UNIVERSITÉ D'ARTOIS

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

centre de recherche en informatique de lens

Contributions à la résolution générique des problèmes de satisfaction de contraintes

THÈSE

présentée et soutenue publiquement le 30 novembre 2007

en vue de l'obtention du

Doctorat de l'Université d'Artois

(Spécialité Informatique)

par

Julien Vion

Composition du jury

Président :	Éric Grégoire	CRIL
Rapporteurs :	Philippe Jégou	LSIS
	Bertrand Neveu	INRIA
Examinateurs :	Chu Min Li	LaRIA
Directeurs de recherche :	Christophe Lecoutre	CRIL
	Lakhdar Saïs	CRIL

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS – CNRS FRE 2499
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.univ-artois.fr

Mise en page avec memcril (B. Mazure, CRIL) et thloria (D. Roegel, LORIA).

Remerciements

Je tiens à remercier toutes les personnes qui m'ont suivi et accompagné tout au long de ma thèse, et en premier lieu Christophe Lecoutre et Lakhdar Saïs, ainsi que Stéphane Cardon. Je leur dois à tous les meilleures contributions de cette thèse.

Je remercie également chaleureusement les rapporteurs de mon travail, Philippe Jégou et Bertrand Neveu, ainsi que Chu Min Li pour avoir accepté de participer au jury et de consacrer du temps à lire ce manuscrit.

Merci à tous les membres du CRIL et personnels enseignant et administratifs du département Informatique de l'IUT de Lens pour leur présence chaleureuse jour après jour. Merci particulièrement à Éric Grégoire pour m'avoir accueilli dans « son équipe », pour sa participation au jury, et pour m'avoir permis de participer à de nombreuses conférences nationales et internationales qui furent autant d'expériences particulièrement formatrices au métier de chercheur.

Merci également à Patrick Taillibert et son équipe de recherche, pour m'avoir fait découvrir et donné goût à la recherche scientifique. J'espère qu'il aura l'occasion de lire et apprécier cette thèse.

Merci à mes parents et ma famille, qui m'ont accompagné et encouragé tout au long de mon parcours d'études, et finalement, un merci tout particulier à Anne la B., pour son soutien indéfectible et sans égal dans les moments les plus difficiles !

Table des matières

Bibliographie

Liste des Algorithmes

Introduction

De très nombreux problèmes logiques, d'aide à la décision ou issus de l'ingénierie des systèmes sont des problèmes combinatoires. Il s'agit de problèmes de décision ou d'optimisation de nature exponentielle : Si on étend linéairement la taille du problème (par exemple, en ajoutant une variable ou une valeur), on multiplie le nombre d'hypothèses à envisager pour résoudre le problème. Ils sont donc extrêmement difficiles à résoudre, pour un humain comme pour un ordinateur, et nécessitent l'utilisation de techniques avancées d'intelligence artificielle ou de recherche opérationnelle pour être résolus par un ordinateur.

La programmation par contraintes (PC) a été développée dès les années 70 afin de réussir à formaliser facilement et à résoudre informatiquement de tels problèmes. Les langages de programmation logiques, comme Prolog, ou fonctionnels, comme Lisp, sont des langages déclaratifs, par opposition aux langages impératifs classiques comme Ada ou C. L'objectif derrière ces langages de programmation déclaratifs est de définir une méthode de programmation « intelligente », où un utilisateur non expert n'aurait qu'à définir sa problématique à l'ordinateur et obtenir immédiatement une solution. Les domaines scientifiques ayant trait au développement de tels systèmes ont été depuis regroupés sous l'appellation « Intelligence Artificielle ».

Le problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) se situe au cœur de la problématique de la programmation par contraintes. Étant donné un problème combinatoire quelconque formalisé sous forme d'un réseau de contraintes et de variables, l'objectif est de déterminer s'il existe une solution au problème, c'est à dire une instanciation d'une valeur à chaque variable de sorte que toutes les contraintes soient satisfaites. Le problème de satisfaction de contraintes reste un des problèmes combinatoires de référence. Il appartient à la classe de complexité NP, ce qui signifie que l'on conjecture très fortement qu'il n'existe pas d'algorithme capable de résoudre ce problème en temps polynomial, mais qu'il faut obligatoirement utiliser des algorithmes dont les complexités en temps et/ou en espace évoluent exponentiellement avec la taille du problème. Le problème de satisfaction de contrainte est aussi NP-complet, ce qui signifie qu'il capture toute l'essence des problèmes combinatoires. Tout problème NP (qui inclut les classes de complexités plus « faciles ») peut être formalisé sous forme d'un CSP. En améliorant les techniques de résolution pratiques des CSP, on améliore ainsi notre capacité à résoudre tout problème combinatoire. Ces méthodes sont d'ores et déjà utilisées dans de nombreux domaines industriels, comme la planification, l'ordonnancement, le diagnostic, le contrôle d'erreurs dans les circuits électroniques, etc.

Deux approches sont couramment utilisées pour résoudre les problèmes de satisfaction de contraintes : l'inférence et la recherche. Les méthodes d'inférence utilisent des propriétés du réseau de contraintes, appelées consistances, pour déduire des informations complémentaires, notamment des valeurs ou des multiplets de valeurs ne pouvant conduire à une solution. Ces méthodes permettent en particulier de réduire fortement la taille du problème, afin de faciliter le travail de recherche, systématique (exponentielle) ou incomplète. Les méthodes de recherche tentent de minimiser l'espace à explorer afin de découvrir une solution, ou de prouver l'absence de solution à un problème. On utilise pour cela essentiellement des heuristiques, c'est à dire des méthodes capables d'évaluer la pertinence des différentes

hypothèses qui seront effectuées au cours d'une recherche. La recherche peut être systématique, lorsque l'on essaie d'explorer totalement les conséquences de l'attribution d'une valeur à une variable. Une autre approche possible serait d'affecter une valeur à chaque variable sans se soucier des contraintes, puis ensuite d'essayer d'améliorer (*réparer*) la solution jusqu'à ce que toutes les contraintes soient satisfaites. Les réparations sont alors faites sans ordre précis, laissant ainsi un maximum de liberté de choix aux heuristiques, mais en perdant la certitude d'atteindre une solution et d'en prouver l'absence.

Dans le premier chapitre de ce document, on trouvera la formalisation du problème de satisfaction de contraintes, ainsi que plusieurs exemples de problèmes simples, une définition de nos notations et enfin un rappel des principales propriétés qui seront exploitées dans cette thèse. Nous développons ensuite un certain nombre d'améliorations aux différentes techniques d'inférence et de recherche précedemment décrites.

Le deuxième chapitre présente les contributions que nous avons développées sur l'inférence, toutes basées sur la consistance d'arc. Nous avons d'abord cherché à améliorer les algorithmes d'établissement de la consistance d'arc eux-mêmes. Nous sommes ainsi parvenus, dans le cas des contraintes binaires, à une optimisation simple de l'algorithme qui semble le plus rapide aujourd'hui, AC-3^{rm}. Grâce à l'utilisation des opérations bit-à-bit, nous pouvons profiter au maximum des performances des microprocesseurs des ordinateurs. Afin de traiter efficacement les problèmes impliquant des domaines de très grande taille (en particulier les problèmes d'ordonnancement et de planification), nous avons ensuite étudié les propriétés de différentes consistances de domaine bien connues (consistance d'arc, singleton consistance d'arc ainsi que la consistance de chemin restreinte) lorsque leur effet est limité aux bornes des domaines. Nous donnons ici des résultats théoriques ainsi que plusieurs résultats expérimentaux sur diverses séries de problèmes industriels. Finalement, nous décrivons les travaux que nous avons effectués autour d'une nouvelle forme de consistance nommée la consistance duale. Définie à partir des propriétés de la consistance d'arc, celle-ci est équivalente à une autre propriété largement étudiée par la communauté scientifique : la consistance de chemin. À la lumière de cette nouvelle définition, nous proposons plusieurs algorithmes permettant d'établir la consistance duale (et donc la consistance de chemin), et montrons qu'ils sont en pratique plus rapides que les meilleurs algorithmes établissant la consistance de chemin. La consistance de chemin a le défaut bien connu de nécessiter de modifier la structure du graphe en ajoutant un grand nombre de contraintes pour être établie, pour un coût très important tant en temps qu'en espace. Pour limiter cet inconvénient majeur, une variante limitant l'action de la consistance de chemin aux contraintes déjà présentes dans le réseau initial avait déjà été proposée sous le nom de « chemin consistance conservative ». Nous avons établi formellement que la variante conservative de la consistance duale est strictement plus forte que la consistance de chemin conservative, et l'algorithme établissant la consistance duale conservative reste également plus rapide en pratique que les algorithmes établissant la consistance de chemin conservative.

