	249426	149562	135713	609473	
31		206415	130851	145220	229720	206415	130851	145220	229720	
37	102735	233581	132643	92070	201672	233581	132643	92070	201672	
43		205376	149055	99331	289489	205376	149055	99331	289489	
49	102735	357861	176114	191366	389047	357861	176114	191366	389047	
55		304489	179580	151199	387586	304489	179580	151199	387586	
61		263191	113543	200164	220405	263191	113543	200164	220405	
67		317448	131662	139075	198916	317448	131662	139075	198916	
73		403323	148471	140780	191992	403323	148471	140780	191992	
79		248703	122163	211371	268997	248703	122163	211371	268997	
85		281928	121460	181720	225720	281928	121460	181720	225720	
91		243172	116682	205754	347517	243172	116682	205754	347517	
97		272052	125150	203347	170534	272052	125150	203347	170534	
100		255595	121030	181174	181174	255595	121030	181174	181174	
1	557	128	125	80	227	7293	5619	5235	13824	time
4		688	214	445	1318	40445	10144	36637	99398	
7		838	358	550	1155	48605	16864	47408	81283	
10		883	301	454	1459	48950	13773	37818	96400	
13		1187	349	826	1894	68752	16589	71411	128730	
16		1552	349	666	1379	91197	16557	58646	91318	
19		1350	429	528	1533	77133	20310	45405	99613	
22		1341	461	493	2192	77261	21648	41139	142176	
25		1043	557	483	2403	57805	26147	38371	153990	
31		867	483	515	899	50371	22474	39619	56829	
37		979	493	326	781	54346	22957	25488	49648	
43		858	554	349	1117	49357	26192	27014	66218	
49	557	1504	653	685	1520	90829	31862	56435	95264	
55	557	1270	669	534	1507	73267	32499	41078	93997	
61		1114	416	714	832	65328	18931	53318	50275	
67		1322	486	493	755	76474	22707	36982	46054	
73		1690	550	502	714	101622	25626	40083	42874	
79		1040	448	752	998	61322	20771	60035	57760	
85		1187	448	653	819	69974	20454	52992	48269	
91		1018	426	736	1251	58704	19488	63571	85213	
97		1136	461	726	605	64259	21338	58509	42483	
100		1139	458	653	659	100781	37693	80829	81168	

0.60 > et < $p1 = 0.75, p2 = 0.25$ >, la majorité des problèmes ne sont pas satisfaisants, par conséquent, *MAC*-2001 est souvent meilleur que *IPDB*. D'une manière générale, utiliser *deg* dans *IPDB* est plus efficace qu'utiliser d'autres *VOHs*.

FIGURE 4.28 – temps d'exécution et nombre de CCs pour $p1 = 0.25$

4.4.5 Expérimentations sur le problème de planification de réunions dynamiques $DMSP$

Dans ce qui suit, nous présentons nos résultats pour la classe de $DMSPs$ $< 15, 25, 4, 5, 10, 1, 80 >$ et modifions le taux de perturbation de $\sim 1\%$ à

FIGURE 4.29 – temps d'exécution et nombre de CCs pour $p1 = 0.75$

$\sim 100\%$. Nous avons généré 30 différents problèmes traités par chaque algorithme et les résultats sont une moyenne de ces 30. Dans les expériences suivantes, la comparaison est seulement entre *MAC-2001* et *IPDB* en utilisant *deg* (la meilleure *VOH* utilisée en réparation, comme indiqué dans les

expérimentations précédentes, et comme déjà prouvé [4]).

Concernant les *MSPs* dynamiques (fig. 4.30), $IPDB_{deg}$ est plus efficace

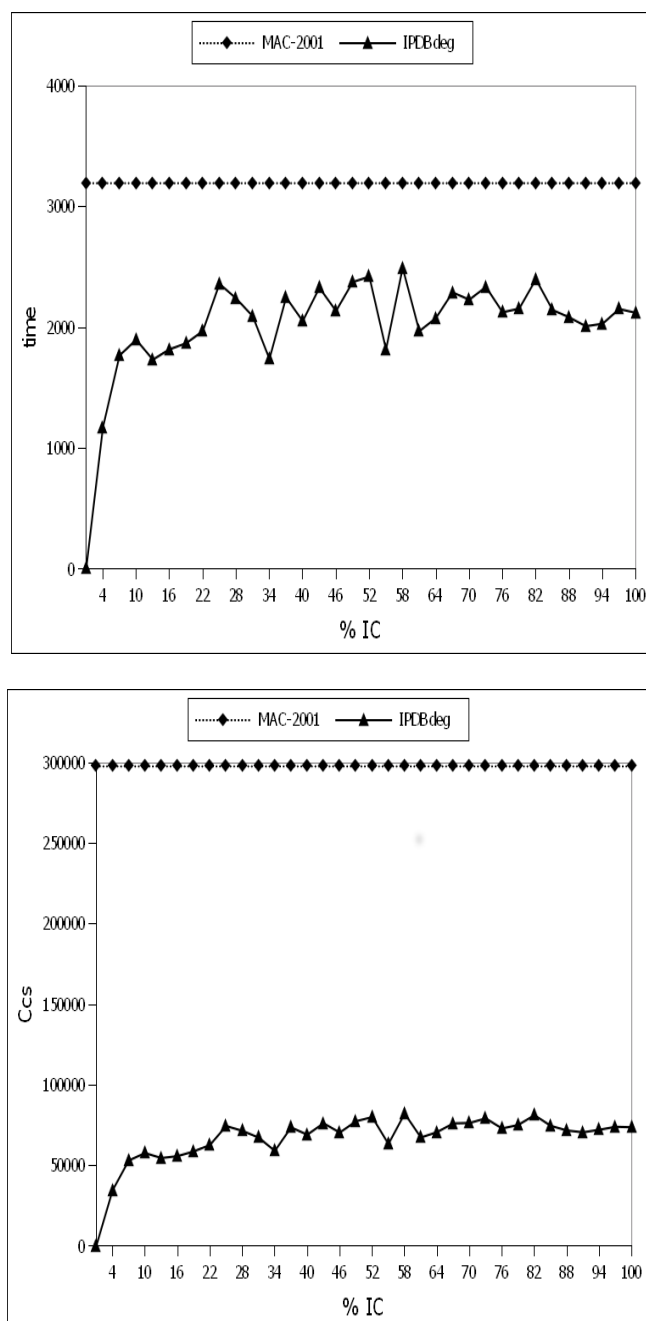


FIGURE 4.30 – Temps d'exécution et nombre de *CCs* pour les *DMSPs*

que *MAC-2001* quel que soit le taux de perturbation, en particulier en ce qui concerne le nombre de *CCs*.

L'utilisation de $IPDB_{deg}$ pour réparer des problèmes qui font objet d'une petite perturbation est suffisamment efficace. Cependant, à partir d'une petite perturbation donnée ($\sim 10\%$), l'effort utilisé par $IPDB_{deg}$ pour réparer des problèmes, chose qui n'est pas stable, est presque le même.

4.4.6 Discussion

Discussion concernant la réparation

Comme le montrent les expérimentations sur les problèmes satisfaisants dans les phases de transition (tabs. 4.11 et 4.12), $IPDB$ est largement plus puissant que $EPDB$ en terme de temps d'exécution. Ceci est particulièrement dû à la façon dont $IPDB$ déduit l'ordre entre les variables en conflit. Il utilise la structure de données supplémentaire, *successors*, pour déduire directement les ordres. Mais en ce qui concerne le nombre de *CCs*, l'utilisation de $IPDB$ avec une *VOH* donnée est la même que l'utilisation de $EPDB$ avec le même *VOH*. En fait, $IPDB$ et $EPDB$ s'exécutent de la même manière. La seule différence est la déduction des ordres, dans $EPDB$, à partir des *nogoods* et *safety conditions*, tandis que $IPDB$ les déduit à partir de *successors*, qui collecte les informations des ordres existantes dans les *nogoods* et *safety conditions*. Ainsi, les deux approches génèrent le même nombre de *CCs*.

À partir d'une petite perturbation, plus ou moins intéressant en fonction de la *VOH* utilisée, $IPDB$ nécessite presque le même effort de réparation, quel que soit le taux de contraintes injectées. Cela est dû aux zones d'appartenance des problèmes choisis, qui sont les zones les plus complexes (les phases de transition). Cet effort n'est pas stable, mais il est soumis à quelques chutes brusques. En fait, l'effort de réparation dépend de l'affectation avant perturbation (l'ancienne solution) et sa consistance avec les nouvelles contraintes. Ainsi, la réparation d'un problème perturbé avec certaines contraintes peut être plus facile ou plus difficile que réparer un autre problème perturbé avec plus de contraintes, selon quelle est l'affectation qui satisfait plus de nouvelles contraintes.

Lorsque les problèmes sont durs et satisfaisants, $IPDB_{deg}$ est un bon concurrent de *MAC-2001*, même pour la résolution (100% de *IC*). Néanmoins, lorsqu'une autre *VOH* est utilisée dans $IPDB$ (2^{me} partie des tabs. 4.11 et 4.12), $IPDB$ devient moins efficace en comparaison avec *MAC-2001*. En fait, l'objectif de *deg* dans la réparation est de maintenir les sous-problèmes les plus connectés (variables avec un grand voisinage) intacts aussi longtemps que possible, afin de minimiser les risques d'affecter les voisins, en cas de vidages des domaines.

Discussion concernant la résolution

En ce qui concerne les problèmes où la dureté des contraintes est inférieure au pic (en dessous de la phase de transition), bien que l'injection de la totalité des contraintes en perturbation, c-à-d la résolution (fig. 4.28 et 4.29), *IPDB* est plus efficace que *MAC-2001*, quelle que soit la *VOH* utilisée et la densité des problèmes. Dans ces domaines, la plupart des problèmes sont satisfaisants, donc l'utilisation de la cohérence de l'arc n'est pas suffisamment efficace. En fait, l'intérêt majeur de *MAC* est de détecter plus tôt les futurs vidages de domaines, afin de modifier les affectations partielles qui ne peuvent pas faire partie d'une solution. Cependant, au dessous des pics, les vidages des domaines sont rarement rencontrés.

Par conséquent, pour des problèmes insatisfaisants, où la dureté des contraintes est supérieure au pic, *MAC-2001* est plus efficace que *IPDB*.

Dans les pics, la majorité des problèmes sont insatisfaisants, alors, *MAC-2001* est globalement plus efficace que *IPDB*.

Comme pour les expérimentations concernant la réparation, lorsque *deg* est utilisé, *IPDB* a un meilleur comportement, surtout pour les problèmes de faible densité. Pour les densités élevées, le réseau est très connecté, donc les variables ont des valeurs de *deg* assez proches.

4.5 Lazy Repairing Backtracking

4.5.1 Motivation

EPDB a l'avantage d'être largement flexible en comparaison avec *PDB* : il exploite la flexibilité de *PDB* pour construire un ordre optimal entre les variables, en utilisant les heuristiques de choix de variables (*VOHs*), afin de converger vers la terminaison avec un minimum de tests de contraintes.

l'inconvénient des deux approches est la vérification de l'ordre actuel entre les variables concernées avant la génération d'un *nogood*, dans la situation de conflit ou de *dead-end*. Cette vérification d'ordres, qui sont stockés dans l'ensemble de *safety conditions* et de *nogoods*, est nécessaire pour éviter de boucler, mais consomme généralement un temps de recherche considérable, puisque dans la plupart des cas les ordres ne sont pas directement déduits.

Pour mieux expliquer cet inconvénient, supposons l'exemple suivant :

EPDB est utilisé pour réparer un *DCSP*. La contrainte actuelle à tester est C_{12} . Cette contrainte est violée, d'où la nécessité de générer un *nogood* entre X_1 et X_2 , ainsi, l'ordre courant entre X_1 et X_2 est vérifié.

La table 4.13 montre l'ensemble des ordres courants existants :

En fait, $X_2 < X_1$ puisque $X_2 < X_i$ (1), $X_i < X_j$ (6), $X_j < X_k$ (3) et

TABLE 4.13 – ordres existants

<i>safety conditions</i> :	<i>nogoods</i> :
$X_2 < X_i$ (1)	$X_k = v_k \implies X_l \neq v_l$ (4)
$X_i < X_m$ (2)	$X_k = v_k \implies X_1 \neq v_1$ (5)
$X_j < X_k$ (3)	$X_i = v_i \implies X_j \neq v_j$ (6)

$X_k < X_1$ (5). Mais l'ordre entre X_1 et X_2 n'est pas directement déduit, puisque les variables inclus dans cet ordre, à savoir X_2 , X_i , X_j , X_k et X_1 , sont incluses dans d'autres *nogoods* et *safety conditions* qui sont stockés avant les ordres concernés : X_i est incluse dans (2) qui est placé avant (6) et X_k dans (4) qui se trouve avant (5), ainsi, (2) et (4) sont aussi vérifiés.

En outre, déterminer un ordre exige un temps de recherche coûteux même si aucun ordre n'existe entre les variables en conflit. Malheureusement, cela doit être prouvé en vérifiant tous les *nogoods* et *safety conditions* qui les concernent directement (les ordres où ils apparaissent) et indirectement (les ordres où leurs voisins apparaissent directement ou indirectement).

Comme exemple, supposant encore une fois que *EPDB* est utilisé pour réparer un *DCSP*, et que C_{12} , la contrainte courante à tester, est violée. Pour générer un *nogood*, l'ordre courant entre X_1 et X_2 doit être vérifié.

La table 4.14 montre l'ensemble des ordres courants existants : L'approche teste (1) puis (2) pour vérifier si X_2 précède X_1 , et (5) puis (4) afin de vérifier si X_1 précède X_2 , avant de conclure qu'aucun ordre n'existe entre les deux variables, et construire un nouvel ordre en utilisant la *VOH* choisie.

TABLE 4.14 – ordres existants

<i>safety conditions</i> :	<i>nogoods</i> :
$X_2 < X_i$ (1)	$X_k = v_k \implies X_l \neq v_l$ (4)
$X_i < X_m$ (2)	$X_1 = v_1 \implies X_k \neq v_k$ (5)
$X_j < X_n$ (3)	

Pour cette raison, dans *Lazy Repairing Backtracking (LRB)* [3], les ordres sont déduits directement en utilisant la *VOH* choisie. Ainsi, puisque les *safety conditions* dans *EPDB* sont utilisés seulement pour mémoriser les ordres exprimés par les *nogoods* supprimés, et que les *nogoods* sont aussi utilisés pour enregistrer les justifications de rejets (afin d'éviter le choix d'une valeur qui causera un conflit déjà rencontré d'une part, et de restaurer d'autre part les valeurs quand les causes de leur rejet ne sont plus valides), les *safety conditions* sont rejetés contrairement aux *nogoods* qui sont gardés.

Le seul soucis qui peut exister, quand un ordre est établi directement en utilisant une *VOH*, c-à-d sans vérifier l'existence d'ordre entre les variables concernées, est la terminaison quand les valeurs des *VOHs* changent. Pour remédier à ce problème, seuls les *VOHs* statiques sont utilisés, afin d'avoir un ordre fixe entre les variables tout au long du processus de recherche, surtout que les *VOHs* statiques, *deg* et *pdeg* [1] (voir la section suivante : *Profound degree*), ont les meilleurs comportement dans la réparation [1, 4].

4.5.2 LRB comparé à EPDB

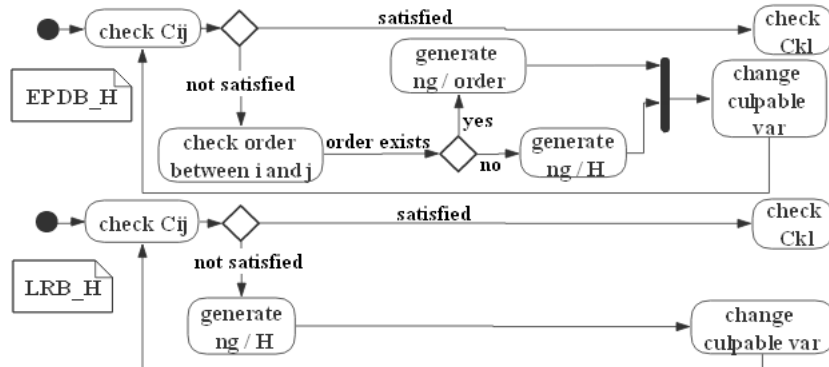


FIGURE 4.31 – Comportement de EPDB et LRB

Comme montré dans la figure 4.31, après le test d'une contrainte C_{ij} , *EPDB* ne teste une autre contrainte C_{kl} que si C_{ij} est satisfaite. Si C_{ij} est violée, il teste si un ordre existe déjà entre les variables concernées i et j . Par la suite, il génère un *nogood* ng , qui respecte l'ordre s'il existe. Si aucun ordre n'existe entre i et j , le *nogood* respecte la *VOH* choisie H . Ainsi, la

conclusion de ng change de valeur et C_{ij} est re-testée.