Le troisième chapitre est dédié aux travaux que nous avons réalisés autour des méthodes de recherche. Notre étude s'est orientée selon deux axes. Tout d'abord, nous avons cherché à équiper l'algorithme systématique MAC d'une heuristique de choix de valeurs, sujet peu abordé dans la littérature. Pour cela, nous sommes partis des heuristiques utilisées pour résoudre le problème SAT, un autre problème NP-complet similaire aux CSP. Il existe plusieurs simples conversions de CSP vers SAT. En combinant la formulation générale de l'heuristique de branchement de Jeroslow-Wang « à deux faces » avec deux techniques de codage d'un CSP en problème SAT, nous retrouvons deux des heuristiques de choix de valeurs existantes pour les CSP, et nous en déduisons de nouvelles. Ces nouvelles heuristiques exploitent la bidirectionnalité des contraintes et les propriétés des branchements binaires effectués par l'algorithme MAC. Finalement, nous avons étudié une simple hybridation entre un algorithme de recherche locale travaillant par pondération de contraintes (méthode Breakout) et l'algorithme systématique MGAC. En faisant correspondre les pondérations obtenues par la recherche locale avec les pondérations utilisées par l'heuristique de choix de valeurs $dom/wdeg$ d'une part, et en enregistrant les réfutations effectuées

par MGAC d'autre part, nous obtenons un algorithme hybride simple mais capable de résoudre plus de problèmes que MGAC.

De manière transversale, l'ensemble des techniques développées au cours de cette thèse ont été implantées dans une nouvelle bibliothèque (API pour *Application Programming Interface* ou Interface de Programmation) de résolution de CSP pour les applications Java. Cette API est décrite dans le quatrième et dernier chapitre de cette thèse. Nous décrivons notamment l'utilisation de l'API pour résoudre des problèmes d'Open Shop ainsi que pour extraire des noyaux minimalement insatisfiables à partir de CSP surcontraints.

Chapitre 1

Introduction aux Problèmes de Satisfaction de Contraintes

Ce chapitre introduit les notions formelles de variables, contraintes, réseaux de contraintes et problèmes de satisfaction de contraintes, en indiquant les notations et les propriétés de base. Nous donnons plusieurs exemples de problèmes simples, d'origine industrielle ou académique, avant de rappeler les notions de complexité algorithmique qui sont au cœur des problématiques d'intelligence artificielle.

Nous effectuons ensuite un rapide survol des principales propriétés et algorithmes permettant de résoudre en pratique le problème de satisfaction de contraintes, et en particulier la consistance d'arc (AC) et l'algorithme systématique maintenant la consistance d'arc pendant la recherche (MAC). Nous indiquons également les propriétés de consistance nous permettant d'effectuer des inférences sur les réseaux de contraintes, et des notions sur la recherche locale pour la satisfaction de contraintes. Ces propriétés seront développées dans la suite de cette thèse.

1.1 Problèmes de satisfaction de contraintes

Les problèmes de satisfaction de contraintes (Constraint Satisfaction Problems ou CSP) sont au cœur de la programmation par contraintes. Le problème à résoudre est formalisé par un ensemble de variables et un ensemble de contraintes formant un *réseau de contraintes* (Constraint Network ou CN). L'objectif du problème est de déterminer s'il existe une valeur pour chaque variable telle que toutes les contraintes soient satisfaites. Les variables peuvent être définies sur un domaine continu ($\subseteq \mathbb{R}$) ou discret ($\subseteq \mathbb{Z}$). Les contraintes peuvent impliquer un nombre arbitraire de variables.

Un CSP est avant tout un problème de décision. La réponse attendue est soit « **vrai** », si il existe une solution, soit « **faux** » si aucune solution n'existe. Cependant, en pratique, c'est généralement la découverte d'une solution valable qui est recherchée. Des variantes du problème de base existent, qui sont davantage tournées vers l'optimisation : par exemple, le problème Max-CSP consiste à trouver une solution optimale, qui satisfait le plus de contraintes possible. Une fonction d'optimisation peut également être associée à un CSP simple. L'objectif est alors de trouver une solution valide qui minimise ou maximise le résultat de la fonction d'optimisation.

1.1.1 Formalisation

Par la suite, on s'intéressera aux CSP discrets, c'est à dire aux CSP dont le domaine des variables consiste en un ensemble fini de valeurs ($\subseteq \mathbb{Z}$). Pour chacune des notations introduites dans cette section, on trouvera un exemple basé sur un réseau de contraintes simple au début de la section 1.1.2.

Définition 1 (Réseau de contraintes)**.** Un réseau de contraintes (*Constraint Network* ou CN) est un couple $(\mathscr{X},\mathscr{C})$ où :
- $\mathscr{X}$ est un ensemble fini de variables ; À chaque variable X est associé un domaine $\mathrm{dom}^P(X)$, indiquant l'ensemble de valeurs autorisées pour X,
- $\mathscr{C}$ est un ensemble fini de contraintes.

Le domaine d'une variable X du réseau P est noté $\mathrm{dom}^P(X)$ mais quand ce sera possible sans ambiguité, on notera $\mathrm{dom}(X)$ au lieu de $\mathrm{dom}^P(X)$.

Chaque contrainte met en jeu un nombre quelconque de variables. L'ensemble des variables impliquées par une contrainte (le *scope* de la contrainte) sera noté $\mathrm{vars}(C)$. Le nombre de variables impliquées par une contrainte, $|\mathrm{vars}(C)|$, est appelé l'arité de la contrainte.

Définition 2 (Instanciation)**.** Étant donné un CN $P = (\mathscr{X}, \mathscr{C})$, une instanciation I d'un ensemble de variables distinctes notées $\mathrm{vars}(I)$, tel que $\mathrm{vars}(I) \subseteq \mathscr{X}$, est un ensemble de couples où à chaque variable $X \in S$ est associée une valeur $a \in \mathrm{dom}(X)$.

On notera X_a un couple (X, a), $X_a \in I$ si le couple (X, a) apparaît dans l'instanciation I et $X \in \mathrm{vars}(I)$ si X apparaît dans l'un des couples de l'instanciation I. La longueur d'une instanciation I correspond au nombre d'éléments de I.

Définition 3 (Contrainte)**.** Une contrainte portant sur les variables $\mathrm{vars}(C)$ est une application de l'ensemble des instanciations telles que $\mathrm{vars}(C) = \mathrm{vars}(I)$ dans le domaine des booléens $\{$**vrai**, **faux**$\}$. À chaque contrainte C du CN P sont associées deux relations : $\mathrm{rel}^P(C)$ indique l'ensemble des instanciations de $\mathrm{vars}(C)$ autorisées par la contrainte et $\mathrm{cfl}^P(C)$ l'ensemble (complément) des instanciations de $\mathrm{vars}(C)$ interdites par la contrainte.

De même que pour $\mathrm{dom}(X)$, on notera $\mathrm{rel}(C)$ au lieu de $\mathrm{rel}^P(C)$ et $\mathrm{cfl}(C)$ au lieu de $\mathrm{cfl}^P(C)$ quand ce sera possible sans ambiguité.

On dira qu'une instanciation I est consistante ssi toutes les contraintes C telles que $\mathrm{vars}(C) \subseteq \mathrm{vars}(I)$ sont satisfaites par l'instanciation. On notera que si une instanciation I n'est pas consistante, alors toutes les instanciations I' telles que $I \subseteq I'$ seront également inconsistantes. Une contrainte C est aussi satisfaite par une instanciation I (telle que $\mathrm{vars}(C) \subseteq \mathrm{vars}(I)$) si et seulement si $\exists J \in \mathrm{rel}(C) \mid J \subseteq I$.

Une contrainte peut être définie « en intention » par une formule mathématique. En pratique, les contraintes sont souvent définies comme une application du produit cartésien du domaine des variables $\mathrm{vars}(C)$ vers les booléens. L'ensemble $\mathrm{vars}(C)$ étant ordonné, la contrainte peut alors être définie « en extension » par une liste des multiplets autorisés ou interdits. Par exemple, si une contrainte porte sur les variables (X, Y), alors l'instanciation $\{X_a, Y_b\}$, est autorisée par la contrainte ssi elle autorise le multiplet (a, b). On pourra alors noter $(a, b) \in \mathrm{rel}(C)$.