LRB , comme $EPDB$, après le test de la contrainte C_{ij} , teste une autre contrainte C_{kl} seulement si C_{ij} est satisfaite. Sinon, LRB génère un *nogood* ng , en respectant l'heuristique choisie H . Ainsi, la conclusion de ng change de valeur et C_{ij} est testée à nouveau.

4.5.3 Exemple d'exécution

Considérons le CSP illustré dans la figure 4.32 :

- $X = \{X1, X2, X3, X4\}$.
- $D = \{D_1, D_2, D_3, D_4\}$ such as $D_i = \{1, 2, 3\} \forall i \in [1, 4]$.
- $C = \{X1 < X2, X1 \neq X4, X2 = X3, X2 \neq X4\}$.

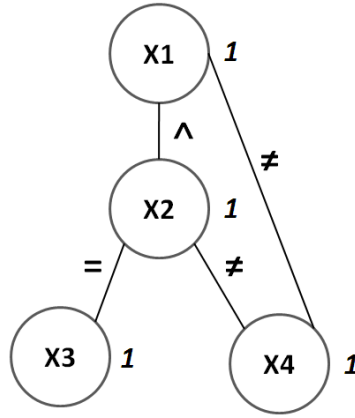


FIGURE 4.32 – réseau de contraintes

Toutes les variables sont affectées par la valeur 1. En raison de simplification, nous considérons que la première affectation (avant la perturbation) a été trouvée en utilisant une simple approche de résolution (FC par exemple), ainsi, au début de la réparation, aucun *nogood* ou *safety condition* ne sont enregistrés suite à la recherche juste avant. Thus, no order is imposed between variables.

Le tableau 4.15 décrit $EPDB$ utilisant l'heuristique deg . L'approche calcule, avant de commencer la recherche, deg de chaque variable dans le réseau : $deg(X1) = 2$, $deg(X2) = 3$, $deg(X3) = 1$ et $deg(X4) = 2$.

Toutes les variables ont la même valeur affectée 1. Cette affectation viole plus qu'une contrainte. Une heuristique d'ordonnancement de contraintes [37] peut être utilisée pour améliorer la recherche, mais pour traiter différents cas, nous supposons que les contraintes sont ordonnées comme indiqué dans la colonne *to check*.

La première contrainte vérifiée *checked* est celle entre $X1$ et $X2$ (ligne 1). Cette contrainte est violée et, comme aucun ordre n'existe encore entre $X1$

et $X2$ (la colonne *add* est vide), la valeur de l'heuristique de $X1$ et $X2$ est vérifiée pour déterminer l'ordre. $deg(X1) < deg(X2)$, donc le nogood généré est (1). $X1$ change de valeur et la contrainte est vérifiée avant de sélectionner une autre (ligne 2). C_{12} n'est encore pas consistante, alors l'existence d'un ordre entre $X1$ et $X2$ est vérifié avant de générer un nogood. (1) est testé alors (2) est généré. $X1$ change la valeur et la contrainte, qui n'est toujours pas consistante, est ré-vérifiée (ligne 3). Pour générer le nogood, l'existence d'un ordre entre $X1$ et $X2$ est revérifié. Ainsi, (1) est testé et (3) est généré. $X1$ n'a plus de valeurs dans son domaine, alors *EPDB* résout (1), (2) et (3) comme (4), supprime les trois nogoods non valides (ligne 4), enregistre leur ordre exprimé comme *safety condition* (5) et change la valeur de $X2$, la cause unique du vidage de domaine de $X1$ (ligne 5). $X1$ est affecté par sa première valeur valide, et la contrainte, qui est consistante, est revérifiée (ligne 6). Comme C_{12} est satisfaite, elle est déplacée vers la liste *free cstrs*, qui contient toutes les contraintes testées et satisfaites, et la prochaine contrainte à vérifier est testée. C_{14} est violée (ligne 7). L'existence d'un ordre entre $X1$ et $X4$ est vérifiée, mais aucun ordre dans *add* ne contient $X4$. Ensuite, la valeur de l'heuristique est vérifiée. Comme $deg(X1) = deg(X4)$, n'importe quel ordre est choisi, nous supposons que dans des cas similaires, l'ordre choisi est l'ordre lexicographique. Ainsi, le nogood est généré comme (6). $X4$ change de valeur, alors C_{14} devient consistante (ligne 8) et elle est ainsi déplacée vers *free cstrs*. La prochaine contrainte à vérifier est C_{24} qui est violée (ligne 9), (5) et (6) sont testés, alors (7) est généré, la valeur de $X4$ est changée (ligne 10) et C_{14} , la seule contrainte dans *free cstrs* concernant $X4$, est déplacée vers la liste *to check*. La prochaine contrainte vérifiée est C_{23} qui est violée (ligne 11), les ordres testés sont (5) puis (6), et (7). Comme aucun ordre n'existe entre les deux variables, l'heuristique choisie est utilisée pour générer (8), alors la valeur de $X3$ est modifiée (ligne 12). C_{23} est consistante, donc elle est déplacée vers *free cstrs*. C_{14} est alors vérifiée (ligne 13), elle est satisfaite alors elle est déplacée vers *free cstrs*. *to check* est vide (ligne 14), donc toutes les contraintes sont satisfaites et la solution est trouvée. Le tableau 4.16 décrit *LRB* en utilisant la même heuristique *deg*. L'approche calcule, avant de commencer la recherche, de même que *EPDB*, *deg* de toutes les variables du réseau.

LRB procède comme *EPDB* avec les différences suivantes :

- pour générer un nogood, *LRB* teste directement les valeurs de la *VOH* choisie. Ainsi, dans la colonne *test*, chaque violation de contrainte (nogood à générer) provoque un seul test d'ordre, contrairement à *EPDB* (dans tab. 4.15, la colonne *test* des lignes 9 et 11 contient plus qu'un tester).
- *LRB* n'a pas besoin de l'information sur l'ordre enregistré dans les *safety conditions*, car l'ordre est déterminé par la *VOH* utilisée, alors, lors de la suppression d'un nogood, aucune condition de sécurité n'est conservée (dans tab. 4.16, après la suppression des nogoods dans

TABLE 4.15 – Exécution de $EPDB_{deg}$

	to check	free cstrs	X_1 X_2 X_3 X_4	checked	test	add	remove
1	C_{12} C_{14} C_{24} C_{23}	-	1, 1, 1, 1	$C_{12} \times$	deg	$X_2=1 \Rightarrow X_1 \neq 1$ (1)	
2	C_{12} C_{14} C_{24} C_{23}	-	2, 1, 1, 1	$C_{12} \times$	(1)	$X_2=1 \Rightarrow X_1 \neq 2$ (2)	
3	C_{12} C_{14} C_{24} C_{23}	-	3, 1, 1, 1	$C_{12} \times$	(1)	$X_2=1 \Rightarrow X_1 \neq 3$ (3)	
4	C_{12} C_{14} C_{24} C_{23}	-	- , 1, 1, 1			$X_2 \neq 1$ (4)	(1),(2),(3)
5	C_{12} C_{14} C_{24} C_{23}	-	- , 2, 1, 1			$X_2 < X_1$ (5)	
6	C_{12} C_{14} C_{24} C_{23}	-	1, 2, 1, 1	$C_{12} \checkmark$			
7	C_{14} C_{24} C_{23} C_{12}		1, 2, 1, 1	$C_{14} \times$	deg	$X_1=1 \Rightarrow X_4 \neq 1$ (6)	
8	C_{14} C_{24} C_{23} C_{12}		1, 2, 1, 2	$C_{14} \checkmark$			
9	C_{24} C_{23} C_{12} C_{14}		1, 2, 1, 2	$C_{24} \times$	(5),(6)	$X_2=2 \Rightarrow X_4 \neq 2$ (7)	
10	C_{24} C_{23} C_{14} C_{12}		1, 2, 1, 3	$C_{24} \checkmark$			
11	C_{23} C_{14} C_{12} C_{24}		1, 2, 1, 3	$C_{23} \times$	(5),(6),(7),deg	$X_2=2 \Rightarrow X_3 \neq 1$ (8)	
12	C_{23} C_{14} C_{12} C_{24}		1, 2, 2, 3	$C_{23} \checkmark$			
13	C_{14} C_{12} C_{24} C_{23}		1, 2, 2, 3	$C_{14} \checkmark$			
14	- C_{12} C_{24} C_{23} C_{14}		1, 2, 2, 3	NULL			

la colonne *remove* de la ligne 4, la colonne *add* de la ligne 5 est vide).

TABLE 4.16 – Exécution de LRB_{deg}

	to check	free cstrs	X_1 X_2 X_3 X_4	checked	test	add	remove
1	C_{12} C_{14} C_{24} C_{23}	-	1, 1, 1, 1	$C_{12} \times$	deg	$X_2=1 \Rightarrow X_1 \neq 1$ (1)	
2	C_{12} C_{14} C_{24} C_{23}	-	2, 1, 1, 1	$C_{12} \times$	deg	$X_2=1 \Rightarrow X_1 \neq 2$ (2)	
3	C_{12} C_{14} C_{24} C_{23}	-	3, 1, 1, 1	$C_{12} \times$	deg	$X_2=1 \Rightarrow X_1 \neq 3$ (3)	
4	C_{12} C_{14} C_{24} C_{23}	-	- , 1, 1, 1			$X_2 \neq 1$ (4)	(1),(2),(3)
5	C_{12} C_{14} C_{24} C_{23}	-	- , 2, 1, 1				
6	C_{12} C_{14} C_{24} C_{23}	-	1, 2, 1, 1	$C_{12} \checkmark$			
7	C_{14} C_{24} C_{23} C_{12}		1, 2, 1, 1	$C_{14} \times$	deg	$X_1=1 \Rightarrow X_4 \neq 1$ (5)	
8	C_{14} C_{24} C_{23} C_{12}		1, 2, 1, 2	$C_{14} \checkmark$			
9	C_{24} C_{23} C_{12} C_{14}		1, 2, 1, 2	$C_{24} \times$	deg	$X_2=2 \Rightarrow X_4 \neq 2$ (6)	
10	C_{24} C_{23} C_{14} C_{12}		1, 2, 1, 3	$C_{24} \checkmark$			
11	C_{23} C_{14} C_{12} C_{24}		1, 2, 1, 3	$C_{23} \times$	deg	$X_2=2 \Rightarrow X_3 \neq 1$ (7)	
12	C_{23} C_{14} C_{12} C_{24}		1, 2, 2, 3	$C_{23} \checkmark$			
13	C_{14} C_{12} C_{24} C_{23}		1, 2, 2, 3	$C_{14} \checkmark$			
14	- C_{12} C_{24} C_{23} C_{14}		1, 2, 2, 3	NULL			

4.5.4 Description de l'algorithme

Pour réutiliser la solution et les techniques de raisonnement passées, LRB commence la recherche avec un $DCSP$ qui contient la dernière solution, tout en gardant la structure de *nogoods ngList* enregistrée au cours du dernier processus de recherche.

LRB se déroule exactement comme $EPDB$, avec les différences principales suivantes :

- Pour générer un *nogood* lorsqu'un conflit est détecté, LRB ne vérifie pas l'ordre entre les variables concernées comme le fait $EPDB$. Ainsi, dans la ligne 9 de la figure 4.4, la VOH choisie est directement utilisée pour établir l'ordre.

```

Procedure LRB(DCSP, newCstrs, SVOH)
1.  DCSP.cstrs.update(newCstrs);
2.  solutionFound  $\leftarrow$  False;
3.  emptyNgGenerated  $\leftarrow$  False;
4.   $c \leftarrow$  getACstr(DCSP.cstrs);
5.  while ( $\neg$ solutionFound  $\&\&$   $\neg$ emptyNgGenerated) do
6.    if ( $\neg$ isConsistant(c)) then
7.      conflictVars  $\leftarrow$  getListOfVars(c);
8.      ng  $\leftarrow$  GenerateNgWithHeuristic(conflictVars, SVOH);
9.      DCSP.ngList.add(ng);
10.     Repair(Rhs(ng), SVOH);
11.   else
12.     move(c, DCSP.cstrs, DCSP.freeCstrs);
13.      $c \leftarrow$  getACstr(DCSP.cstrs);
14.     if (isNULL(c)) then
15.       solutionFound  $\leftarrow$  true;
16.   DCSP.ReturnResult();

Function Repair(var, SVOH)
17. if (var.Domain.isEmpty()) then
18.   ng2  $\leftarrow$  ResolutionNg(var, SVOH);
19.   if (ng2.isEmpty()) then
20.     emptyNgGenerated  $\leftarrow$  True;
21.     Return False;
22.   if ( $\neg$ Repair(Rhs(ng2), SVOH)) then
23.     Return False;
24.   DCSP.ngList.removeWithLhs(var);
25.   moveWith(var, DCSP.freeCstrs, DCSP.cstrs);
26.   var.ChangeValue();
27.   Return True;

Function ResolutionNg(var, SVOH)
28. listVar = CSP.NgList.getAllVarLhs(var);
29. GenerateNgWithHeuristic(listVar, SVOH);

```

FIGURE 4.33 – Description de *LRB*

- De même, pour générer le *nogood* de résolution lorsqu'un vidage de domaine est produit (situation de *dead-end*), *LRB* ne vérifie pas l'ordre entre les variables responsables de ce *dead-end* contrairement à *EPDB*. Ainsi, dans la ligne 35 de la figure 4.4, la *VOH* choisie est directement utilisée pour générer le *nogood*.
- Puisqu'aucune vérification d'ordre n'est requise, *LRB* n'utilise pas la liste de *safety conditions* pour stocker les ordres exprimés par les *nogoods* supprimés. Ainsi, les lignes 2 et 26 de la figure 4.4 sont supprimés.

4.5.5 preuve de terminaison

Nous avons prouvé que *EPDB* termine.

LRB procède de la même manière que *EPDB*, mais sans utiliser les *safety conditions* qui servent à mémoriser les anciens ordres exprimés par les *nogoods* supprimés. En fait, *LRB* n'a pas besoin de *safety conditions* puisqu'il établit l'ordre directement depuis la *VOH* utilisée. Pour ne pas boucler et garantir ainsi la terminaison, il utilise uniquement les *VOHs* statiques.

4.5.6 Étude expérimentale

Expérimentations concernant la réparation

Dans cette section, nous évaluons *LRB* par rapport à *EPDB*, pour montrer la performance de *LRB* par rapport à *EPDB*. Nous comparons également avec *MAC-2001* [17, 16], pour identifier le taux maximal de perturbation qui tolère la réparation, c-à-d à partir de quel taux de perturbation l'utilisation d'un bon algorithme de résolution est plus efficace pour traiter *DCSP*.

MAC-2001 est combiné à la l'heuristique *dom/deg* comme dans [48], tandis que les heuristiques utilisées en *LRB* et *EPDB* sont *pdeg* et *deg*. Pour prouver que la performance de *LRB* est principalement due à une bonne *VOH*, nous utilisons également *lexico*, l'heuristique d'ordre lexicographique, qui est utilisée à la place de l'ordre arbitraire, car *LRB* supporte uniquement les *VOHs* statiques pour éviter de boucler. En tant que critères d'évaluation, nous mesurons le temps d'exécution et le nombre de tests de contraintes (*CCs*).

Expérimentations sur les *DCSPs* binaires aléatoires uniformes :

Les tests suivants sont effectués sur des *DCSPs* binaires aléatoires uniformes à densité faible et élevée, respectivement $< 30, 10, 0.25, 0.45 >$ et $< 30, 10, 0.75, 0.19 >$. Les problèmes choisis sont des problèmes satisfaisants appartenant à la phase de transition [23] (la zone la plus compliquée).

Pour chaque paire $< p1, p2 >$, 30 différentes instances sont traitées par chaque algorithme et les résultats sont les moyennes de ces 30 exécutions.

Le taux de contraintes injectées *IC* varie de 1% à 100%, tel que pour chaque perturbation donnée de *d*%, les contraintes ajoutées sont identiques à celles de la perturbation de *d* − 1%, Plus 1% de contraintes supplémentaires. Les 30 problèmes finaux (après la perturbation) sont les mêmes pour chaque taux de perturbation.

Pour les approches de réparation de *DCSPs*, à savoir *LRB* et *EPDB*, l'effort calculé est celui de la réparation de la solution passée, c-à-d qui existait avant la perturbation.

Comme *p2* est connu, nous avons essayé de remplacer, dans le calcul de *pdeg*, le paramètre $1/2$ par *p2*. Mais puisque les résultats n'étaient pas signifi-

tivement meilleurs par rapport à ceux où le paramètre $1/2$ est utilisé, nous choisissons de garder ce paramètre pour toutes sortes de problèmes. Le délai d'attente maximal est fixé à 7 minutes. Les expériences sont effectuées sur la plate-forme *Jav*, sur un *PC i7* (d'un processeur de 3.6 GHz et de 16 Go de RAM). En ce qui concerne le nombre de *CCs* (2^{me} partie des tabs. 4.17 et 4.18), utiliser *LRB* avec une *VOH* donnée est presque identique à utiliser *EPDB* avec la même *VOH*. Mais concernant le temps d'exécution (1^{re} partie des tabs. 4.17 et 4.18), *LRB* dépasse *EPDB* très remarquablement, quelle que soit la *VOH* utilisée.

À l'exception des petites perturbations, *MAC-2001* génère souvent moins de *CCs* que *LRB* (2^{me} partie des tabs. 4.17 et 4.18), principalement lorsque *LRB* Utilise l'heuristique *deg*.

À partir d'une petite perturbation (13% pour les problèmes peux denses et 10% pour ceux trop denses), *LRB* nécessite presque le même effort de réparation (tabs. 4.17 et 4.18), quel que soit Le taux de contraintes injectées *IC*. Cet effort n'est pas stable, indépendamment de l'heuristique utilisée, mais il est soumis à des chutes brusques.

En ce qui concerne *LRB*, l'utilisation de *pdeg* est souvent plus bénéfique que d'utiliser *deg*. *LRB_{pdeg}* est toujours plus rapide que *MAC-2001* (1^{re} partie des tabs. 4.17 et 4.18), même en perturbant 100% de contraintes, ce qui est équivalent à résoudre les problèmes entiers. *LRB_{deg}* est parfois moins efficace que *MAC-2001*, mais il reste un bon concurrent.

Expérimentations sur le problème de planification de réunions dynamiques *MSP* : Nous présentons nos résultats pour la classe de *DMSPs* $< 15, 30, 4, 5, 10, 1, 85 >$ et modifions le taux de perturbation de $\sim 1\%$ à $\sim 100\%$. Nous avons généré 30 différents cas traités par chaque algorithme et les résultats sont la moyenne de ces 30 exécutions. Comme *LRB_{lex}* et *EPDB* ne sont pas de bons concurrents, comme indiqué dans les expériences ci-dessus, nous incluons dans notre comparaison uniquement *MAC-2001*, *LRB_{pdeg}* et *LRB_{deg}*.

Concernant les *DMSPs* (fig. 4.34), *LRB_{pdeg}* est plus efficace que *MAC-2001* quel que soit le taux de perturbation, en particulier en ce qui concerne le nombre de *CCs*. Cependant, *MAC-2001* est plus efficace que *LRB_{deg}* par rapport au temps d'exécution (à l'exception d'une perturbation de 1%), même si le taux de *CCs* est comparable à celui de *LRB*.

Expérimentations concernant la résolution

Les expérimentations précédentes concernent des problèmes satisfaisants. Les expériences sur les *DCSPs* aléatoires sont plus précisément concentrés sur des problèmes satisfaisants dans des zones difficiles de densités faibles et élevées. Ces expériences montrent que *LRB* en utilisant une *VOH*, princi-

TABLE 4.17 – Temps d'exécution en secondes et nombre de *CCs* ($p1 = 0.25$, $p2 = 0.45$)

% IC	MAC-2001	LRB_{pdeg}	LRB_{deg}	LRB_{lex}	$EPDB_{pdeg}$	$EPDB_{deg}$	
1	0.094	0.006	0.008	1.471	0.508	0.723	time
4		0.024	0.026	1.835	2.242	2.662	
7		0.048	0.065	2.047	4.842	6.254	
10		0.059	0.073	2.304	5.788	7.047	
13		0.079	0.089	2.650	7.517	8.373	
16		0.082	0.093	2.860	8.081	9.338	
19		0.082	0.097	2.809	8.099	9.243	
22		0.085	0.094	2.876	8.214	9.479	
25		0.080	0.095	2.878	7.759	9.479	
28		0.080	0.099	2.860	8.020	9.467	
31		0.078	0.101	2.891	7.263	8.093	
34		0.075	0.102	2.881	7.330	8.402	
40		0.062	0.075	2.869	8.321	8.069	
46		0.069	0.078	2.893	9.521	10.455	
52		0.080	0.093	2.894	8.919	10.799	
58		0.065	0.076	2.914	8.897	8.989	
64		0.075	0.084	2.942	8.593	8.258	
70		0.082	0.097	2.932	7.678	9.113	
76		0.073	0.106	2.871	8.189	10.227	
82		0.085	0.099	2.903	8.266	10.381	
88		0.083	0.100	2.889	8.203	10.337	
94		0.085	0.099	2.897	8.510	10.315	
100		0.084	0.099	2.868	8.230	9.888	
1	169168	15108	20199	3032339	15108	20199	CCs
4		58651	65202	3806153	59779	68687	
7		117728	157599	4227876	123667	157298	
10		145623	181242	4764235	148286	180593	
13		192355	214418	5527584	189534	209805	
16		201644	225471	5992181	205562	230519	
19		201099	235586	5879701	204280	228941	
22		207509	230595	6049455	206419	233566	
25		196193	233567	6047661	197054	235723	
28		196130	243182	6001166	203289	237828	
31		191143	247404	6060702	185955	206190	
34		183019	248225	6058750	188308	211733	
40		151765	185843	6015178	212178	202816	
46		170804	192706	6066024	242173	261430	
52		169168	197833	6076485	228440	268696	
58		169168	161864	187689	6094090	227370	
64		186754	207362	6138907	223674	210593	
70		200774	237995	6029206	195718	229670	
76		181101	255407	6029857	213139	257020	
82		169168	207836	6084857	210158	259925	
88		201640	248743	6070954	210870	259146	
94		207763	243764	6068184	217752	257468	
100		206479	241355	6014275	207955	246269	

TABLE 4.18 – Temps d'exécution en secondes et nombre de *CCs* ($p1 = 0.75$, $p2 = 0.19$)

% IC	MAC-2001	LRB_{pdeg}	LRB_{deg}	LRB_{lex}	$EPDB_{pdeg}$	$EPDB_{deg}$	
1	11.054	0.698	0.896	1.874	111.769	194.669	time
4		3.328	4.565	13.510	370.385	-	
7		4.698	5.877	32.571	-	-	
10		6.590	7.427	40.478	-	-	
13		7.379	6.675	42.993	-	-	
16		7.657	7.591	44.719	-	-	
19		7.847	8.638	44.873	-	-	
22		8.835	9.253	44.903	-	-	
25		5.962	8.694	44.027	-	-	
28		7.711	9.874	44.113	-	-	
31		7.848	9.780	44.086	-	-	
34		8.938	10.503	44.004	-	-	
40		8.357	9.736	43.991	-	-	
46		8.183	9.297	44.067	-	-	
52		10.209	9.597	44.148	-	-	
58		9.427	10.268	43.885	-	-	
64		10.794	10.731	43.960	-	-	
70		10.688	11.700	43.985	-	-	
76		8.355	9.925	44.026	-	-	
82		8.946	10.461	44.056	-	-	
88		8.271	9.475	44.145	-	-	
94		7.615	8.883	44.184	-	-	
100		7.600	8.898	44.242	-	-	
1	11091737	2053499	2676894	5300103	2063315	2687746	CCs
4	11091737	10050253	13609155	37787281	7700192	-	
7		14196226	17492480	90474816	-	-	
10		19917644	22168200	112775270	-	-	
13		22323569	19950505	119366418	-	-	
16		23178909	22661403	121865776	-	-	
19		23692559	25854188	122555689	-	-	
22		26684225	27586331	122734624	-	-	
25		18061593	25937447	122755174	-	-	
28		23355364	29466473	122739959	-	-	
31		23793094	29203775	122837687	-	-	
34		27114111	31384172	122669346	-	-	
40		25268174	29127000	122734902	-	-	
46		24710542	27748892	122643912	-	-	
52		30820375	28720750	122943597	-	-	
58		28428460	30704619	122704450	-	-	
64		32524541	32018965	122798424	-	-	
70		32150097	34963393	122669749	-	-	
76		25192875	29655806	122768330	-	-	
82		26922873	31354768	122696907	-	-	
88		24982344	28307565	122684444	-	-	
94		22954833	26581585	122766266	-	-	
100		22965095	26579493	122848545	-	-	

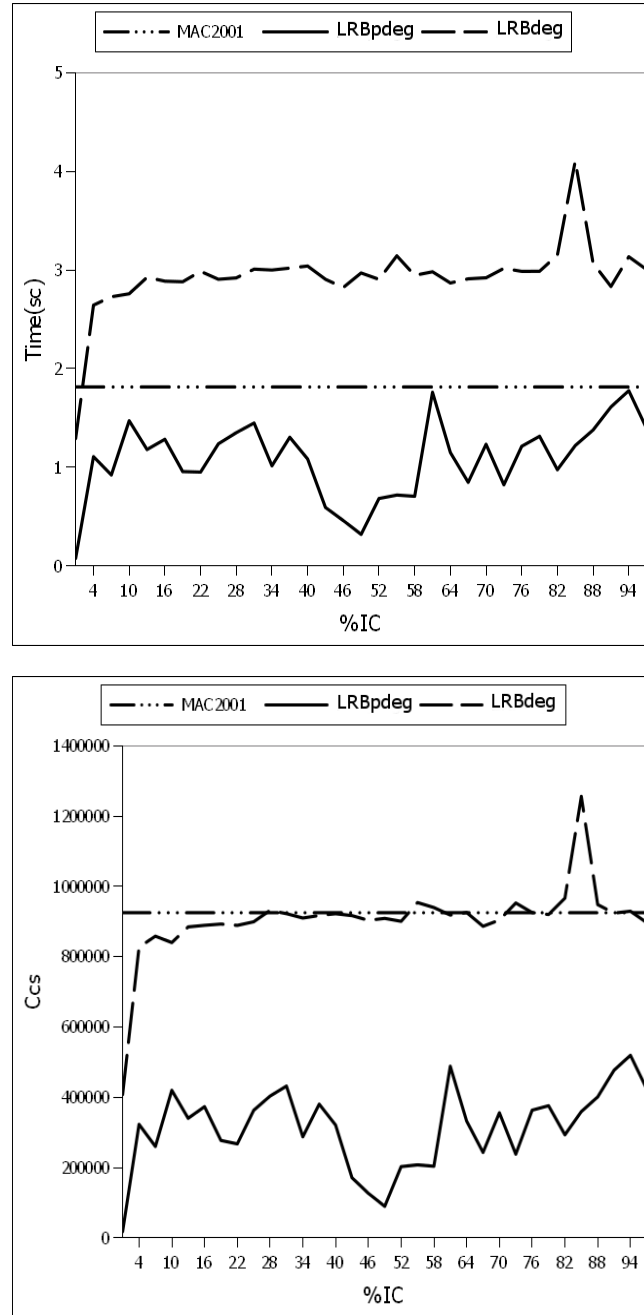


FIGURE 4.34 – Temps d'exécution en secondes et nombre de CCs pour les $MSPs$

palement $pdeg$, est un bon concurrent pour $MAC-2001$ même pour résoudre (100% de IC).

Afin de montrer où $MAC-2001$ est plus efficace que LRB_{pdeg} et LRB_{deg} ,

nous exécutons les trois approches sur des problèmes aléatoires, pas nécessairement satisfaisants, avec les mêmes paramètres précédents n , d et $p1$. Les duretés de contraintes varient de $p2 = 0,1$ à $p2 = 0,9$. 30 différents problèmes sont traités par chaque algorithme et les résultats sont les moyennes de ces 30 exécutions. Les expérimentations sont effectuées sur la plate-forme *Java*, sur un *PC Pentium IV* (d'un processeur de 2.8 GHz et de 3.24 Go de RAM).

Les figures 4.35 et 4.36 montrent que *LRB*, en utilisant *deg* ou *pdeg*, est efficace pour traiter les problèmes satisfaisants, c-à-d $< p1 = 0.25, p2 \in [0.1, 0.45[>$ et $< p1 = 0.75, p2 \in [0.1, 0.19[>$, alors que *MAC-2001* est efficace pour détecter la non-satisfaction des problèmes dans $< p1 = 0, 25, p2 \in]0, 45, 0, 9] >$ et $< p1 = 0, 75, p2 \in]0, 19, 0, 9] >$. Dans les zones complexes, c-à-d les phases de transition $< p1 = 0.25, p2 = 0.45 >$ et $< p1 = 0.75, p2 = 0.19 >$, la majorité de problèmes sont non satisfaisants, par conséquent, *MAC-2001* est généralement plus performant que *LRB*, principalement pour les densités élevées, où des problèmes satisfaisants sont rares.

Pour les problèmes de haute densité (fig. 4.36), utiliser *deg* ou *pdeg* dans *LRB* est presque le même.

4.5.7 Discussion

Discussion concernant la réparation

Comme le montrent les expérimentations sur les problèmes satisfaisants dans les phases de transition (tabs. 4.17 et 4.18), *LRB* est largement plus puissant que *EPDB* en terme de temps d'exécution. Ceci est principalement dû à l'évitement de la vérification des ordres progressivement enregistrés, ce qui se fait en *EPDB* chaque fois qu'une inconsistance est détectée. Mais en ce qui concerne le nombre de *CCs*, l'utilisation de *LRB* avec une *VOH* donnée est presque identique à utiliser *EPDB* avec la même *VOH*. Normalement, pour une heuristique fixe, *LRB* et *EPDB* se déroulent de la même manière, avec la différence des vérifications des ordres pour *EPDB*. Ainsi, ils doivent générer le même nombre de *CCs*. Cependant, la perturbation, qui est une modification de contraintes, peut modifier les valeurs des heuristiques utilisées, puisque les deux *VOHs* *pdeg* et *deg* dépendent des contraintes existantes. Ainsi, dans *EPDB*, les *nogoods* stockés avant la perturbation imposent un ordre qui n'est pas nécessaire le même que celui exprimé par les valeurs des heuristiques actuelles, d'où la différence du nombre de *CCs* entre *LRB* et *EPDB*.