Un réseau normalisé est un réseau dans lequel il n'existe pas deux contraintes portant sur un même ensemble de variables ($\nexists (C_1, C_2) \in \mathscr{C}^2 \mid C_1 \neq C_2 \wedge \mathrm{vars}(C_1) = \mathrm{vars}(C_2)$). On considèrera en général uniquement les réseaux normalisés sans perte de généralité, puisqu'il est toujours possible de regrouper les contraintes de même *scope* par conjonction.

Le problème de satisfaction de contraintes (*Constraint Satisfaction Problem* ou CSP) consiste à déterminer si une solution au CN existe. Une solution est une instanciation de l'ensemble des variables du CN telle que toutes les contraintes soient satisfaites. Un CN n'admettant pas de solution est dit incohérent ou insatisfiable. Un CN admettant au moins une solution est dit cohérent ou satisfiable. Il s'agit d'un problème NP-complet (voir section 1.1.4).

Définition 4 (Contrainte, CN binaires)**.** Une contrainte binaire est une contrainte impliquant exactement deux variables. Un CN $P = (\mathscr{X}, \mathscr{C})$ est binaire quand il implique uniquement des contraintes binaires : $\forall C \in \mathscr{C}, |\mathrm{vars}(C)| = 2$.

Un CN peut être caractérisé par plusieurs grandeurs : le nombre de variables et la taille maximale de leurs domaines, le nombre de contraintes ou la densité du CN dans le cas d'un CN binaire, la dureté (*tightness*) des contraintes. La dureté des contraintes est surtout prise en compte dans les problèmes aléatoires afin de mettre en évidence le phénomène de seuil.

Nous utiliserons les notations suivantes :

- n : le nombre de variable du problème
- d : la taille du plus grand domaine
- e : le nombre de contraintes
- k : l'arité maximale des contraintes
- D : la densité d'un CN binaire ($e/C_n^2 = 2e/(n^2 - n)$)
- t : la dureté d'une contrainte, définie comme le rapport entre $|\mathrm{cfl}(C)|$ et le nombre d'instanciations possibles de $\mathrm{vars}(C)$ ($\prod_{X \in \mathrm{vars}(C)} |\mathrm{dom}(X)|$).
- $\Gamma(X)$: l'ensemble des contraintes impliquant X : $\Gamma(X) = \{C \in \mathscr{C} \mid X \in \mathrm{vars}(X)\}$. $|\Gamma(X)|$ correspond au degré de X.

Dans un réseau binaire, $k = 2$. Un réseau binaire est complet quand $D = 100\%$. On appellera parfois « graphe de contraintes » un réseau binaire. En effet, un réseau binaire de contraintes est généralement représenté sous forme de graphe, les variables formant les nœuds du graphe et les contraintes binaires les arêtes (non orientées). Un graphe de contraintes simple est donné en exemple dans la section 1.1.2.

Définition 5 (Nogood)**.** Un nogood est une instanciation identifiée comme invalide, c'est à dire ne pouvant apparaître dans aucune solution. En particulier, toutes les instanciations non autorisées par une contrainte sont des nogoods.

Il existe plusieurs notions d'équivalence entre deux réseaux. La plus directe consiste à dire que deux réseaux sont équivalents si ils admettent le même ensemble de solutions. Toutefois, il est plus utile de définir une notion élargie d'équivalence entre deux réseaux : deux réseaux sont équivalents si on peut déduire les solutions de l'un à partir des solutions de l'autre en temps polynomial. Tout CN peut être ainsi ramené à un réseau binaire en temps et en espace polynomiaux [ROSSI *et al.* 1990,].

Théorème 1. Pour tout CN n-aire, il existe un CN binaire tel que les solutions du CN n-aire peuvent être déduites des solutions du CN binaire.

Il est donc fréquent lors de l'étude théorique des algorithmes de se limiter aux réseaux binaires. Cependant, trouver le CN binaire équivalent (au sens élargi) n'est pas toujours envisageable en pratique, et les algorithmes pouvant traiter les CN n-aires sont généralement préférés [BACCHUS ET VAN BEEK 1998]. Il existe en pratique plusieurs techniques permettant de convertir un CN n-aire en CN binaire :

- La **conversion par réseau dual** consiste à considérer un graphe dans lequel chaque contrainte du réseau initial P_{init} est représentée par une variable (dénommée c-variable) dans le réseau dual P_{dual}. Chaque valeur de chaque c-variable correspond à une instanciation (de longueur $|\mathrm{vars}(C)|$) autorisée par la contrainte C correspondante dans P_{init}. Les contraintes binaires de P_{dual} interdisent alors de choisir deux instanciations (postant sur les c-variables) correspondant à des instanciations de P_{init} conflictuelles (deux instanciations I_1 et I_2 sont conflictuelles ssi $\exists (X_a, Y_b) \in I_1 \times I_2 \mid X = Y \wedge a \neq b$).
- La **conversion par variables cachées** consiste à introduire des variables supplémentaires (dénommées h-variables) en complément des variables initiales du réseau pour chaque contrainte n-aire du réseau initial. Chaque valeur des h-variables correspond à une instanciation (de longueur $|\mathrm{vars}(C)|$) autorisé de la contrainte C correspondante. Les contraintes binaires font alors le lien entre chaque valeur des h-variables et l'instanciation correspondante de la variable du réseau initial. Toutes les contraintes non-binaires du réseau initial sont supprimées et remplacées de manière équivalente par l'ensemble des h-variables et des contraintes binaires introduites.

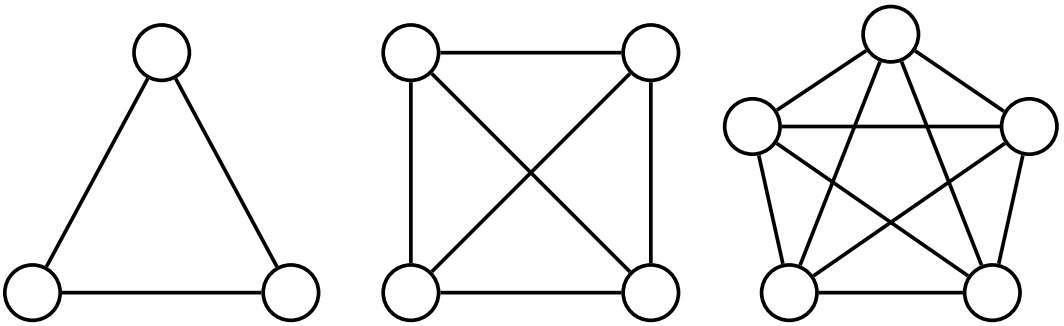


FIGURE 1.1 – Une 3-clique, 4-clique et 5-clique

Ces deux conversions peuvent introduire des c-variables ou h-variables de domaine très élevé (d^k) si l'arité des contraintes du graphe initial est importante et la dureté de celles-ci faible. Il n'est donc pas toujours possible en pratique de convertir un réseau n-aire en réseau binaire.

Définition 6 (Contrainte universelle)**.** Une contrainte universelle est une contrainte qui autorise toutes les instanciations possibles ($\mathrm{cfl}(C) = \emptyset$).

Une contrainte universelle est toujours satisfaite, quelle que soit l'instanciation courante des variables qu'elle implique. Pour une contrainte universelle, $t = 0$. Ces contraintes sont en pratique inutiles pour résoudre un CSP et peuvent être retirées du réseau. On peut aussi considérer que les variables qui ne sont reliées par aucune contrainte sont reliées par une contrainte universelle. Si une seule contrainte interdit tous les multiplets possibles pour les variables qu'elle implique (pour une telle contrainte, $t = 1$) le réseau est insatisfiable.

Définition 7 (Supports et Conflits)**.** Les supports $\mathcal{S}pI(C, X_a)$ avec $X \in \mathrm{vars}(C)$ d'une valeur dans une contrainte d'arité quelconque est l'ensemble des instanciations de longueur $|\mathrm{vars}(C)|$ satisfaisant C :
$\mathcal{S}pI(C, X_a) = \{I \in \mathrm{rel}(C) \mid X_a \in I\}$
Les conflits $\mathcal{C}fI(C, X_a)$ avec $X \in \mathrm{vars}(C)$ correspondent à l'ensemble :
$\mathcal{C}fI(C, X_a) = \{I \in \mathrm{cfl}(C) \mid X_a \in I\}$

Dans le cas des contraintes binaires, on définit la notion de *valeur support* : les valeurs supports d'une valeur X_a dans une contrainte binaire C telle que $\mathrm{vars}(C) = \{X, Y\}$, sont l'ensemble $\mathcal{S}pV(C, X_a)$ des valeurs Y_b ($b \in \mathrm{dom}(Y)$) telles que $\{X_a, Y_b\} \in \mathcal{S}pI(C, X_a)$. On définit de même les *valeurs conflit*, notées $\mathcal{C}fV(C, X_a)$.