A partir d'une certaine petite perturbation, *LRB* nécessite presque le même effort de réparation, quel que soit le taux de contraintes injectées, car les problèmes sont choisis depuis les zones les plus complexes (les phases de transition). Cet effort n'est pas stable, mais il est soumis à des chutes brusques. En fait, l'effort de réparation dépend de l'affectation avant perturbation et

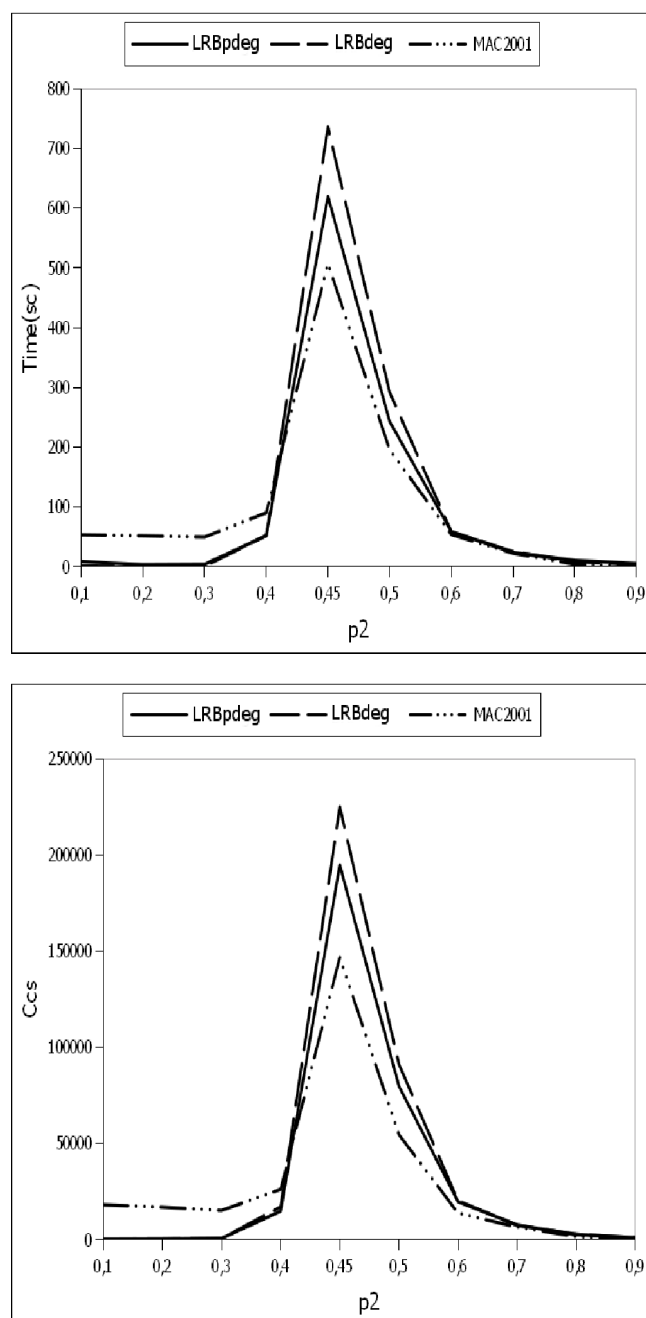


FIGURE 4.35 – Temps d'exécution en secondes et nombre de CCs pour $p1 = 0.25$

de sa consistance avec les nouvelles contraintes. Ainsi, réparer un problème perturbé avec certaines contraintes peut être plus facile ou plus difficile que de réparer un autre problème perturbé avec plus de contraintes, selon quelle

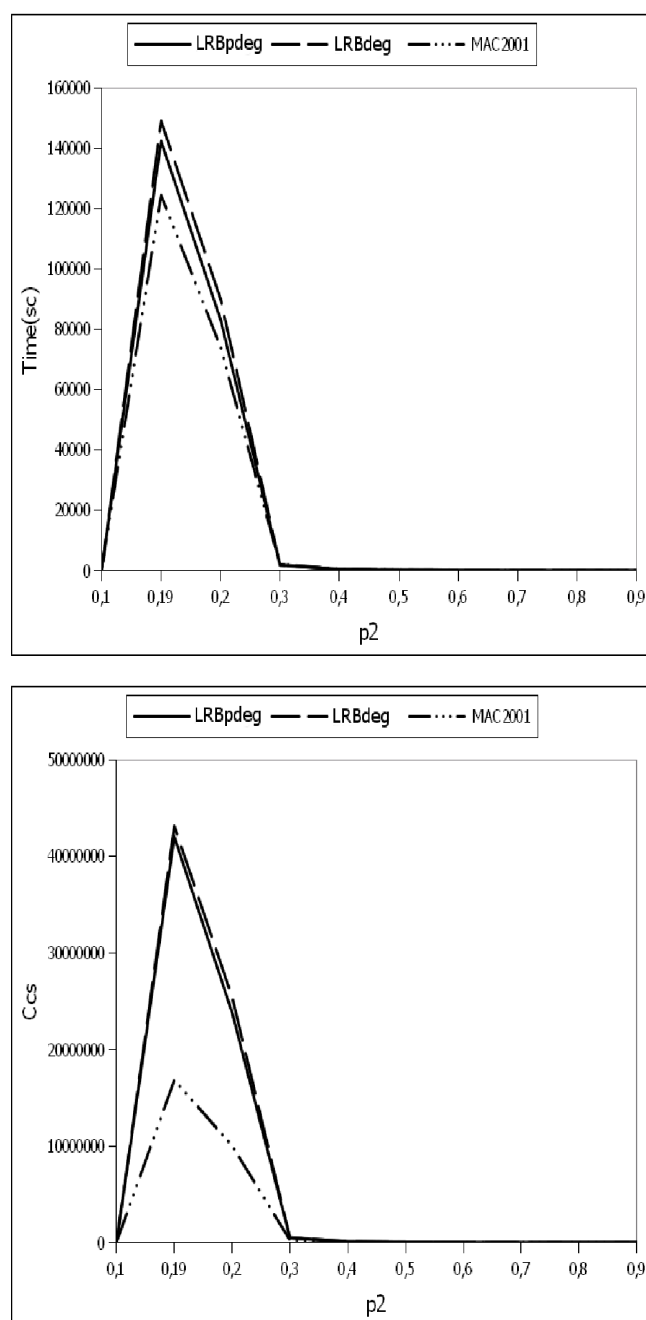


FIGURE 4.36 – Temps d'exécution en secondes et nombre de CCs pour $p1 = 0.75$

est l'affectation qui satisfait le plus de nouvelles contraintes.

L'utilisation de $pdeg$ dans LRB (tabs. 4.17 et 4.18 et fig. 4.34) est souvent plus efficace que l'utilisation de deg , puisque $pdeg$ estime la position de

chaque variable en fonction de l'ensemble du réseau, tandis que *deg* calcule la position de chaque variable concernant son voisinage.

Lorsque les problèmes sont durs et satisfaisants, *LRB*, principalement lors de l'utilisation de *pdeg*, est un bon concurrent de *MAC-2001*, même pour la résolution (100% de *IC*). Néanmoins, quand aucune *VOH* n'est utilisée (*LRB_{lex}* dans la 1^{re} partie des tabs. 4.17 et 4.18), *LRB* devient inefficace comparé à *MAC-2001*. En fait, le but des *VOHs* en réparation est d'engendrer des changements de réseau minimaux, en gardant les sous-problèmes difficiles inchangeables, le plus longtemps possible, afin de minimiser les risques d'affecter d'autres sous-problèmes. Par conséquent, le rendement de *LRB* est également dû à l'utilisation d'une bonne *VOH*.

Discussion concernant la résolution

En ce qui concerne les problèmes où la dureté des contraintes est inférieure au pic (sous la phase de transition), bien que la totalité des contraintes est injectée en perturbation, ce qui est équivalent à la résolution (fig. 4.35 et 4.36), *LRB* en utilisant soit *deg* ou *pdeg* est plus efficace que *MAC-2001*, indépendamment de la densité des problèmes. Dans ces régions, la plupart des problèmes sont satisfaisants, donc l'utilisation de la consistance d'arc n'est pas suffisamment efficace. En fait, l'intérêt majeur de *MAC* est de détecter plus tôt les futurs vidages des domaines, afin de modifier les affectations partielles qui ne peuvent pas être incluses dans une solution, ce qui est rarement rencontrées en dessous du pic.

Par conséquent, pour les problèmes insatisfaisants, où la dureté des contraintes est supérieure au pic, *MAC-2001* est plus efficace que *LRB*.

Dans la phase de transition, la majorité des problèmes sont insatisfaisants, alors *MAC-2001* est souvent plus efficace que *LRB*.

En ce qui concerne les densités élevées, *LRB_{pdeg}* ne dépasse pas remarquablement *LRB_{deg}*. Ceci est dû à la connectivité élevée du réseau. Par conséquent, chaque variable a la majorité des autres variables comme voisins, alors calculer *pdeg* revient presque à calculer *deg*.

Chapitre 5

Les CSPs Dynamiques et Distribués

5.1 Introduction

L'Intelligence Artificielle Distribuée (*DAI* pour *Dynamic Artificial Intelligence*) [30] s'intéresse à l'interaction et à la coordination entre des agents artificiels pour résoudre un problème donné. La plupart des problèmes réels de *DAI* (comme les problèmes de planification [65], les réseaux de capteurs [86], les problèmes d'allocation de ressources [10] et les systèmes d'asservissement [58], [69]) sont dynamiques. Par exemple, lors de la planification d'une réunion distribuée, tout participant doit pouvoir modifier son calendrier privé alors que le processus de recherche de la solution est en train de tourner, de nouvelles réunions devront peut-être être programmées. Ces comportements imprévisibles et stimulants peuvent être liés à plusieurs facteurs :

- Utilisateur : les exigences de l'utilisateur peuvent changer dans un contexte interactif ;
- Environnement : les conditions/caractéristiques de l'environnement peuvent évoluer dans le cadre d'une conception dynamique ;
- Agent : les décisions autonomes de l'agent peuvent changer dans le contexte d'un système distribué ;

Certes, il est possible de résoudre le nouveau problème modifié à partir de zéro. Cette technique ne se rappelle rien du processus de raisonnement précédent et présente certains inconvénients :

- Inefficacité, qui est inacceptable dans le contexte des applications en temps réel (planification, ordonnancement, etc.), où le temps de réorganisation est limité ;
- Instabilité des solutions successives, qui peut être désagréable dans le cadre d'une conception interactive ou d'une activité de planification, si certains travaux ont commencé sur la base de la solution précédente ou s'il est nécessaire de maintenir les allocations de ressources aussi stables que possible, etc.

De nombreuses approches complètes et incomplètes telles que *DynABT* [57], l'intégration de procédures heuristiques [81], *DBA*[47] et *DSA* [85] ont été proposées pour résoudre les *DCSPs* dynamiques. Généralement, la solution trouvée par les approches ci-dessus concernant un *DCSP* dynamique est basée sur la précédente, mais ces approches ne peuvent pas considérer les modifications jusqu'à résoudre l'instance du problème actuel, ainsi, elles ne peuvent pas supporter les perturbations en temps réel. Nous distinguons entre deux scénarios de perturbation :

- Le scénario à faible perturbation : dans lequel le temps entre deux perturbations successives est suffisant pour résoudre un problème, dans ce cas, l'utilisation de la solution précédente semble être simple et naturelle.
- Le scénario rapide : dans ce cas, il n'y a pas assez de temps pour résoudre les problèmes intermédiaires, le problème est perturbé en permanence et les problèmes intermédiaires deviennent rapidement

obsolètes.

Il existe de nombreuses applications récentes pour lesquelles les approches doivent inclure le traitement de dynamisme dans leur processus natif, comme pour le modèle de négociation de nombreux à beaucoup, en particulier pour le problème commercial à haute fréquence, la simulation de la division de cellules biologiques, etc. Ainsi, notre but est de proposer une nouvelle approche pour la réparation des solutions des *DCSPs* dynamiques en traitant leurs changements en temps réel.

5.2 formalisme des CSPs Dynamiques et distribués

Un *CSP* distribué (*DisCSP*) [83] est un problème de satisfaction des contraintes dans lequel les variables et les contraintes sont réparties entre plusieurs agents. Un *DisCSP* peut être décrit comme un quatre-uplet (X, D, C, A, α) , où :

- X, D et C est un *CSP*.
- A est un ensemble d'agents.
- $\alpha : X \rightarrow A$ est une fonction qui lie chaque variable $x \in X$ à son agent $\alpha(x)$.

Nous supposons que :

- Les agents n'ont pas une vision globale du problème pour des raisons de confidentialité et de sécurité
- Un agent ne peut communiquer qu'avec les agents qu'il connaît
- Le retard dans la livraison d'un message est négligeable
- les messages sont reçus dans le même ordre que celui de leurs envois.

La résolution d'un *DisCSP* consiste à trouver une affectation de valeurs à toutes les variables, telles que toutes les contraintes soient satisfaites. Les Problèmes dynamiques de satisfaction de contraintes distribuées (*DDisCSPs*) peuvent être définis comme un six-uplet $(X, D, C, A, \alpha, \delta)$ où :

- X, D, C, A et α est un *DisCSP* ;
- δ est la fonction de changement qui introduit des changements à différents intervalles de temps. C'est une fonction qui représente l'évolution du problème au fil du temps.

Le formalisme des *DDisCSPs* est utilisé pour modéliser des problèmes distribués dans la nature et qui changent au fil du temps. Pour simplifier la description des algorithmes dans ce qui suit, nous supposons que chaque agent du problème gère seulement une variable et toutes les contraintes sont binaires. Il existe plusieurs formes de dynamisme dans un problème *DDisCSP*. Ces changements peuvent être convertis en ajout ou suppression de contraintes. Par exemple, la suppression d'une valeur d'un domaine peut être traduite par l'ajout d'une contrainte unaire à la variable concernée, la révision d'une contrainte existante peut être convertie en une suppression puis un ajout d'une nouvelle contrainte avec les nouveaux paramètres. Ainsi,

nous nous intéressons dans ce travail uniquement à l'ajout et à la suppression des contraintes comme une forme de dynamisme des problèmes *DDisCSPs*.

5.3 Méthodes de réparation de DisCSPs dynamiques

5.3.1 Asynchronous Backtracking

Asynchronous Backtracking (ABT) [84] [83] est l'un des algorithmes références pour la résolution des problèmes *DisCSP*. Il est exécuté de manière systématique, autonome et de manière asynchrone par chaque agent qui prend ses décisions, informe les autres agents. Aucun agent ne doit attendre les décisions des autres. L'algorithme calcule une solution globale cohérente (ou détecte l'insolvabilité) dans un temps fini. La complétude et la terminaison ont été prouvées [83]. *ABT* exige une priorité d'ordre entre les agents et les contraintes sont dirigées entre les agents impliqués : l'agent le plus prioritaire envoie sa décision *value – sending* à l'agent le moins prioritaire qui évalue les contraintes liants ces deux derniers. Lorsqu'un agent plus prioritaire envoie à ses voisins moins prioritaires sa décision (affectation de ses variables), ces derniers tentent de trouver une affectation cohérente avec celle reçue de l'émetteur. Si l'agent est incapable de trouver une affectation cohérente, il retourne en arrière en envoyant à son émetteur un *nogood_{message}*, pour changer son affectation. Les agents tiennent une liste de messages *nogood* qui est utilisée pour guider la recherche. Une solution est trouvée s'il y a un silence dans le réseau, alors que l'insolvabilité est déclarée lorsqu'un *nogood* vide est découvert. *ABT* envoie de nombreux messages obsolètes et utilise un grand espace pour stocker les *nogoods*, ainsi, diverses améliorations (*ABT-famille* [15], *ABT-Dyn* [87] et *Agile-ABT* [72]) ont été proposées pour réduire le nombre de messages obsolètes et l'espace requis pour le stockage des *nogoods*, ou aussi pour supporter l'ordre dynamique des agents. Parce que notre algorithme proposé (présenté dans le chapitre suivant) est une extension de l'approche *ABT* et redéfinit les mêmes fonctions que celles de *ABT*, nous choisissons de présenter et décrire plus formellement l'algorithme *ABT* dans le chapitre suivant.

5.3.2 Dynamic Asynchronous Backtracking

Dynamic Asynchronous Backtracking (DynABT) est un algorithme asynchrone, pour les problèmes *DDisCSPs*, qui se termine et qui est complet [57]. Il est basé sur *ABT*, et conçu pour traiter les problèmes dynamiques en réparant successivement les solutions intermédiaires lorsque les problèmes changent. L'approche traite périodiquement les perturbations survenant après la résolution du problème, à savoir les ajouts et les suppressions de contraintes. *DynABT* utilise la solution du problème initial, et les connaissances que les agents ont appris lors de la recherche de la so-

lution (*AgentViews*, la liste des *nogoods*, etc) et les contraintes enregistrées dans les *nogoods* pour détecter et supprimer les *nogoods* obsolètes lorsqu’une contrainte est enlevée. Lors de la détection d’incohérence, l’agent compose un *nogood* de la forme $x_i = a \cap x_j = b\{C_1, ..C_n\} \Rightarrow x_k \neq c$. La justification($\{C_1, ..C_n\}$) incluse dans le *nogood* indique quelles suppressions de contraintes rendront ce *nogood* obsolète. Les agents dans l’algorithme *DynABT* maintiennent, comme dans *ABT*, une liste d’agents de priorité supérieure avec leurs valeurs au sein de leurs *agentview* et aussi une liste de valeurs qui ne sont pas compatibles avec leurs *agentview* dans leurs *nogood-store*. Les agents les plus prioritaires envoient aux agents moins prioritaires, des messages *info/messages* contenant leurs affectations de valeur. Quand une *info/message* est reçu, l’agent met à jour son *agentview* et contrôle sa cohérence. Chaque agent de *DynABT*, de la même façon que *ABT*, initialise ses variables et commence la recherche pour trouver une solution du problème. Le système intègre périodiquement les changements survenant sur le problème afin de corriger la solution trouvée. Les changements du problème sont traités en deux phases : la phase de propagation (lignes 12-18) et la phase de résolution (ligne 9). Dans la phase de propagation, les agents sont informés par les contraintes à ajouter/supprimer, puis ils mettent à jour leurs listes de contraintes, *neighbours*, *agentViews* et *nogoods* si nécessaire. Les messages *AddConstraint*, *RemoveConstraint* et *AdjustNogood* sont utilisés au cours de cette phase pour gérer le comportement des agents. Après la propagation de tous les changements, le nouveau problème est défini, le booléen *canProceed* est met à vrai, et les agents peuvent ainsi passer à la phase de résolution pour résoudre le nouveau problème. Lorsqu’un agent reçoit un message *addConstraint* (lignes 19-21), il met à jour sa liste de contraintes *constraints list* et sa listes de voisins *neighbours list* si nécessaire. En cas de suppression de contrainte, l’agent en question reçoit un message de type *removeConstraint* (lignes 22-26), l’agent modifie son contexte en supprimant de son *neighbour list* et aussi de son *agent view* l’agent avec qui il partage la contrainte à supprimer et puis il supprime la contrainte de sa *constraints list*. Lorsqu’une contrainte est supprimée, un message de *adjustNogood* (lignes 27-34) est diffusé à tous les agents non impliqués directement dans cette contrainte. Les agents qui reçoivent ce message mettent à jour leurs listes de *nogoods* : ils enlèvent les *nogoods* dont la justification est la contrainte rejetée, et récupèrent les valeurs interdites à cause de la contrainte supprimée. Ainsi, le nouveau problème est dans un point de consistance avant de procéder la phase de résolution.

Procedure DynABT()

```

1.  $changes \leftarrow 0; changeBox \leftarrow empty; canProceed \leftarrow true$ 
2.  $ABT^+(ABT \text{ with nogoods containing justifications})$ 
3. repeat
4.    $changes \leftarrow monitorChanges$ 
5.   if(changes) then
6.      $canProceed \leftarrow false$ 
7.     PropagateChanges(changeBox)
8.     current value  $\leftarrow$  value from the last solution
9.      $ABT^+()$ 
10.  end if
11. until termination condition met

```

Procedure PropagateChanges(changeBox)

```

12. while  $changeBox \neq empty \cap canProceed \leftarrow false$  do
13.    $con \leftarrow getChange; changeBox \leftarrow changeBox - con$ 
14.   Switch(con.msgType)
15.    $con.removeConstraint : removeConstraint(con);$ 
16.    $con.addConstraint : includeConstraint(con);$ 
17.    $con.adjustNogood : incoherentConstraint(con);$ 
18. end while

```

Procedure IncludeConstraint(con)

```

19.  $newCons \leftarrow con.getConstraint()$ 
20. add new neighbours in  $newCons$  to neighbour list
21.  $constraintList \leftarrow constraintList \cup newCons$ 

```

Procedure ExcludeConstraint(con)

```

22.  $incoherentConstraint(con)$ 
23.  $constraint \leftarrow con.getConstraint()$ 
24. Remove unique neighbours in constraint
    from neighbour list
25. Delete unique neighbours from agentView
26. Remove constraint from constraintList

```

Procedure AdjustNogoods()

```

27.  $IncoherentConstraint(con)$ 
28.  $constraint \leftarrow con.getConstraint()$ 
29. for each nogood in nogoodStore do
30.   if  $contains(nogood, constraint)$  then
31.     return eliminated value in nogood to domain
32.   remove nogood from nogoodStore
33.   end if
34. end for

```

FIGURE 5.1 – Pseudo-code de l'algorithme *DynABT* [57]

Chapitre 6

Contribution aux DDisCSPs

6.1 description du LiveABT

LiveABT [9] est une nouvelle variante de l'algorithme *ABT* pour traiter les problèmes *DDisCSPs*, cette variante supporte nativement trois types de messages supplémentaires de la même manière que *DynABT* : *AddConstraint*, *RemoveConstraint* et *AdjustNogood*. Cependant, *LiveABT* les gère d'une manière différente que celle de *DynABT*, dans le but de tenir en compte les perturbations en temps réel.

Comme décrit dans le chapitre précédent, *ABT* est un algorithme distribué, c-à-d que les variables sont réparties entre plusieurs agents autonomes : chaque agent s'exécute lui-même, en gérant seulement le contexte où il est impliqué. Par conséquent, pour supporter les perturbations, il n'est pas nécessaire lors de l'injection d'une contrainte d'arrêter l'ensemble du système multi-agents pour modifier partiellement une partie du réseau, comme le fait *DynABT*, puisque la perturbation n'impacte pas directement tous les agents.

Ainsi, puisque toutes les communications entre les agents du réseau sont assurées par des messages, tel que chaque agent réagit convenablement lors de la réception d'un message (par exemple, la réception d'un message *Ok* est systématiquement suivie d'une mise à jour de *AgentView* de l'agent récepteur, et la réception d'un message *nogood* est traitée par le changement de valeur si on n'est pas dans le cas du domaine vide), gérer les perturbations dans *ABT* peut être simplement intégré dans son protocole de communication.

Par conséquent, pour incorporer les messages *AddConstraint* ou *RemoveConstraint*, le traitement peut être fait dans la phase de résolution exactement comme pour n'importe quel autre message.

Ainsi, dans *LiveABT*, la phase de propagation n'est pas requise. Une injection de perturbation est reflétée par le traitement immédiat d'un message envoyé à partir du système, qui est généralement le *master*, ce traitement se compose de deux parties : Traiter la perturbation au niveau local et la propager dans tout le réseau.

6.2 LiveABT comparé à DynABT

Comme décrit dans fig. 6.1, *DynABT* est exécuté en deux grandes phases, la phase de propagation pour la mise à jour du problème et la phase de résolution pour résoudre le problème. Pratiquement, lors de l'injection d'une perturbation à un moment t , *DynABT* stocke la perturbation pour un traitement futur et continue la résolution du problème. Après avoir résolu le problème initial (ou détecter l'insolvabilité), *DynABT* propage les perturbations (phase de propagation) et répare la dernière solution trouvée ou la dernière affectation. *LiveABT*, différemment à *DynABT*, est exécuté en une

seule phase (phase de résolution) et traite les perturbations intrinsèquement au moment de leurs survenance, contrairement à *DynABT* qui ne traite et ne propage les perturbations qu'après avoir terminé la phase de résolution.

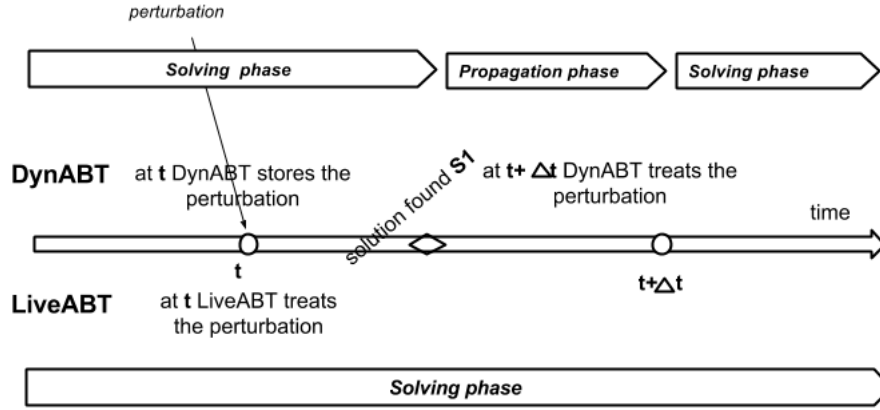


FIGURE 6.1 – comportement de *LiveABT* comparé à *DynABT*

6.3 exemple de simulation

Considérons l'exemple d'un *DDisCSP* binaire contenant 4 *agents* : A_1 , A_2 , A_3 et A_4 . Chaque agent a une seule variable, respectivement X_1 , X_2 , X_3 et X_4 . Les contraintes du problème sont indiquées dans fig. 6.2 :

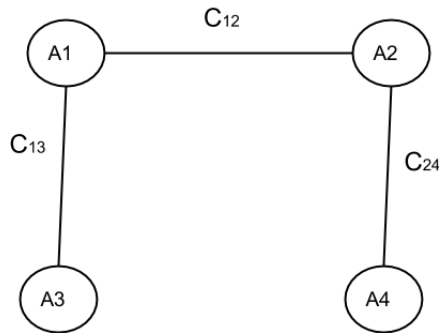


FIGURE 6.2 – Exemple d'un *DisCSP*

6.3.1 Perturbation 1 : L'ajout de contrainte

Lorsqu'une perturbation affecte le problème, telle qu'une nouvelle contrainte doit être ajoutée (C_{23} entre les agents A_2 et A_3 dans fig. 6.3).

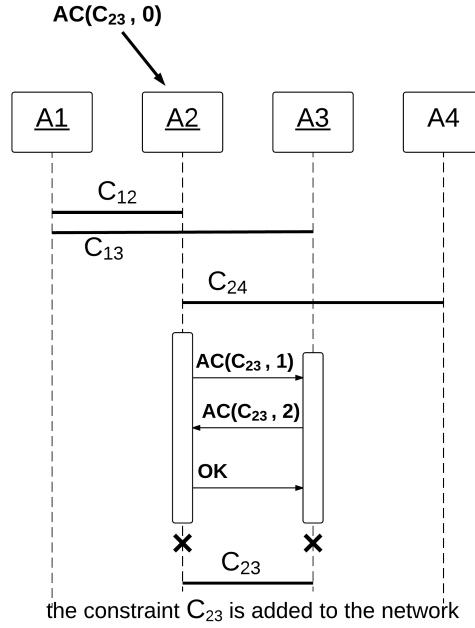


FIGURE 6.3 – diagramme de séquence d'un ajout de contrainte

Le Manager du système de perturbation envoie un message d'ajout de contrainte ($AC(A2, C_{23}, stage)$ au stade initial ($stage = 0$) à l'agent de priorité supérieure concerné par la contrainte (l'agent $A2$). Ce message contient l'adresse de la deuxième partie de la contrainte (l'agent $A3$) tel que $A2$ ignore $A3$, qui n'est pas encore son voisin, et contient aussi le stade de l'injection de la contrainte qui est incrémenté à chaque étape. Lorsque $A2$ reçoit $AC(A2, C_{23}, stage)$, il ajoute $A3$ dans sa liste des voisins, met C_{23} dans sa liste de contraintes et envoie le message $AC(A3, C_{23}, stage++)$ à $A3$. Quand $A3$ reçoit le message d'ajout de contrainte, il ajoute $A2$ à ses voisins, met à jour son *agentView*, enregistre la nouvelle contrainte C_{23} dans sa liste de contraintes et informe $A2$ que la contrainte est bien ajoutée en lui envoyant le message $AC(A2, C_{23}, stage++)$. $A2$ réagit en envoyant un message *OK* à $A3$, pour évaluer la nouvelle contrainte et changer sa valeur si nécessaire. A la fin du processus *addConstraint*, le réseau devient comme indiqué dans fig. 6.4.

Ainsi, pour ajouter une contrainte dans le réseau, 4 messages sont nécessaires.

6.3.2 Perturbation 2 : La suppression de contrainte

a- Suppression d'une contrainte :

Si la perturbation est une suppression de contrainte, *LiveABT* supprime

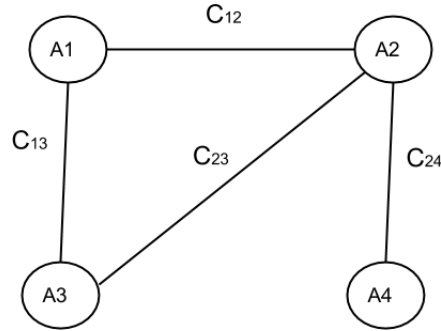


FIGURE 6.4 – le problème après l'ajout de contrainte

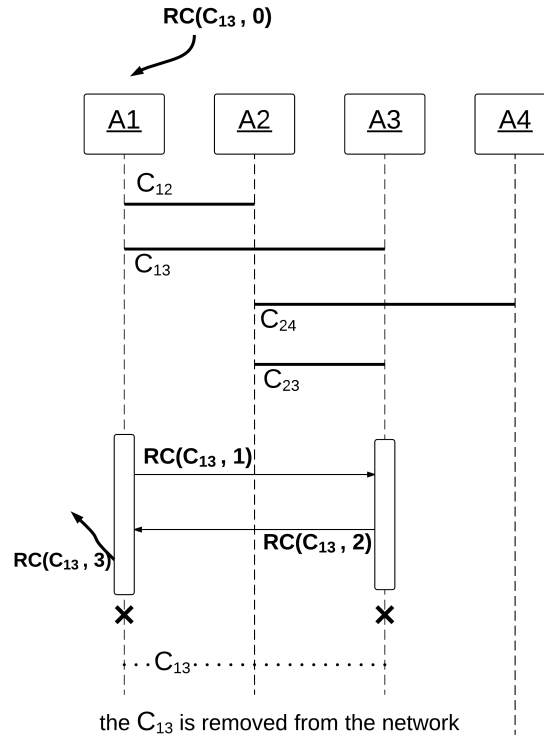


FIGURE 6.5 – Diagramme de séquence d'une suppression de contrainte 1/2

la contrainte. Ainsi, différemment au cas précédent, il diffuse un message *AdjustNogood* pour nettoyer les *nogoods* enregistrés en supprimant tout *nogood* avec la contrainte en question comme justification.

Lorsque *A1* reçoit un message de suppression de contrainte ($RC(A1, C_{13}, 0)$) comme indiqué dans fig.6.5, il supprime l'agent *A3* de la liste de ses voisins, supprime la contrainte C_{13} de sa liste de contraintes et envoie le message $RC(A3, C_{13}, stage++)$ à *A3*. L'agent *A3* supprime également *A1* de sa liste

de voisins, met à jour son *AgentView*, supprime la contrainte C_{13} de sa liste de contraintes et envoie à $A1$ le message $RC(A1, C_{13}, stage++)$ en informant que la propagation de la suppression de contrainte est terminée. Ainsi, l'agent $A1$ met au courant le système, généralement le *master*, par le message $RC(sysme, C_{13}, stage+)$. Par conséquent, pour supprimer une contrainte du réseau, uniquement 4 messages sont nécessaires.

b- Mise à jour des nogoods :

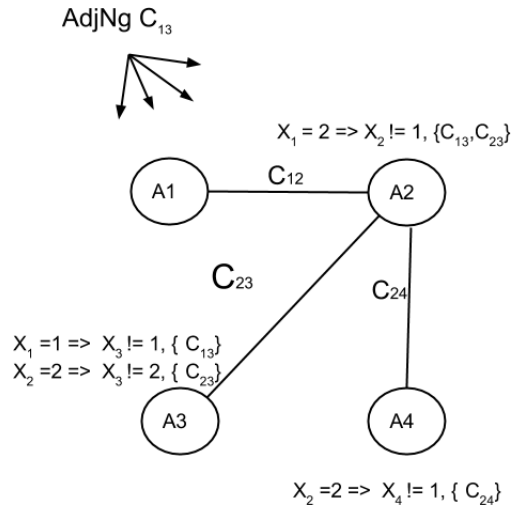


FIGURE 6.6 – Suppression de contrainte 2/2

Chaque agent, lors de la réception du message *AdjustNogood* (*AdjNg* C_{13} comme indiqué dans fig. 6.6), vérifie ses *nogoods* pour les nettoyer et met à jour son domaine. Ainsi, dans cet exemple, $A2$ supprime la justification C_{13} de ses *nogoods*, en gardant ce *nogood*, puisque C_{13} n'est pas sa seule justification. $A3$ restaure la valeur 1 dans son domaine et nettoie ses *nogoods* (supprime le premier *nogood*). De plus, $A4$ conserve ses *nogoods* après vérification.

Pour nettoyer tous les *nogoods*, n messages sont nécessaires, tel que n est le nombre d'agents du réseau.