Définition 8 (Clique)**.** Dans un graphe G constitué d'un ensemble de nœuds $\mathscr{N}$ et d'arêtes $\mathscr{A}$, une clique est un ensemble de nœuds $\mathscr{Q} \subseteq \mathscr{N}$ tel que chaque couple de nœuds de $\mathscr{Q}$ soit relié par une arête.

On définit de la même manière une clique de variables $\mathscr{Q}$ dans un graphe de contraintes $P = (\mathscr{X}, \mathscr{C}) : \forall (X, Y) \in \mathscr{Q}^2 \mid X \neq Y, \exists C \in \mathscr{C} \mid \mathrm{vars}(C) = \{X, Y\}$

Définition 9 (Graphe complet)**.** Un graphe est complet quand l'ensemble de ses nœuds forme une clique.

La figure 1.1 montre des cliques de trois à cinq variables. On appellera une clique de k variables une k-clique. Dans un réseau binaire complet, $e = C_n^2 = \frac{n(n-1)}{2}$ et $D = 100\%$.

Définition 10 (Ordre sur les réseaux)**.** Un réseau de contraintes $P' = (\mathscr{X}', \mathscr{C}')$ est dit plus petit qu'un autre réseau $P = (\mathscr{X}, \mathscr{C})$, noté $P' \leq P$, ssi chacune des conditions suivantes est vérifiée :
– $\mathscr{X}' = \mathscr{X}$

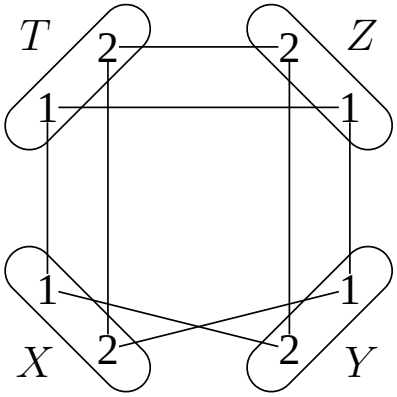


FIGURE 1.2 – Un réseau de contraintes simple

– $\forall X \in \mathscr{X}, \mathrm{dom}^{P'}(X) \subseteq \mathrm{dom}^{P}(X)$
– $\mathscr{C}' = \mathscr{C}$
– $\forall C \in \mathscr{C}, \mathrm{rel}^{P'}(C) \subseteq \mathrm{rel}^{P}(C)$

P' est strictement plus petit que P, noté $P' < P$, ssi $P' \leq P$ et :
– $\exists X \in \mathscr{X} \mid \mathrm{dom}^{P'}(X) \subset \mathrm{dom}^{P}(X)$,
– ou $\exists C \in \mathscr{C} \mid \mathrm{rel}^{P'}(C) \subset \mathrm{rel}^{P}(C)$.

Test de contrainte : Les algorithmes travaillant sur les CSP sont amenés à tester les contraintes à de nombreuses reprises. Un test de contrainte consiste à vérifier si une instanciation satisfait une contrainte, soit en pratique à calculer le résultat de la fonction d'un des éléments du produit cartésien du domaine des variables impliquées par la contrainte. Il s'agit de l'opération de base de tout algorithme travaillant sur un CSP, qu'il soit complet ou incomplet. On peut noter que le coût en terme d'opérations CPU nécessaires pour tester une contrainte est très variable en fonction de la nature de celle-ci. Les contraintes implicites peuvent représenter des conditions très complexes, nécessitant de nombreux calculs pour déterminer si la contrainte est satisfaite ou non. Les contraintes en extension, représentées sous forme de matrice de booléens, sont généralement considérées comme les contraintes les plus rapides à tester, mais ne sont envisageables que pour des arités faibles. Les algorithmes cherchent généralement à minimiser le nombre de tests de contraintes à effectuer pour résoudre le problème, mais il arrive fréquemment que la diminution du nombre de tests de contraintes entraine une complexification des algorithmes, et par là même de l'apparition de calculs supplémentaires ou de gestion de structures de données au sein des algorithmes [VAN DONGEN 2004].

1.1.2 Exemples de problèmes

Un CN simple

Pour illustrer les notions introduites, considérons le réseau simple représenté figure 1.2. Il s'agit d'un réseau binaire comportant quatre variables X, Y, Z et T admettant chacune deux valeurs (1 ou 2), et quatre contraintes C_{XY}, C_{YZ}, C_{TZ} et C_{TX}. Les arcs représentent les multiplets autorisés par les contraintes. Si il n'y a aucun arc entre deux variables, on considère qu'il n'y a pas de contraintes (par la suite, on utilisera toujours cette représentation). Ici, il n'y a pas de contrainte entre X et Z ni entre Y et T. C_{YZ}, C_{TZ} et C_{TX} sont des contraintes d'égalité ($Y = Z$, $T = Z$ et $T = X$), et C_{XY} est une

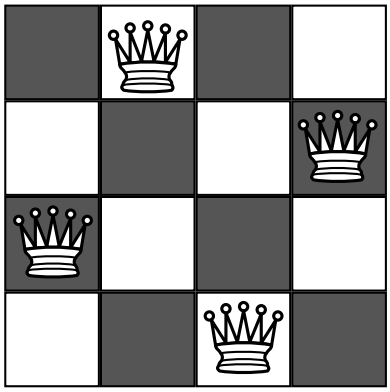


FIGURE 1.3 – Une solution au problème des 4 dames

contrainte de différence ($X \neq Y$). Ce réseau est insatisfiable et admet les caractéristiques suivantes : $n = 4$, $d = 2$, $e = 4$, $k = 2$, $D = 4/6 = 67\%$.

Pour chaque contrainte de ce graphe, il existe un ensemble de quatre instantiations locales pouvant valider ou non la contrainte. Par exemple, C_{XY}, qui porte sur les variables (X, Y), peut être calculée pour l'ensemble de multiplets suivant : $\{(1,1),(1,2),(2,1),(2,2)\}$, correspondant aux instanciations $\{(X_1,Y_1),(X_1,Y_2),(X_2,Y_1),(X_2,Y_2)\}$. Chacune des contraintes autorise deux multiplets et en interdit deux autres. Par exemple, C_{XY} autorise les instanciations suivantes : $\mathrm{rel}(C_{XY}) = \{\{X_1,Y_2\},\{X_2,Y_1\}\}$ (ainsi que toute instanciation I de longueur supérieure à deux telle qu'une des instanciations de $\mathrm{rel}(C_{XY})$ soit incluse dans I). La dureté de chaque contrainte vaut donc $t = 50\%$.

L'ensemble des instanciations de longueur 2 supportant X_1 dans C_{XY} est $\mathcal{S}pI(C_{XY}, X_1) = \{\{X_1,Y_2\}\}$. L'ensemble des valeurs support de X_1 dans C_{XY} est donc $\mathcal{S}pV(C_{XY}) = \{Y_2\}$. L'ensemble des valeurs conflit de X_1 dans C_{XY} est : $\mathcal{C}fV(C_{XY}, X_1) = \{Y_1\}$.

Si on remplace la contrainte de différence C_{XY} par une contrainte d'égalité, le réseau devient satisfiable et admet deux solutions : $\{X_1, Y_1, Z_1, T_1\}$ et $\{X_2, Y_2, Z_2, T_2\}$

Les n dames

Les n dames est un puzzle logique très connu, consistant à placer n dames sur un tablier de $n \times n$ cases de telle sorte qu'aucune ne soit en mesure d'en prendre une autre, suivant les règles des échecs (les dames peuvent prendre en ligne droite horizontalement, verticalement et en diagonale suivant un nombre quelconque de cases). Deux conditions sont donc à considérer :

- Deux dames ne peuvent être sur la même ligne.
- Deux dames ne peuvent être sur la même colonne.
- Deux dames ne peuvent être sur la même diagonale.