6.4 Exécution

La description de *LiveABT* [9] est représentée dans fig.6.7. *LiveABT* est la procédure principale (lignes 1-11). Il est implémenté dans chaque agent. Chaque agent détecte ses parents (*AgentView*) et ses fils (ligne 1) et se met à l'écoute du réseau. Si l'agent reçoit un message *CheckAgentView*, il exécute la fonction *CheckAgentView* (lignes 12-19). Si le message provient d'un agent de priorité plus élevée, l'agent récepteur met à jour son

Procedure *LiveABT_kernel()*

```

1.  compute  $\Gamma + (self)$  and  $\Gamma - (self)$ ;
2.  CheckAgentView(null);
3.  end  $\leftarrow$  false;
4.  while ( $\neg end$ ) do
5.    msg  $\leftarrow$  getMsg();
6.    switch (msg.type)
7.      Stop : end  $\leftarrow$  true;
8.      AddConstraint : AC(con.getUpper, con, 0);
9.      RemoveConstraint : RC(con.getUpper, con, 0);
10.     Adjustngood : IncoherentConstraint(con);
11.     else : CheckAgentView(msg);

```

Procedure *CheckAgentView*(*msg*)

```

12. if msg is relevant then
13.   update local context;
14. if Consistent(myValue, myContext) then Return;
15. myValue  $\leftarrow$  ChooseValue();
16. if myValue then
17.   for each child  $\in$   $\Gamma + (self)$  do
18.     sendMsg : Info(child, myValue);
19. else Backtrack;

```

Procedure *Backtrack*()

```

20. newNogood  $\leftarrow$  Solve(myNogoods);
21. if newNogood = empty then
22.   end  $\leftarrow$  true;
23.   sendMsg : Stop(system);
24. else
25.   sendMsg : Back(newNogood);
26.   update local context;
27.   CheckAgentView(null);

```

Function *ChooseValue*()

```

28. for each v  $\in$  D(self) not eliminated by myNogoods do
29.   if Consistent(v, myContext) then Return (v);
30.   else
31.     Add(X = valX  $\Rightarrow$   $\neg v$ ; C{ci} , myNogoods);
32. Return (null);

```

Procedure *AddConstraint*(*con*)

```

33. switch (stage)
34.   0 : Add con.neighbour to my Neighbours List;
35.     Add con to Constraints List;
36.     sendMsg : AC(con.getlower, con, stage++);
37.   1 : Add con.neighbour to Neighbours List;

```

```

38.   Add con to Constraints List ;
39.   sendMsg : AC(con.getupper, con, stage++);
40.   2 : sendMsg : Info(con.getlower(), myvalue);

Procedure RemoveConstraint(con)
41. IncoherentConstraint(con);
42. switch (stage)
43.   0 : Remove con.neighbour from Neighbours List ;
44.     Remove unique neighbour from Agent Views ;
45.     Remove con from Constraint List ;
46.     sendMsg : RC(con.getlower, con, stage++);
47.   1 : Remove con.neighbour from Neighbours List ;
48.     Remove unique neighbour from Agent View ;
49.     Remove con from Constraint List ;
50.     sendMsg : RC(con.getupper, con, stage++);
51.   2 : sendMsg : RC(system, con, stage++);

Procedure IncoherentConstraint(con)
52. for each nogood in nogoodStore do
53.   if nogood.justification.Contains(con) then
54.     nogood.justification.Remove(con);
55.   if nogood.justification.isEmpty() then
56.     restore nogood.eliminatedValue to domain;
57.     nogoodStore.Remove(nogood);

```

FIGURE 6.7 – Pseudo-code de l'algorithme *LiveABT*

agentView (lignes 12-13), si sa valeur n'est pas consistante (lignes 14-15), il appelle la fonction *ChooseValue* (lignes 28-32) pour instancier sa variable. Si sa variable est instanciée, l'agent envoie un message *Info* à ses agents fils pour les informer de mettre à jour leurs *agentviews* (lignes 16-18). Sinon, la procédure *Backtrack* est lancée afin d'informer l'agent père pour modifier sa valeur, en raison de son domaine vide (lignes 20-27).

A tout moments, n'importe quel agent peut recevoir une perturbation illustrée par un message de type *AddConstraint* ou *RemoveConstraint* (lignes 8-9).

Si un agent reçoit le message *AddConstraint* (ligne 8), *LiveABT* réagit en appelant la procédure *AddConstraint* (lignes 33-40), qui met à jour toutes les informations sur les deux agents impliqués dans la contrainte, et l'intègre dans le réseau. *stage* représente la progression de l'ajout (ou la suppression) d'une contrainte. C'est un compteur lié à chaque perturbation, et il est utilisé par l'agent pour déterminer quelle action à exécuter puisqu'un agent peut être impliqué dans de nombreuses perturbations. Les messages *AC* et *info* sont échangés entre les agents concernés par la nouvelle contrainte pour mettre à jour leurs contextes et pour activer la nouvelle contrainte dans le réseau.

Si une contrainte est supprimée (ligne 9), *LiveABT* appelle la procédure *RemoveConstraint* (lignes 41-51). Comme pour *AddConstraint*, les messages *RC* sont échangés entre les agents concernés pour accomplir l'élimination de la contrainte du réseau. Une fois que la contrainte est supprimée, un message de *AdjustNogood* est diffusé pour nettoyer tous les *nogoods* qui contiennent la contrainte supprimée comme justification (lignes 52-57), afin de remettre les valeurs interdites à leurs domaines. La procédure *IncoherentConstraint* garantit que les valeurs invalidées par les contraintes supprimées seront remises à leurs domaines.

Le message *Stop* (ligne 7) est diffusé de la part du *master* à tous les agents du réseau, avec succès, lorsqu'un silence est détecté, si une solution est trouvée, ou avec un échec, si un *nogood* vide est généré.

6.5 Terminaison et complétude

La terminaison de *LiveABT* est assurée puisque *ABT* termine [83]. En fait, *LiveABT* respecte le paradigme *ABT* et utilise le même mécanisme pour traiter et communiquer toutes les perturbations dans le réseau (les messages *AddConstraint* et *RemoveConstraint* deviennent pareils aux messages standards d'*ABT*, tels les messages *Ok* et *nogood*), par conséquent, l'approche termine.

En ce qui concerne la complétude, si le problème final (après les perturbations) a une solution, *LiveABT* la trouve. En fait, après l'intégration de toutes les perturbations, le protocole *LiveABT* devient similaire au standard *ABT*, qui est complet [83].

6.6 Expérimentations

Nous avons choisi de comparer expérimentalement *LiveABT* à *DynABT* puisque *DynABT* est la dernière approche proposée pour les problèmes *DDisCSP*. De plus, *DynABT* a déjà prouvé sa grande performance par rapport à *DynDBA* [47] en termes du temps d'exécution et de charge du réseau. Les algorithmes sont évalués sur des problèmes *DDisCSPs* binaires aléatoires et uniformes, des problèmes de coloriage de graphe et aussi sur les problèmes de planification de réunions *MSPs*. Pour simuler comme en réalité les moments d'injection du taux de perturbation p , nous choisissons de diviser p sur un ensemble de quantités q (distribution normale) $[t_1, t_q]$, qui sont générés au moment de la résolution, et à chaque instant $t_i \in [t_1, t_q]$, le problème est perturbé par $\frac{p}{q}$ perturbations.

En fait, il existe différentes façons de perturber le problème initial. Nous avons choisi le même scénario de perturbation présenté dans [57] qui maintient la densité fixe entre le problème initial et le problème final, ainsi, les perturbations sont divisées en deux parties de $\frac{p}{2 \times q} \%$ de contraintes à ajouter

et $\frac{p}{2 \times q}\%$ de contraintes à supprimées. Par conséquent, le problème initial et celui perturbé ont la même densité. Par exemple, Soit un problème avec 60 contraintes à perturber à l'ordre de $p = 10\%$, selon le scénario adopté, 6 contraintes seront changées tels que 3 seront supprimées et 3 autres seront ajoutées. Nous générons au hasard q moments (instants d'injection). La distribution peut être de la forme $q = 3 = (1, 1, 1)$, c-à-d que trois moments ont été générés, à chaque moment une contrainte est ajoutée et une autre est supprimée, ou $q = 2 = (1, 2)$, à savoir, au premier moment d'injection une contrainte est ajoutée et une autre est supprimé et lors de la seconde perturbation, deux contraintes sont ajoutées/supprimées, nous pouvons aussi avoir $q = 1 = (3)$ ce qui implique que toutes les perturbations se produisent au même moment.

Toutes les expérimentations sont effectuées sur la plate-forme *Dischoco2* [73]. Nous évaluons les performances des algorithmes en utilisant les métriques : le nombre total de messages échangés entre les agents ($\#Msg$) [45] et le nombre non-concurrent des contraintes testées ($\#NCCCs$) [51].

6.6.1 Expérimentations sur les problèmes aléatoires

Un *DDisCSP* binaire aléatoire uniforme P est une séquence de *DisCSPs* binaire aléatoire uniforme : $P_0, P_1, \dots, P_{\alpha-1}, P_\alpha$ où P_0 est le problème original et pour chaque P_i , de nouvelles contraintes sont ajoutées/supprimées au/du problème précédent. Dans ce qui suit, nous choisissons les deux taux de perturbations : 6% qui représente un environnement à faible perturbations et 30% qui représente un environnement oscillé.

Un problème *DisCSPs* est caractérisé par $\langle n, d, p1, p2 \rangle$, où n est le nombre d'agents/variables, d le nombre de valeurs dans chaque domaine, $p1$ représente la connectivité (densité) du réseau et $p2$ est la dureté des contraintes. Les tests sont effectués sur des problèmes de $\langle 20, 10, 0.2, 0.1 \rangle$ à $\langle 20, 10, 0.2, 0.9 \rangle$ avec un pas de $p2 = 0.1$. Nous notons que nous avons effectué des tests aussi sur les problèmes de $\langle 30, 12, 0.7, 0.1 \rangle$ à $\langle 30, 12, 0.7, 0.9 \rangle$ et les résultats ont montré la même amélioration. Pour chaque dureté fixe $p2$, 50 différentes instances ont été perturbées aléatoirement et traitées par chacune des deux approches. Les résultats présentés sont les moyennes des 50 exécutions.

La région de $p2 \in [0.1, 0.6]$ est la région de solvabilité, la région de $p2 = 0.7$ est un mélange de problèmes solvables et non solvables et au-delà de 0.7 les problèmes sont insolubles.

Les résultats montrent que *LiveABT* se comporte mieux que *DynABT*, par rapport aux deux critères de comparaison, à savoir le nombre de *NCCCs* et de *Msgs* (figs. 6.8 et 6.9), indépendamment de la solvabilité du problème, en particulier lorsque les perturbations sont importantes (fig. 6.9).

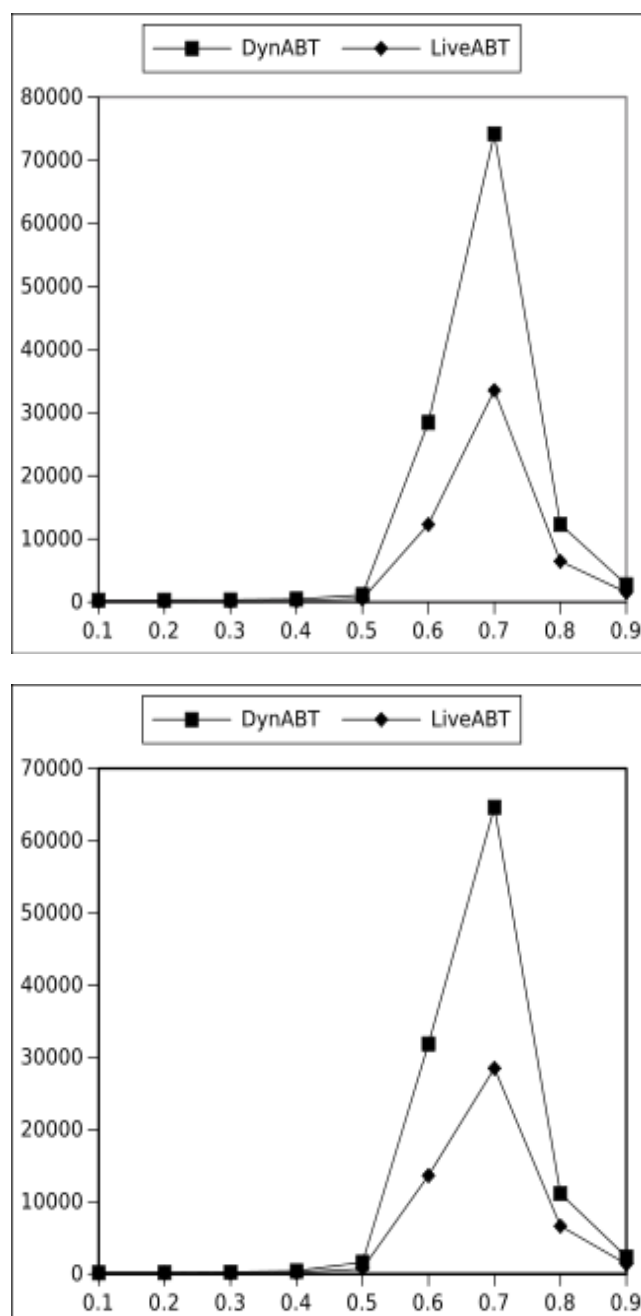


FIGURE 6.8 – Problèmes aléatoires : Nombre de messages envoyés et $NCCCs$ pour 6% de perturbation

6.6.2 Expérimentations sur les problèmes de coloriage de graphe

Pour évaluer notre approche pour d'autres types de problèmes, nous choisissons les problèmes de coloriage de graphes distribués. Ce problème peut

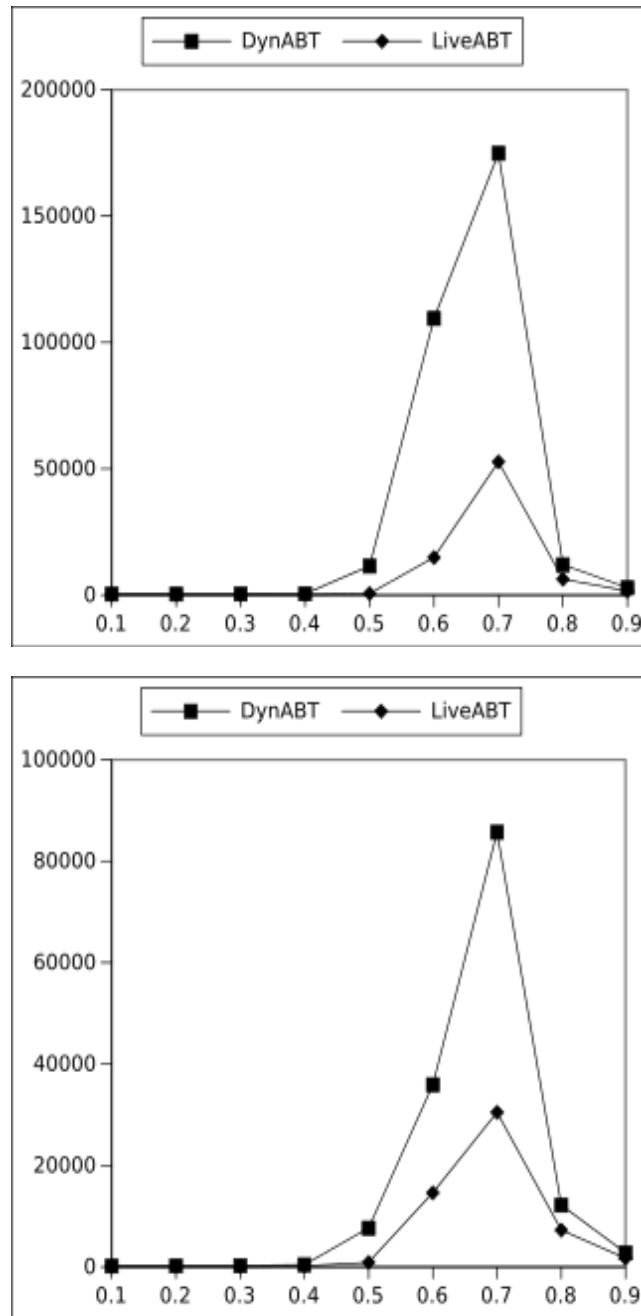


FIGURE 6.9 – Problèmes aléatoires : Nombre de messages envoyés et $NCCCs$ pour 30% de perturbation

représenter facilement de nombreux problèmes réels tels que les problèmes d'allocation de canaux pour les systèmes de communication mobiles, dans

lesquels les cellules adjacentes ne peuvent pas utiliser les mêmes canaux pour éviter les interférences.

Le problème de coloriage de graphe distribué est caractérisé par trois paramètres, à savoir, le nombre d'agents/variables n , le nombre de couleurs pour chaque agent k , et le nombre de liens (contraintes) entre les agents m .

Nous avons aléatoirement généré des problèmes avec 50 agents/variables et m arcs par la méthode [54]. Les problèmes générés sont connectés et solvables, la configuration de $k = 3$ et $m = n \times 2.7$ est la zone critique (les problèmes sont les plus difficiles à résoudre) [20]. Nous étudions le comportement des deux approches *LiveABT* et *DynABT* par rapport aux taux de perturbations.

Pour chaque taux de perturbation, 50 différentes instances sont générées aléatoirement suivant la méthode signalée ci dessus, ces instances sont perturbées de 2% à 40% par un pas de 2% et traitées par chacune des deux approches. Les résultats présentés sont les moyennes de ces 50 exécutions.