Ce problème est satisfiable pour $n = 1$ et $n > 3$. La figure 1.3 montre une des deux solutions au problème des 4 dames (l'autre solution s'obtient par réflexion verticale).

En considérant qu'on ne peut placer qu'une dame par ligne, le problème se formalise sous forme de CSP par n variables $X1$ à Xn (une par ligne) de n valeurs (les colonnes). La valeur de chaque variable correspond à la colonne où la dame représentée par la variable doit être placée. La solution représentée par la figure 1.3 est $(2,4,1,3)$. Les contraintes peuvent être définies implicitement par la formule $|Xi - Xj| \neq |i - j| \wedge Xi \neq Xj, \forall (i,j) \in [1..n]^2 \mid i \neq j$. Le réseau ainsi obtenu est complet ($D = 100\%$).

Le problème des n dames est un problème populaire, simple mais non trivial : il est assez difficile à résoudre pour un être humain sur un échiquier ($n = 8$), et nécessite même un nombre important de retour-

arrières quand on essaye de le résoudre avec un algorithme systématique classique (voir section 1.3.1) : plus de 20 000 pour $n = 100$, même avec des heuristiques évoluées. De plus, la question du nombre de solutions distinctes ou symétriques pour un n quelconque reste ouverte. D'un point de vue CSP, la taille du problème devient rapidement très importante : le nombre de variables et la taille des domaines sont directement égaux à n. Le nombre de contraintes dans un graphe complet binaire est quadratique ($C_n^2 = \frac{n(n-1)}{2}$). Le problème est aussi très sous-contraint, et il n'est pas très difficile de trouver une première solution. Il existe d'ailleurs un algorithme simple capable de trouver une première solution en temps linéaire [1]. Les algorithmes de recherche incomplets ont généralement un comportement linéaire sur ce problème, mais ne peuvent trouver qu'une partie des solutions. Énumérer l'ensemble des solutions nécessite un algorithme systématique, et est beaucoup plus long, le nombre de solutions explosant très rapidement (92 solutions pour $n = 8$, 14 200 pour $n = 12$, plus de 3,9.10^{10} pour $n = 20$...)

On notera que, considérant qu'il existe un grand nombre de solutions symétriques au problème des n dames, il est possible de réduire considérablement l'espace de recherche en détectant celles-ci par avance et en indiquant aux algorithmes qu'il est inutile de rechercher une partie des solutions. Cela peut s'obtenir par exemple en limitant le domaine de la variable représentant la première ligne aux $n/2$ premières cases (arrondi au supérieur en cas de n impair). Une fois toutes les solutions obtenues pour les $n/2$ premières cases, l'autre moitié des solutions peut s'obtenir par réflexion verticale. Il existe d'autres symétries inhérentes au problème des n dames, via d'autres réflexions ou rotations.

La détection et la suppressions des symétries de manière générique et efficace dans les CSP reste un problème ouvert et fait l'objet de nombreux travaux de recherche ([BENHAMOU ET SAÏS 1994], par exemple). La recherche de symétries dans un problème combinatoire est équivalente à la résolution d'un problème d'isomorphisme de graphes, qu'on ne sait résoudre que de manière exponentielle.

Coloration de graphes

Le problème de coloration de graphes généralise un grand nombre de problèmes réels de planification et d'ordonnancement. Il consiste à affecter une couleur à chaque nœud d'un graphe sans que deux nœuds voisins soient de la même couleur. On cherche généralement à trouver le nombre de couleurs minimal pour un graphe donné pour que le problème ait une solution. Ce problème remonte à l'origine même de la théorie des graphes. On peut le définir sous forme de CSP tout simplement en reproduisant le graphe à colorer sous forme d'un réseau de contraintes. Toutes les contraintes sont des contraintes de différence.

On peut par exemple utiliser un problème de coloration de graphes pour colorer une carte géographique, comme sur la figure 1.4.

On peut également considérer le problème dual, consistant à colorer les arêtes des graphes de manière à ce que toutes les arêtes partant d'un même nœud soient de couleurs différentes. On doit alors utiliser des contraintes de différence d'arité plus élevée, modélisables par la contrainte globale *all-different* (voir la fin de cette sous-section) ou encore des cliques de contraintes binaires de différence.

Le Problème des Pigeons (Pigeon Hole Problem)

Le problème des pigeons se formule trivialement : comment placer $n+1$ pigeons dans n boîtes, avec au plus un pigeon par boîte ? Ce problème n'a évidemment pas de solution (il est insatisfiable). On peut donc tout à fait le résoudre en temps constant en théorie, mais s'il est naïvement codé sous forme d'un réseau de contraintes binaires (il s'agit alors du problème de coloration de graphes, avec $n+1$ nœuds et n couleurs, les contraintes formant une clique complète sur tous les nœuds) ou sous forme de clauses SAT (voir section 1.1.3), sa résolution devient exponentielle et le problème est alors particulièrement difficile à résoudre.

1. Cf http ://fr.wikipedia.org/wiki/Problème_des_huit_dames

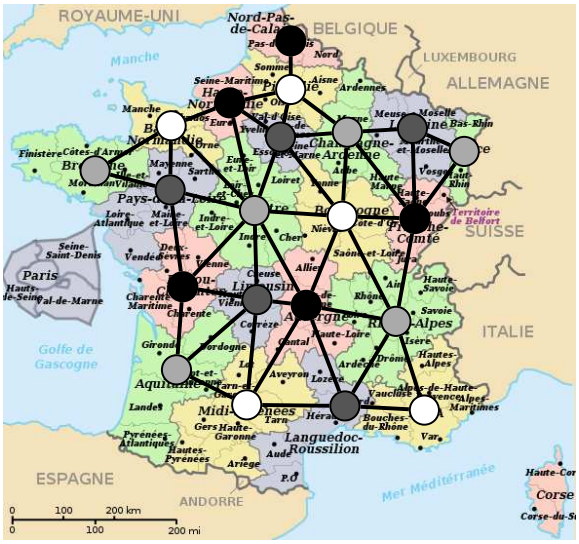


FIGURE 1.4 – La carte de France des régions métropolitaines colorée en 4 couleurs

	Tâche			
Travail	1	2	3	4
J1	54	34	61	2
J2	9	15	89	70
J3	38	19	28	87
J4	95	34	7	29

	Tâche			
Travail	1	2	3	4
J1	M3	M1	M4	M2
J2	M4	M1	M2	M3
J3	M1	M2	M3	M4
J4	M1	M3	M2	M4

TABLE 1.1 – Durée (à gauche) et numéro de machine (à droite) des tâches d'un problème d'atelier 4×4

Certains algorithmes génériques systématiques pour le problème SAT ou CSP implantant une détection des symétries savent cependant résoudre le problème des Pigeons en temps polynomial [BENHAMOU ET SAÏS 1994]. Il est également possible de détecter un « cas de pigeons » avec la contrainte *all-different* : si il reste moins de valeurs possibles que de variables impliquées par la contrainte non instanciées, alors l'instanciation courante de ces variables est insatisfiable. Ces techniques ne semblent malheureusement pas assez performantes à l'heure actuelle pour être appliquées à n'importe quel type de problème dans le cadre d'un prouveur CSP.

Problèmes d'ordonnancement d'atelier

L'ordonnancement d'atelier consiste à organiser dans le temps le fonctionnement d'un atelier pour utiliser au mieux les ressources humaines et matérielles disponibles dans le but de produire les quantités désirées dans le temps imparti. On s'intéressera ici aux problèmes de type Job Shop et Open Shop. Ce sont des généralisations du problème de planification bien connu du Voyageur de Commerce. Ces problèmes ont été longuement étudiés, notamment en Recherche Opérationnelle, et de nombreuses techniques de résolution heuristiques spécifiques ont été développées. Nous étudierons ici leur formalisation sous forme de CSP et leur résolution via des méthodes génériques.