Les résultats dans fig. 6.10 montrent que *DynABT* dépasse légèrement *LiveABT* lorsque les perturbations sont faibles. Cependant, *LiveABT* teste moins de contraintes (*NCCCs*) et échange moins de messages que *DynABT* dans les environnements à moyenne et à haute perturbation.

Nous notons également que lorsque les problèmes sont assez perturbés, *LiveABT* déploie moins d'efforts pour trouver une solution, contrairement au *DynABT* pour lequel l'effort augmente de façon linéaire en fonction du taux de perturbation.

6.6.3 Expérimentations sur les problèmes de planification de réunions

Le problème *MSP* distribué (*DisMSP*) peut être modélisé comme suit :

- Ensemble d'agents/participants $A = \{A_1, A_2, \dots, A_n\}$.
- Ensemble de variables X : chaque agent $A_i \in A$ contient un ensemble de variables $X_i = \{x_{1i}, x_{2i}, \dots, x_{m_i i}\}$ tels que :
 - $X = \sum_{i=1}^n X_i$
 - m_i est le nombre de réunions auxquelles A_i participe
 - $\forall x_{ji} \in X_i, x_{ji}$ représente une réunion à laquelle A_i participe.
- Ensemble de domaines D : {créneaux horaires de l'ensemble des réunions}.
- Ensemble de contraintes C :
 - contrainte intra-agent : La contrainte du temps d'arrivée Pour chaque paire de variables ($x_{ji}, x_{j'i}$ avec $j \neq j'$) il y a une contrainte de temps d'arrivée (*arrival-time*) : $t_{jj'}(|x_{j'i} - x_{ji}| \geq t_{jj'})$, si i assiste aux deux réunions j et j' .
 - contrainte inter-agent : contrainte d'égalité Pour chaque paire de variables ($x_{ji}, x_{j'i'}$ avec $i \neq i'$), il y a une contrainte d'égalité $x_{ji} = x_{j'i'}$, s'il y a deux participants i et i' qui assistent à une même réunion j .

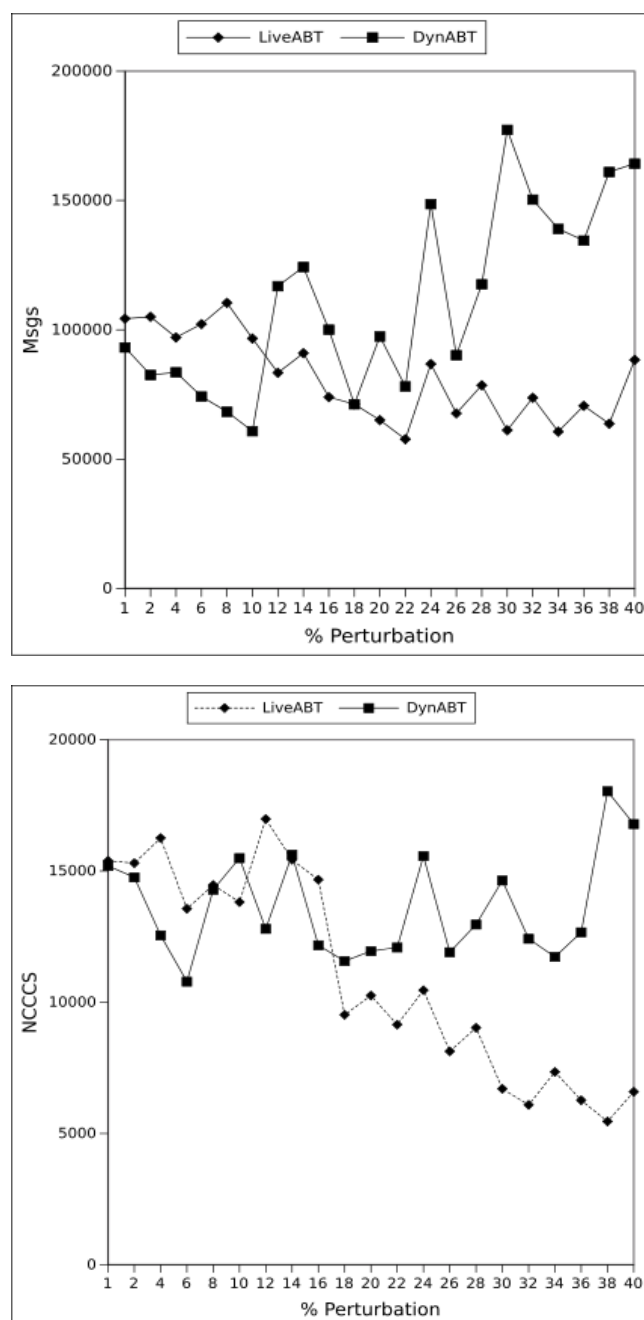


FIGURE 6.10 – Problème de coloriage de graphe : le comportement de *LiveABT* et *DynABT* selon le taux de perturbation

Nous présentons nos résultats pour la classe $\langle 20, 4, 11, 3, 10, 1, 80 \rangle$ et nous modifions le taux de de perturbation de contraintes (les contraintes concernées par les perturbations sont les contraintes du temps d'arrivée *arrival*-

time) de 2% à 30%. Nous avons généré 50 cas différents résolus par chaque algorithme et les résultats sont les moyennes de ces 50 exécutions.

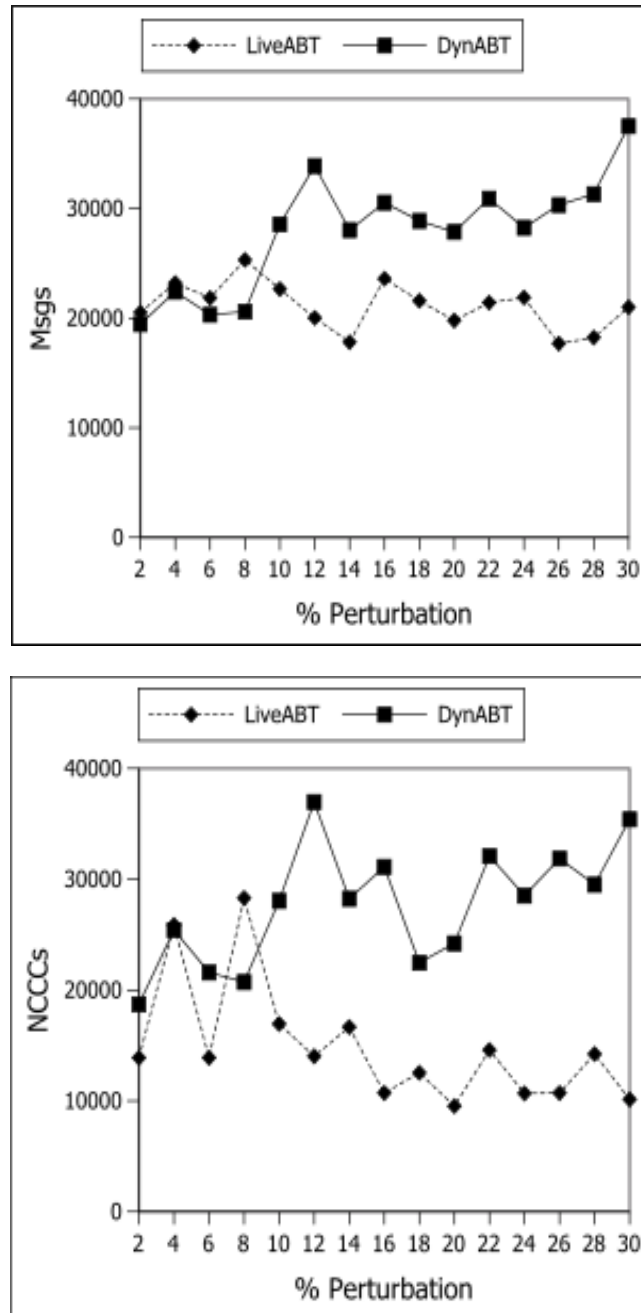


FIGURE 6.11 – Le problème de planification de réunions : le comportement du *LiveABT* et *DynABT* par rapport au taux de perturbation

Les résultats dans fig. 6.11 montrent que lorsque le taux de perturbation est faible, *DynABT* et *LiveABT* se comportent presque de la même façon selon les deux métriques choisies, cependant, lorsque le taux de perturbation augmente (à partir de ≥ 10), *LiveABT* envoie moins de messages et teste moins de contraintes par rapport au *DynABT*.

6.6.4 Discussion

Pour les problèmes aléatoires, quel que soit le taux de perturbation, à savoir 6% et 30%, les résultats montrent que *LiveABT* fait moins de test de contraintes et envoie moins de messages que *DynABT* pour traiter un problème. En fait, *LiveABT* a besoin de 4 messages lors de l'injection d'une nouvelle contrainte, et $N_A + 4$ messages pour le traitement de la suppression de contrainte, où N_A est le nombre d'agents dans le réseau. Cependant, en plus de cette quantité ($N_A + 4 + 4$), pour traiter une perturbation (une addition et une suppression), *DynABT* a besoin de plus de messages (propager la perturbation) à diffuser pour arrêter et redémarrer la résolution ($2 \times N_A$). En ce qui concerne le nombre de contraintes à tester (*NCCCCs*), lorsque *DynABT* démarre la réparation de la solution, chaque agent lance le processus *ABT* c-à-d évalue ses contraintes, donc, *DynABT* teste au moins (si la perturbation n'a aucun effet sur la solution actuelle), N_C contraintes plus que *LiveABT*, tel que N_C est le nombre de contraintes du problème. D'autre part, le comportement de *LiveABT* et *DynABT* sur les problèmes de planification de réunions et les problèmes de coloriage de graphes montre que lorsque le problème est plus perturbé, l'effort de *LiveABT* diminue, ceci est dû à l'impact de la suppression d'une contrainte qui est beaucoup plus important que l'ajout d'une contrainte. Plus précisément, lors de la suppression d'une contrainte, tous les agents sont affectés, et ils doivent nettoyer leurs *nogoods* et récupérer les valeurs supprimées de leurs domaines tandis que l'ajout d'une contrainte en temps réel n'affecte en général que les deux agents impliqués.

Ainsi, si le traitement des perturbations se fait en temps réel lors de la résolution, le problème sera relaxé.

Par contre, l'effort de *DynABT* augmente lorsque le taux de perturbation augmente parce que *DynABT* accumule l'effort des phases successives de résolution.

Enfin, l'inconvénient majeur de *DynABT* est la phase de propagation, dans laquelle l'algorithme propage les perturbations, donc, pour les petites perturbations (peu de phases de propagation), *DynABT* est un bon concurrent de *LiveABT* comme montré dans les fig. 6.10 et 6.11.

Chapitre 7

Conclusion et perspectives

Dans le cadre de cette thèse, nous avons proposés de nouvelles techniques pour réparer les *CSP* dynamiques.

Nous avons tout d'abord concrétisé notre travail en modélisant un problème pratique d'allocation de ressources, qui consiste à répartir les étudiants/stagiaires de la (*FMPC*) aux différents services hospitaliers du *CHU* de Casablanca, sous forme de *CSP*.

Concernant les techniques de réparation de *DCSPs*, nous avons proposé des approches de réparation, que ce soit pour les *DCSPs* centralisés ou distribués, ainsi que de nouvelles heuristiques d'ordonnancement.

Pour les approches de réparation de *DCSPs* centralisés, nous sommes partis de *PDB*, et avons proposé trois approches basées sur cette technique. Nous avons proposé *EPDB*, qui bénéficie de la flexibilité de *PDB* tout en utilisant les heuristiques d'ordonnancement, afin de mieux diriger la recherche en gardant les sous problèmes les plus complexes stables autant que possible. Par la suite, nous avons proposé *LRB* et *IPDB* qui apportent une amélioration intéressante à *EPDB*, et donc à *PDB*, dans la mesure où ils réduisent considérablement les vérifications d'ordres qui consomment un temps de calcul très remarquable. *LRB* réduit ce nombre de tests d'ordres en fixant préalablement l'ordre entre les variables, par l'utilisation d'heuristiques de choix de variables statiques uniquement, tandis que *IPDB* est capable de déduire rapidement les ordres entre les variables, en la dotant d'une nouvelle structure *successors* qui permet de fournir cette information directement. Nos résultats expérimentaux montrent que notre première approche améliore considérablement *PDB*, tandis que la deuxième et troisième approches sont compétitives avec l'état de l'art, au moment où elles produisent des résultats comparables à ceux de la stratégie *MAC*.

Concernant les heuristiques d'ordonnancement, nous avons proposé une nouvelle heuristique de choix de variables, basée sur l'heuristique *deg*, qui évalue les variables du réseau de contraintes selon leurs positions dans l'ensemble du réseau, nous avons aussi proposé des heuristiques d'ordonnancement de

contraintes, afin d'établir une stratégie concernant l'ordre à suivre pour tester les contraintes à satisfaire. D'après les expérimentations, l'utilisation des ces nouvelles heuristiques améliorent la réparation des problèmes dynamique, en guidant mieux l'algorithme de réparation.

Enfin, en ce qui concerne les approches de réparation de *DCSPs* distribués, nous avons proposé *LiveABT*, un nouveau protocole basé sur l'approche *ABT*, qui vise à réparer les problèmes *DDisCSPs* en temps réel. En se basant sur les résultats des expérimentations sur différents types de problèmes, nous avons montré que cette approche dépasse *DynABT*, quelle que soit la solvabilité des problèmes.

Perspectives

Concernant la modélisation des stages sous forme de *CSP*, déjà faite, nous comptons comparer l'efficacité et la qualité de cette application, en termes de modélisation et de résolution, avec celle déjà existante.

Pour les *DCSPs* centralisés, nous envisageons implémenter un solveur de résolution de *CSPs*, basé sur *IPDB*, qui bénéficie des avantages de cette approche et résout les *CSPs* en réparant une affectation complète qui viole un nombre de contraintes minimal, dans le but de converger plus rapidement vers une solution. Nous comptons aussi développer une version *IPDB* pour les problèmes à perturbation minimale [29], pour préserver les solutions des *DCSPs* autant que possible et garantir ainsi une meilleure stabilité du réseau de contraintes dynamiques.

En ce qui concerne les *DCSPs* distribués, nous avons l'intention de développer l'agent *LiveABT* pour détecter de manière autonome les perturbations des différents agents (changement d'adresse, déconnexion ou ajout d'agents) et réagir de manière appropriée sans attendre les instructions du *Master*. Nous prévoyons également mettre à niveau *LiveABT* et l'appliquer dans d'autres cas réels, en particulier dans le système de contrôle robotique distribué et le problème du réseau intelligent.