Un problème d'ordonnancement d'atelier consiste en un ensemble de *tâches* de durées définies à réaliser en un temps imparti. Ces tâches utilisent un certain nombre de *ressources* (ou machines) non partageables. L'objectif du problème d'ordonnancement est de trouver une date de début pour chaque

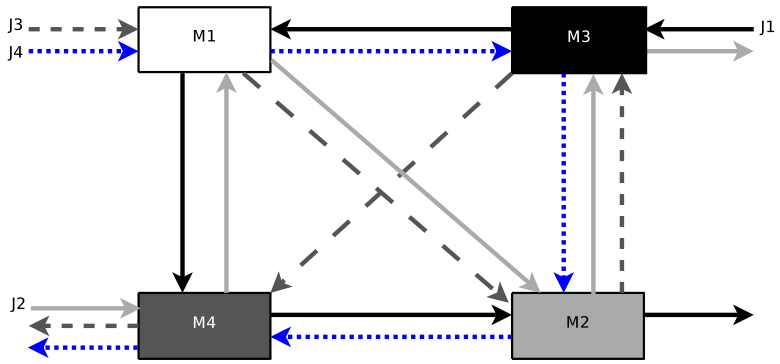


FIGURE 1.5 – Un problème de type Job Shop 4×4 : quatre travaux doivent passer par quatre machines dans un ordre défini

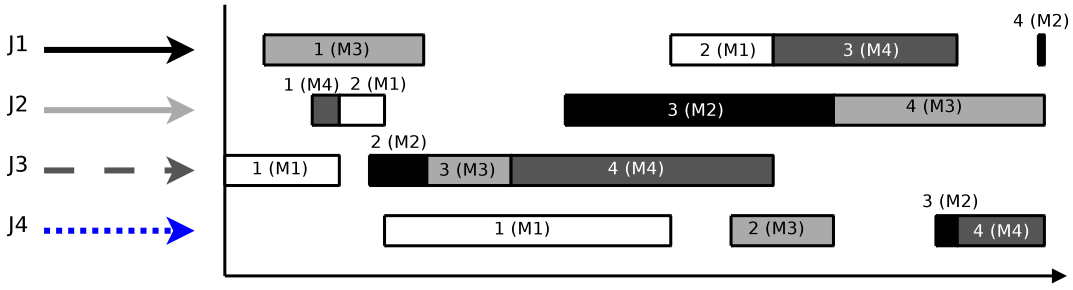


FIGURE 1.6 – Une solution optimale au problème de Job Shop

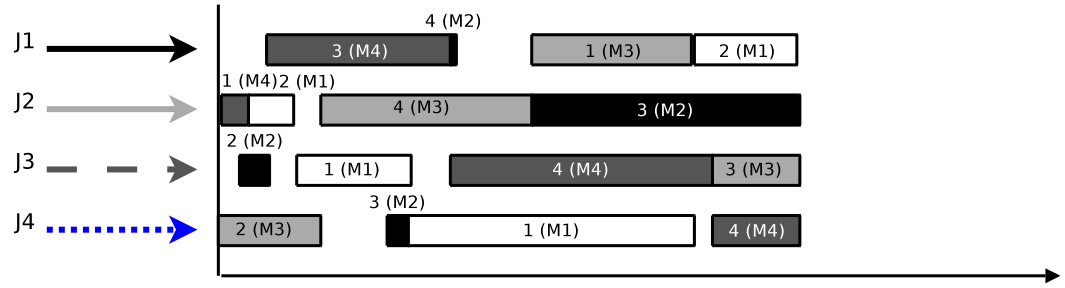


FIGURE 1.7 – Une solution optimale au problème d'Open Shop (mêmes paramètres que le problème de JobShop ci-dessus)

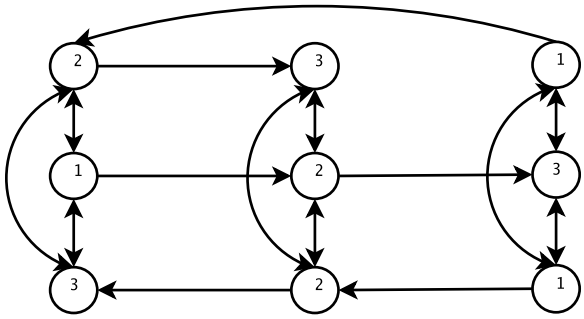


FIGURE 1.8 – Le réseau de contraintes d'un problème de Job Shop

tâche de telle sorte que toutes soient exécutées dans le temps imparti et sans que deux tâches utilisent simultanément la même ressource.

Un problème de Job Shop ou Open Shop est défini par un ensemble de m machines et de j travaux découpés en tâches, chaque tâche devant être exécutée sur une machine différente. On donne d'une part la durée de chaque tâche sous forme de matrice à deux dimensions $j \times m$ et d'autre part, le numéro de la machine sur laquelle chaque tâche doit être effectuée. Dans le cas des problèmes de Job Shop, les tâches pour chaque travail doivent être effectuées dans un ordre défini comme illustré par la figure 1.5. Pour un problème d'Open Shop, les tâches peuvent être effectuées dans n'importe quel ordre.

Deux travaux de référence définissent des générateurs aléatoires d'instances de Job Shop ou Open Shop [TAILLARD 1993, GUÉRET ET PRINS 1999]. Le générateur de Taillard peut générer des problèmes de Job Shop et Open Shop (tableau 1.1 et figure 1.5, par exemple), et le générateur de Guéret & Prins génère des problèmes d'Open Shop (ce générateur ne génère pas de matrice indiquant le numéro des machines pour chaque tâche). La table 1.1 donne un exemple de matrices générées par le générateur de Taillard. La figure 1.5 représente l'ordre des tâches définies par cette table. Chaque flèche correspond à un travail. Par exemple, le travail J1, représenté par la ligne noire pleine démarrant sur la machine M3 en haut à droite, passe ensuite sur la machine M1 (en haut à gauche), puis M4 et M2. Les figures 1.6 et 1.7 montrent une solution aux problèmes de Job Shop et Open Shop décrits par cette table. En relaxant la contrainte d'ordre sur les tâches (dans la solution montrée par la figure 1.7, le travail J1 est réalisé dans le désordre : d'abord la tâche 3 sur la machine M4, puis la tâche 4 sur M2, puis la tâche 1 sur M3 et enfin la tâche 2 sur M1). La solution optimale est plus courte que pour le problème de Job Shop de mêmes paramètres, mais étant donné le nombre accru de possibilités, l'espace de recherche est plus grand et donc le problème plus difficile.

Il y a plusieurs manières de définir un CSP correspondant à un problème de Job Shop ou d'Open Shop. Le plus simple est constitué de $j \times m$ variables. La valeur d'une variable $X_{i,j}$ correspond au temps de départ de la tâche $t_{i,j}$. On définit deux types de contraintes :

- les contraintes de précédence : $X_{i,j} + d_{i,j} < X_{i,j+1}$, souvent notées $X_{i,j+1} - X_{i,j} > d_{i,j}$. Ces contraintes indiquent que la tâche $t_{i,j}$ doit se terminer avant que la tâche suivante $t_{i,j+1}$ ne débute. Ces contraintes sont spécifiques aux problèmes de Job Shop
- les contraintes disjonctives : $X_{i,j} + d_{i,j} < X_{i,k} \vee X_{i,j} > X_{i,k} + d_{i,k}$, ou encore $X_{i,k} - X_{i,j} > d_{i,j} \vee X_{i,j} - X_{i,k} > d_{i,k}$. Ces contraintes indiquent qu'une machine ne peut accepter qu'une tâche à la fois, ou, dans le cas des problèmes d'Open Shop, que deux tâches du même travail ne peuvent être exécutées simultanément.

Dans un Open Shop, l'ensemble des tâches d'un même travail forment une (m-)clique de contraintes disjonctives, de même que l'ensemble des tâches fonctionnant sur une même machine (une j-clique). Pour un Job Shop, les tâches d'un même travail sont liées l'une à l'autre par des contraintes de précédence.

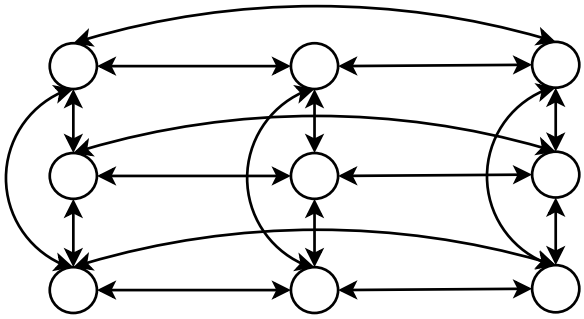


FIGURE 1.9 – Le réseau de contraintes d'un problème d'Open Shop

Les réseaux de contraintes obtenus pour un Job Shop et un Open Shop 3×3 sont représentés figures 1.8 et 1.9, respectivement (les flèches simples $\rightarrow$ sont des contraintes de précédence, les flèches doubles $\leftrightarrow$ sont des contraintes disjonctives).