Liste des abréviations

ABT : Asynchronous BackTracking
AC : Arc Consistency
AI : Artificial Intelligence
BJ : BackJumping
BT : BackTracking
CBJ : Conflict-directed BackJumping
CCs : Constraints Checks
COP : Constraint Optimization Problem
CP : Constraint Programming
CSP : Constraint Satisfaction Problem
DAI : Dynamic Artificial Intelligence
DBA : Distributed Breakout Algorithm
DBT : Dynamic Backtracking
DCO : Dynamic Constraint Ordering
DCSP : Dynamic Constraint Satisfaction Problem
DDisCSP : Dynamic Distributed Constraint Satisfaction Problem
DisCSP : Distributed Constraint Satisfaction Problem
DMSP : Dynamic Meeting Scheduling Problem
DSA : Distributed Stochastic Search Algorithm
DVO : Dynamic Variable Ordering
DynABT : Dynamic Asynchronous Backtracking
EPDB : Extended Partial-order Dynamic Backtracking
FC : Forward Checking
FMPC : Faculté de Médecine et de Pharmacie de Casablanca
HD : Hamming Distance
IC : Injected Constraints
IPDB : Improved Partial-order Dynamic Backtracking
LC : Local Changes
LiveABT : Live Asynchronous BackTracking
LRB : Lazy Repairing Backtracking
MAC : Maintaining Arc Consistency
MSP : Meeting Scheduling Problem
NCCCs : Non-Concurrent Constraints Checks
PDB : Partial-order Dynamic Backtracking
PDeg : Profound Degree
SVA : Solution Value Assignment
SVO : Static Variable Ordering
VOH : Variable Ordering Heuristic

Bibliographie

- [1] Y. Acodad, A. Benamrane, I. Benelallam, and E. Bouyakhf. Profound degree : A conservative heuristic to repair dynamic csps. In *Artificial Intelligence Applications and Innovations - 10th (IFIP) WG 12.5 International Conference, AIAI 2014, Rhodes, Greece, September 19-21, 2014. Proceedings*, pages 140–149, 2014.
- [2] Y. Acodad, A. Benamrane, I. Benelallam, and E. Bouyakhf. On improving the dynamic constraint satisfaction problems repairing. *International Journal of Artificial Intelligence*, 15(2) :79–101, 2017.
- [3] Y. Acodad, A. Benamrane, I. Benelallam, and E. Bouyakhf. Lazy repairing backtracking for dynamic constraint satisfaction problems. *Computing And Informatics*, En cours.
- [4] Y. Acodad, I. Benelallam, S. Hammoujan, and E. H. Bouyakhf. Extended partial-order dynamic backtracking algorithm for dynamically changed environments. In *Tools with Artificial Intelligence (ICTAI), 2012 IEEE 24th International Conference on*, volume 1, pages 580–587, 2012.
- [5] M. Arangú, M. A. Salido, and F. Barber. AC2001-OP : an arc-consistency algorithm for constraint satisfaction problems. In *Trends in Applied Intelligent Systems - 23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE)*, pages 219–228, 2010.
- [6] F. Bacchus and P. van Beek. On the conversion between non-binary constraint satisfaction problems. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI '98/IAAI '98*, pages 311–318, 1998.
- [7] T. Balafoutis and K. Stergiou. Adaptive branching for constraint satisfaction problems. In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, pages 855–860, 2010.
- [8] R. Barták, T. Müller, and H. Rudová. A new approach to modeling and solving minimal perturbation problems. In *Recent Advances in*

- Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming, (CSCLP) 2003, Budapest, Hungary, June 30 - July 2, 2003, Selected Papers*, pages 233–249, 2003.
- [9] A. Benamrane, Y. Acodad, I. Benelallam, and E. Bouyakhf. Liveabt : A real-time repairing protocol for incremental and dynamic discsps. *International Journal of Artificial Intelligence*, 15(1) :126–146, 2017.
 - [10] A. Benamrane, Y. Acodad, I. Benelallam, E. H. Bouyakhf, and M. B. Othmani. Modeling trainings in faculty of medicine and pharmacy of casablanca as constraint satisfaction problem. In *The Thirteenth International Workshop on Constraint Modelling and Reformulation (ModRef) of Principles and Practice of Constraint Programming (CP)*, pages 17–23, 2014.
 - [11] N. Berger. *Modeling and Solving Mixed Continuous/Discrete Constraint Satisfaction and Optimisation Problem*. Theses, Université de Nantes, Oct 2010.
 - [12] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65(1) :179 – 190, 1994.
 - [13] C. Bessière, E. C. Freuder, and J.-C. Regin. Using inference to reduce arc consistency computation. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI’95*, pages 592–598, 1995.
 - [14] C. Bessière, E. C. Freuder, and J.-C. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artif. Intell.*, 107(1) :125–148, Jan. 1999.
 - [15] C. Bessière, A. Maestre, I. Brito, and P. Meseguer. Asynchronous backtracking without adding links : a new member in the abt family. *Artificial Intelligence*, 161(1) :7–24, 2005.
 - [16] C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI’01*, pages 309–315, 2001.
 - [17] C. Bessière, J. C. Régin, R. H. C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence (AI)*, 165(2) :165–185, 2005.
 - [18] C. Bliet. Generalizing partial order and dynamic backtracking. In J. Mostow and C. Rich, editors, *AAAI/IAAI*, pages 319–325. AAAI Press / The MIT Press, 1998.
 - [19] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI’04*, pages 146–150, 2004.

- [20] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *IJCAI*, volume 91, pages 331–340, 1991.
- [21] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau. Using iterative repair to improve the responsiveness of planning and scheduling. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems*, AIPS'00, pages 300–307. AAAI Press, 2000.
- [22] T. Chkili, A. Harouchi, B. M., and H. A. Decret 2.91.527 du 21 kaada 1413 (13 mai 1993) relatif à la situation des externes, des internes et des résidents des centres hospitaliers. *Decret 2.91.527 du 21 Kaada 1413 (13 mai 1993)*, 1993.
- [23] J. M. Crawford and L. D. Auton. Experimental results on the crossover point in random 3-sat. *Artificial intelligence (AI)*, 81(1) :31–57, 1996.
- [24] R. Debruyne and C. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 412–417, 1997.
- [25] R. Debruyne and C. Bessière. Domain filtering consistencies. *Journal of Artificial Intelligence Research (JAIR)*, 14 :205–230, 2001.
- [26] R. Dechter. Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3) :273–312, Jan. 1990.
- [27] R. Dechter and A. Dechter. *Belief maintenance in dynamic constraint networks*. AAAI Press / The MIT Press, 1988.
- [28] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'89, pages 271–277, 1989.
- [29] E. M. El Graoui, I. Benelallam, and E. H. Bouyakhf. A commentary on "hybrid search for minimal perturbation in dynamic csps". *Constraints*, 21(2) :349–354, Apr. 2016.
- [30] J. Ferber. *Multi-agent systems : an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading, 1999.
- [31] E. C. Freuder. Synthesizing constraint expressions. *Commun. ACM*, 21(11) :958–966, Nov. 1978.
- [32] E. C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1) :24–32, Jan. 1982.
- [33] J. Gaschnig. Experimental case studies of backtrack vs. Waltz-type vs. new algorithms for satisficing assignment problems. In *Proceedings of the Second Canadian Conference on Artificial Intelligence*, pages 268–277, 1978.

- [34] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of arc consistency. In *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, pages 327–340, 1997.
- [35] M. L. Ginsberg. Dynamic backtracking. *J. Artif. Intell. Res.*, 1 :25–46, 1993.
- [36] M. L. Ginsberg and D. A. McAllester. Gsat and dynamic backtracking. In *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, PPCP '94*, pages 243–265, 1994.
- [37] S. Hammoujan, Y. Acodad, I. Benelallam, and E. H. Bouyakhf. Dynamic constraint ordering heuristics for dynamic csps. *Applied Mathematical Sciences*, 7 :137 – 140, 2013.
- [38] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3) :263 – 313, 1980.
- [39] E. Hebrard, B. Hnich, B. O’Sullivan, and T. Walsh. Finding diverse and similar solutions in constraint programming. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 1, AAAI’05*, pages 372–377. AAAI Press, 2005.
- [40] P. V. Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2) :291 – 321, 1992.
- [41] N. Jussien, G. Rochart, and X. Lorca. Choco : an Open Source Java Constraint Programming Library. In *CPAIOR’08 Workshop on Open-Source Software for Integer and Constraint Programming (OSSICP’08)*, pages 1–10, 2008.
- [42] M. Khichane. *Optimisation sous contraintes par Intelligence Collective Auto-adaptative*. Theses, Université Claude Bernard, Oct. 2010.
- [43] V. Kumar. Algorithms for constraint-satisfaction problems : A survey. *AI Mag.*, 13(1) :32–44, Apr. 1992.
- [44] C. Lecoutre, L. Sais, and J. Vion. Using SAT encodings to drive CSP value ordering heuristics. *JSAT*, 1(3-4) :169–186, 2007.
- [45] N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.
- [46] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99 – 118, 1977.
- [47] R. Mailler. Comparing two approaches to dynamic, distributed constraint satisfaction. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1049–1056. ACM, 2005.

- [48] D. Mehta and M. R. C. van Dongen. Reducing checks and revisions in coarse-grained mac algorithms. In *International Joint Conference On Artificial Intelligence (IJCAI)*, volume 19, pages 236–241, 2005.
- [49] D. Mehta and M. R. C. van Dongen. Static value ordering heuristics for constraint satisfaction problems. In M. R. C. van Dongen, editor, *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, pages 49–62, 2005.
- [50] D. Mehta and M. R. C. Van Dongen. Probabilistic consistency boosts mac and sac. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 143–148, 2007.
- [51] A. Meisels. Message delays and discsp search algorithms. In *Distributed Search by Constrained Agents*, pages 143–158. Springer, 2008.
- [52] A. Meisels and O. Lavee. Using additional information in discsps search. In *In DCR*, 2004.
- [53] I. Miguel and Q. Shen. Dynamic flexible constraint satisfaction. *Appl. Intell.*, 13(3) :231–245, 2000.
- [54] S. Minton, A. Philips, M. D. Johnston, and P. Laird. Minimizing conflicts : A heuristic repair method for constraint-satisfaction and scheduling problems. *Journal of Artificial Intelligence Research*, 1 :1–15, 1993.
- [55] R. Mohr and T. C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28(2) :225 – 233, 1986.
- [56] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Sciences*, 7 :95 – 132, 1974.
- [57] B. Omomowo, I. Arana, and H. Ahriz. Dynabt : Dynamic asynchronous backtracking for dynamic discsps. In *Artificial Intelligence : Methodology, Systems, and Applications*, pages 285–296. Springer, 2008.
- [58] R. Precup, S. Preitl, and P. Korondi. Fuzzy controllers with maximum sensitivity for servosystems. *IEEE Transactions on Industrial Electronics*, 54(3) :1298–1310, 6 2007.
- [59] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, pages 268–299, 1993.
- [60] Y. Ran, N. Roos, and H. Herik. Approaches to find a near-minimal change solution for dynamic csps. In *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 373–387, 2002.
- [61] J. Reinfelds. Programming as an engineering discipline. In *Frontiers in Education, 2002. FIE 2002. 32nd Annual, IEEE*, 2002.

- [62] N. Sadeh and M. S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artif. Intell.*, 86(1) :1–41, Sept. 1996.
- [63] M. A. Salido. A non-binary constraint ordering heuristic for constraint satisfaction problems. *Applied Mathematics and Computation*, 198(1) :280–295, 2008.
- [64] M. A. Salido and F. Barber. Constrainedness and redundancy by constraint ordering. In *Advances in Artificial Intelligence - IBERAMIA 2004, 9th Ibero-American Conference on AI, Puebla, México, November 22-26, 2004, Proceedings*, pages 124–133, 2004.
- [65] M. A. Salido and A. Giret. Feasible distributed csp models for scheduling problems. *Engineering Applications of Artificial Intelligence*, 21(5) :723–732, 2008.
- [66] S. Sen and E. H. Durfee. Unsupervised surrogate agents and search bias change in flexible distributed scheduling. In *In Proceedings of the First International Conference on Multi-Agent Systems*, pages 336–343. AAAI Press, 1995.
- [67] B. Smith. Object-oriented programming. *AdvancED ActionScript 3.0 : Design Patterns*, Springer, pages 1–25, 2011.
- [68] K. Stergiou. Restricted path consistency revisited. In *Principles and Practice of Constraint Programming (CP)*, pages 419–428, 2015.
- [69] Ö. Türkşen and M. Tez. An application of nelder-mead heuristic-based hybrid algorithms : Estimation of compartment model parameters. *International Journal of Artificial Intelligence*, 14(1) :112–129, 2016.
- [70] W. J. van Hoeve and J. N. Hooker, editors. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 6th International Conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27-31, 2009, Proceedings*, volume 5547 of *Lecture Notes in Computer Science*. Springer, 2009.
- [71] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *Association for the Advancement of Artificial Intelligence (AAAI)*, pages 307–312, 1994.
- [72] M. Wahbi. *Agile Asynchronous Backtracking (Agile-ABT)*, pages 111–130. John, Wiley and Sons, Inc., 2013.
- [73] M. Wahbi, R. Ezzahir, C. Bessiere, and E. H. Bouyakhf. Dischoco 2 : A platform for distributed constraint reasoning. In *Proceedings of the IJCAI’11 workshop on Distributed Constraint Reasoning, DCR’11*, pages 112–121, 2011.
- [74] R. Wallace and E. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Ninth Canadian Conference on Artificial Intelligence*, pages 163–169, 1992.

- [75] R. J. Wallace. Sac and neighbourhood sac. 2012.
- [76] R. J. Wallace. Complexity analysis vs. engineering design in CSP algorithms : Contravening conventional wisdom again. In *23rd International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA)*, pages 73–87, 2016.
- [77] R. J. Wallace. Preprocessing versus search processing for constraint satisfaction problems. In *23rd International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (RCRA) A workshop of the International Conference of the Italian Association for Artificial Intelligence (AI*IA)*, pages 89–103, 2016.
- [78] R. J. Wallace and E. C. Freuder. Stable solutions for dynamic constraint satisfaction problems. In *Principles and Practice of Constraint Programming (CP)*, pages 447–461, 1998.
- [79] R. J. Wallace and E. C. Freuder. Constraint-based multi-agent meeting scheduling : effects of agent heterogeneity on performance and privacy loss. In *Proceeding of the 3rd workshop on distributed constraint reasoning ,DCR’02*, pages 176–182, 2002.
- [80] R. J. Wallace, D. Grimes, and E. C. Freuder. Solving dynamic constraint satisfaction problems by identifying stable features. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 621–627, 2009.
- [81] R. J. Wallace, D. Grimes, and E. C. Freuder. Solving dynamic constraint satisfaction problems by identifying stable features. In *IJCAI*, volume 9, pages 621–627. Citeseer, 2009.
- [82] R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI’03*, pages 1173–1178, 2003.
- [83] M. Yokoo. *Distributed constraint satisfaction : foundations of cooperation in multi-agent systems*. Springer Science & Business Media, 2012.
- [84] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem : Formalization and algorithms. *IEEE Transactions on knowledge and data engineering*, 10(5) :673–685, 1998.
- [85] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. Distributed stochastic search and distributed breakout : properties, comparison and applications to constraint optimization problems in sensor networks. *Artificial Intelligence*, 161(1) :55–87, 2005.
- [86] W. Zhang, Z. Xing, G. Wang, and L. Wittenburg. An analysis and application of distributed constraint satisfaction and optimization

- algorithms in sensor networks. In *AAMAS*, volume 3, pages 185–192, 2003.
- [87] R. Zivan and A. Meisels. Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, 11(2-3) :179–197, 2006.

Résumé

Dans cette thèse, nous nous intéressons au traitement des problèmes dynamiques par le paradigme CP (Constraint Programming), notamment les DCSPs (Dynamic Constraint Satisfaction Problems). L'objectif de notre travail est de proposer des heuristiques et méthodes pour réparer les DCSPs dans le but de réduire l'effort de la réparation.

Notre thèse est constituée de trois axes:

Le premier consiste à concrétiser notre travail, en proposant la modélisation du problème d'affectation des stages hospitaliers sous forme de CSP (Constraint Satisfaction Problem). Le deuxième est consacré à proposer des approches de réparation de DCSPs ainsi que des heuristiques d'ordonnancement.

Les approches proposées Extended PDB (EPDB), Lazy Repairing Backtracking (LRB) et Improved PDB (IPDB) améliorent le comportement de PDB (Partial-order Dynamic Backtracking), une méthode connue de réparation de DCSPs.

Ensuite, nous proposons des heuristiques DCO (Dynamic Constraint Ordering), dont le but est d'adopter une stratégie d'ordre dont les contraintes sont testées. Aussi, nous proposons l'heuristique PDeg (Profound Degree), qui vise à maintenir la stabilité des parties les plus contraignantes du réseau lors de la réparation.

Enfin, le troisième axe concerne les DisCSPs (Distributed Constraint Satisfaction Problems), pour lesquels nous proposons LiveABT (Live Asynchronous Backtracking), une approche de réparation en temps réel.

Abstract

In this thesis, we are interested in the treatment of dynamic problems by the CP (Constraint Programming) paradigm, in particular DCSPs (Dynamic Constraint Satisfaction Problems). The purpose of our work is to propose heuristics and methods for repairing DCSPs in order to reduce the effort of repair.

Our thesis consists of three axes:

The first is to concretize our work by proposing the modeling of the problem of assigning hospital internships as a CSP (Constraint Satisfaction Problem).

The second is dedicated to propose DCSPs repair approaches as well as scheduling heuristics.

The proposed approaches Extended PDB (EPDB), Lazy Repairing Backtracking (LRB), and Improved PDB (IPDB) improve the behavior of PDB (Partial-order Dynamic Backtracking), a known method of repairing DCSPs.

Next, we propose Dynamic Constraint Ordering (DCO) heuristics, whose goal is to adopt an order strategy whose constraints are tested. Also, we propose the PDeg (Profound Degree) heuristic, which aims to maintain the stability of the most restrictive parts of the network during repair.

Finally, the third axis concerns DisCSPs (Distributed Constraint Satisfaction Problems), for which we propose LiveABT (Live Asynchronous Backtracking), a real-time repair approach.