Représentés sous forme de CSP, ces problèmes ont la particularité de présenter des tailles de domaines très importantes (1 000 à 2 000) en comparaison avec les problèmes aléatoires ou académiques couramment étudiés (généralement moins de 100 valeurs).

Problèmes d'Allocation de Fréquence Radio (RLFAP)

Le problème d'allocation de fréquence radio consiste à déterminer des canaux de communication à partir d'un spectre limité, tout en assurant un minimum d'interférences au sein d'un même réseau de communication. Il s'agit d'un problème combinatoire dont une version simplifiée a été proposée par le CELAR (Centre d'Électronique de l'Armement) en 1993 à partir de données réelles. Les données sources de ces problèmes ont été rendues publiques dans le cadre du programme européen EUCLID CALMA (Combinatorial Algorithms for Military Applications). Il s'agit d'un problème relativement simple à représenter, ne comportant que des contraintes binaires et des valeurs entières finies, tout en gardant toutes les caractéristiques d'un problème industriel [CABON *et al.* 1999,].

Les données permettent d'obtenir deux types d'instances : 11 instances scen (instances obtenues à partir de données réelles) et 14 instances graph (générées artificiellement). Seule l'instance $scen11$ n'est pas triviale. Elle est satisfiable. D'autres instances ont été générées à partir de l'instances $scen11$, en supprimant des fréquences [BESSIÈRE *et al.* 2001,]. Les problèmes obtenus, nommés $scen11\text{-}fi$, i étant le nombre de fréquences supprimées. Toutes ces instances sont insatisfiables, et sont particulièrement difficiles quand i devient petit.

CSP aléatoires

Le développement d'algorithmes de résolution de CSP et le développement de prouveurs a rapidement nécessité le développement de benchmarks, afin de pouvoir comparer les performances des différents algorithmes sur des problèmes possédant des caractéristiques diverses. Les problèmes générés entièrement aléatoirement sont le moyen le plus simple d'obtenir une infinité d'instances présentant toutes les caractéristiques de taille, densité et de dureté possibles. Ces problèmes sont également très homogènes, et ont donc un comportement très particulier vis à vis des méthodes de résolution (inférences difficiles, pas de noyaux durs et heuristiques généralement peu utiles).

L'étude des problèmes aléatoires a permis de mettre en évidence un phénomène considéré comme central pour la résolution de problèmes combinatoires : le « phénomène de seuil » ou la « transition de phase». Des contraintes trop dures ou trop nombreuses, ou au contraire trop lâches ou éparses, rendent le

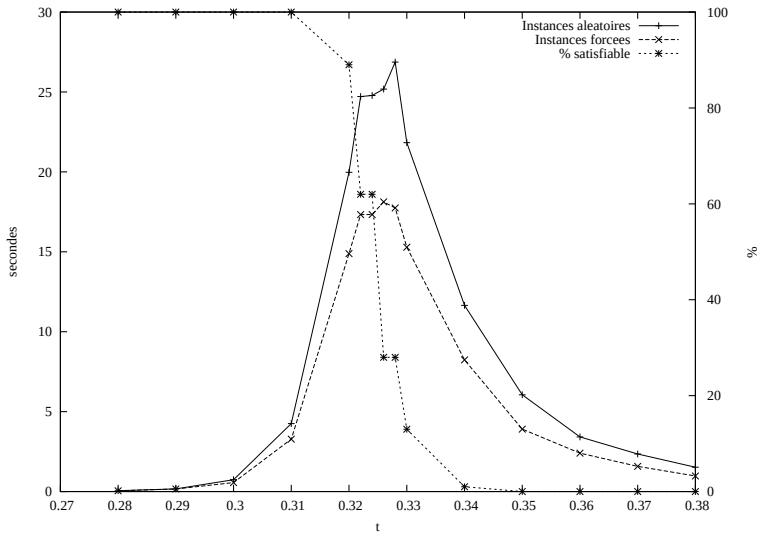


FIGURE 1.10 – Résolution de problèmes de la classe $\langle 2, 35, 17, 249, t\rangle$ avec une méthode complète (MAC)

problème trivialement insatisfiable (sur-contraint) ou satisfiable (sous-contraint), respectivement, mais il existe une zone intermédiaire, nommée « zone critique », pour laquelle le problème est particulièrement difficile à résoudre. La figure 1.10 met en évidence ce phénomène : on tente de trouver une solution par une méthode systématique complète à une série de problèmes aléatoires binaires, comportant 35 variables de 17 valeurs, impliquées par 249 contraintes. On fait varier la dureté t des variables. La transition de phase est située au niveau du pic ($t = 33\%$ environ). À gauche du pic, une majorité de problèmes est insatisfiable. À droite du pic, la majorité est satisfiable. Au niveau du pic, les problèmes sont satisfiables (ou insatisfiables) avec une probabilité de 50%. On observe le même phénomène en maintenant la dureté fixe et en faisant varier la densité du graphe.

Il existe différents modèles de problèmes aléatoires (par exemple, les modèles A ou B [MOLLOY 2003]). L'objectif des différents modèles est, d'une part de faire en sorte que le phénomène de seuil existe (ce n'est pas systématique sur tous les problèmes), et d'autre part de prévoir sa position. En général, un problème aléatoire est défini selon un quintuplet $\langle k, n, d, e, t\rangle$, donnant respectivement l'arité des contraintes, le nombre de variables, la taille des domaines, le nombre, et la dureté des contraintes. Les contraintes sont réparties aléatoirement parmi toutes les contraintes possibles (correspondant aux éléments du produit cartésien des variables entre elles pour lesquels toutes les variables sont distinctes), et les multiplets autorisés par chaque contrainte sont obtenus à partir d'un tirage probabiliste dépendant de t, ou éventuellement une répartition aléatoire des multiplets autorisés parmi le produit cartésien des domaines des variables de telle sorte que la proportion de multiplets autorisés corresponde à t.

Le modèle RB est particulièrement intéressant, puisqu'il permet simplement d'obtenir des instances aléatoires au seuil [XU ET LI 2000], et éventuellement de les forcer à être satisfiables [XU *et al.* 2007,]. Il définit un problème aléatoire selon le quintuplet $\langle k, n, \alpha, r, t\rangle$. α détermine la taille des domaines ($d = n^\alpha$) et r le nombre de contraintes ($e = rn\ln(n)$). Le point critique est obtenu simplement par la formule $t_{cr} = 1 - \exp(-\frac{\alpha}{r})$. Ce point critique est garanti en particulier pour $\alpha > \frac{1}{k}$ et $t \leq \frac{k-1}{k}$. Par exemple, pour $n = 35$, $\alpha = 0{,}8$ ($d = 17$) et $r = 2$ ($e = 249$, soit $D = 42\%$), on obtient $t_{cr} = 33\%$.

Il est possible de forcer les instances à être satisfiables, simplement en rajoutant artificiellement une solution au problème, sans modifier la position du point critique (cf la deuxième courbe figure 1.10).

Outre la possibilité d'obtenir des instances satisfiables, l'intérêt du modèle RB est que la position du

point critique est indépendante du nombre de variables : on peut fixer α et r, et obtenir des problèmes de taille plus ou moins importante en faisant varier n, tout en conservant la position du point critique. On peut ainsi facilement observer le comportement des algorithmes en passant à l'échelle.

Le langage XCSP 2.0

Un langage de description de CSP a été défini à l'occasion des premières compétitions de prouveurs de CSP [VAN DONGEN *et al.* 2006A,]. La dernière version en date est le langage XCSP 2.0, permettant de transcrire des CSPs discrets, avec des contraintes en extension, en intention ou globales [VAN DONGEN *et al.* 2006B,]. Ce langage a permis la mise en place d'une base de problèmes très différents, aléatoires, académiques ou réels [LECOUTRE 2006]. Il est alors possible lors du développement d'un algorithme ou d'une heuristique de tester ses innovations sur un vaste ensemble de problèmes et de déterminer les séries sur lesquelles tel ou tel algorithme est le plus efficace. D'autre part, la définition d'une base commune permet la mise en place de compétitions de prouveurs. La mise en place de compétitions dans la communauté SAT a permis de fortement dynamiser ce secteur de recherche.

[ZHOU ET WALLACE 2005] présente également une conversion entre XCSP et le langage CLP(FD) utilisé dans la plupart des interpréteurs Prolog.

Contraintes globales

Certains CSP impliquent des *contraintes globales*. La communauté scientifique ne s'est pas encore entendue sur la définition formelle d'une contrainte globale [BESSIÈRE ET VAN HENTENRYCK 2003]. Au moins trois concepts peuvent être considérés lorsque l'on définit une contrainte globale. On peut considérer qu'une contrainte globale ne peut être décomposée en contraintes binaires, ou quand cette décomposition est moins expressive (en particulier, une même définition de consistance locale — voir section 1.2 — n'agit pas de la même façon sur la décomposition que sur la contrainte globale), ou tout simplement qu'il existe une manière plus efficace (d'un point de vue de la complexité algorithmique — voir section 1.1.4) de traiter cette contrainte.

La contrainte globale la plus utilisée et la plus connue est la contrainte *all-different* (ou *all-diff*), imposant une valeur différente à toutes les variables impliquées. Il est possible de la décomposer en une clique de contraintes binaires « $\neq$ », mais on peut la traiter bien plus efficacement, tant du point de vue du nombre de déductions possibles que de la complexité algorithmique, par des algorithmes spécifiques [RÉGIN 1994, VAN HOEVE 2001].

1.1.3 Le problème SAT

Le problème SAT est un problème de décision qui consiste à déterminer si une formule booléenne mise sous forme normale conjonctive (CNF, c'est à dire sous la forme d'une conjonction de disjonctions de littéraux — voir ci-dessous) admet ou non une solution. Le problème SAT peut être vu comme un cas particulier de CSP (le domaine de toutes les variables est limité à deux valeurs), mais il est également possible de convertir tout problème CSP discret en problème SAT. Il existe donc un lien très fort entre le problème SAT et les CSP discrets : ils appartiennent de fait à la même classe de complexité (NP-complet, voir ci-dessous).

Pour SAT, une variable booléenne est un littéral, qui peut apparaître positivement (x) ou négativement ($\neg x$) dans une clause. Une clause est une disjonction de littéraux ($c = x_1 \vee \neg x_2 \vee \cdots \vee x_k$). Un problème sous forme normale conjonctive (CNF) consiste en une conjonction de clauses ($\Sigma = c_1 \wedge c_2 \wedge ... \wedge c_e$). Toute formule propositionnelle peut être convertie sous forme CNF équivalente pour la satisfiabilité en temps polynomial (il faut ajouter des variables).

Le problème SAT étant le problème NP-complet de référence, et il est particulièrement important en théorie de la complexité. Il existe un grand nombre de méthodes et de prouveurs très matures permettant de résoudre le problème SAT en pratique sur un vaste ensemble de problèmes « réels », c'est à dire issus du monde de l'industrie.

Résoudre le problème SAT : la procédure DPLL

SAT est le problème NP-complet le plus étudié et il existe un très grand nombre de méthodes pour le résoudre. Comme pour le problème de satisfaction de contraintes, il existe des méthodes systématiques et des méthodes incomplètes (recherche locale, algorithmes évolutionnaires, colonies de fourmis...) Une des méthodes les plus efficaces reste l'algorithme systématique DPLL (pour Davis-Putnam-Logemann-Loveland [DAVIS ET PUTNAM 1960, DAVIS *et al.* 1962,]). Tout comme les algorithmes systématiques pour la résolution de CSP comme MAC (cf section 1.3.1), il alterne hypothèses et inférence pour trouver à coup sûr une solution (en temps exponentiel par rapport au nombre de littéraux) ou à prouver l'inconsistance du problème. La principale méthode d'inférence pour SAT est la *propagation unitaire*. L'algorithme cherche parmi toutes les clauses du problème les clauses de taille 1, c'est à dire contenant un seul littéral, positif ou négatif. Comme dans une formule CNF toutes les clauses doivent être satisfaites, il est indispensable de fixer les littéraux unitaires à **vrai**. Si le même littéral apparaît dans plusieurs clauses unitaires, au moins une fois positivement et une fois négativement, le sous-problème courant est insatisfiable et il faut remettre en cause la dernière hypothèse effectuée. La propagation s'effectue jusqu'à ce que toutes les clauses soient de taille 2 au moins. Ensuite, l'algorithme effectue un choix : il choisit un littéral à affecter à vrai ou à faux. Une heuristique de branchement intervient pour effectuer ce choix. Une nouvelle phase de propagation unitaire est alors effectuée, et ainsi de suite jusqu'à ce qu'une solution soit atteinte ou qu'aucune solution ne soit trouvée après avoir effectué les deux hypothèses **vrai** et **faux** sur le premier littéral de branchement sans succès.

Certains prouveurs SAT modernes utilisent également la technique dite CDCL pour Conflict Driven Clause Learning [ZHANG *et al.* 2001,], particulièrement efficace sur les problèmes « industriels ». Les techniques d'apprentissage ne se sont pas encore révélées fructueuses dans le cadre de la résolution de CSP et ne seront évoquées que de manière limitée dans le présent document. Les prouveurs SAT n'utilisant pas le CDCL utilisent des heuristiques de branchement plus complexes afin de compenser l'absence d'informations déduites des conflits. Ces prouveurs semblent particulièrement efficaces sur les problèmes aléatoires.

Heuristiques de branchement pour SAT

Il existe de nombreuses heuristiques de branchement proposées pour le problème SAT. Les plus récentes, VSIDS et UP, sont respectivement utilisées dans les prouveurs Zchaff [ZHANG *et al.* 2001,] et Satz [LI ET ANBULAGAN 1997]. Le premier utilise le nombre d'occurrences des littéraux dans l'ensemble des nogoods appris au cours de la recherche, et le second mesure l'effet de la propagation unitaire dans la formule quand un littéral est affecté. Auparavant, les prouveurs CSAT [DUBOIS *et al.* 1993,] et POSIT [FREEMAN 1995], parmi bien d'autres, utilisaient des heuristiques plus simples. La plupart sont des variantes de l'heuristique très connue Jeroslow-Wang (JW) [JEROSLOW ET WANG 1990], et évaluent un littéral en fonction de ses propriétés syntaxiques (ici le nombre d'occurrences des littéraux et la longueur des clauses).

L'idée principale ayant donné lieu à l'heuristique Jeroslow-Wang est la suivante : Σ étant une formule CNF comportant n littéraux, Σ admet $p = 2^n$ interprétations. Chaque clause $c \in \Sigma$ supprime $v = 2^{n-|c|}$ lignes de la table de vérité ($|c|$ représente la taille de la clause, c'est à dire le nombre de littéraux dans c). v correspond donc au nombre d'interprétations qui falsifient c. La proportion $v/p = w = 2^{-|c|}$ représente

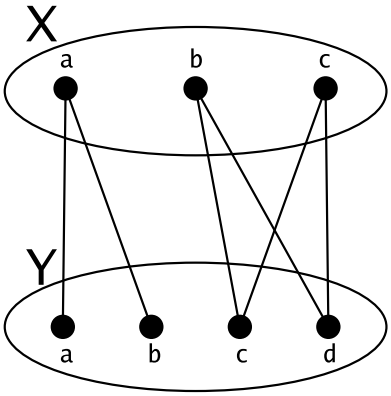


FIGURE 1.11 – Une contrainte et deux variables

la proportion de l'espace de recherche où c est falsifiée. Plus petite est la clause, plus grand est le nombre d'interprétations qui la falsifient. Une clause de taille 1 est falsifiée par 2^{n-1} interprétations, c'est à dire la moitié de l'espace de recherche. En choisissant un littéral apparaissant un grand nombre de fois dans des clauses plus petites, on augmente d'autant l'impact de la propagation unitaire et on réduit ainsi la taille de l'arbre de recherche.

On notera $H(\Sigma, x)$ le score d'une variable x dans une CNF Σ. Ce score est généralement défini comme une fonction dite « à deux faces », $f(h(x), h(\neg x))$, des scores associés aux littéraux positif ($h(x)$) et négatif ($h(\neg x)$). La prochaine variable à affecter est alors choisie parmi les variables ayant le score le plus grand (max) ou le plus petit (min). Généralement, les heuristiques pour SAT sont formulées comme suit :

Définition 11. Soit Σ une formule propositionnelle et x une variable propositionnelle de Σ. Le score, noté $H_w(\Sigma, x)$, de x dans Σ par rapport à une fonction de pondération w est défini comme suit :

$$H_w(\Sigma,x) = \sum_{x \in c} w(|c|) + \sum_{\neg x \in c} w(|c|), \text{ avec } c \in \Sigma$$

Une heuristique SAT, notée $H_w^{\otimes}$, considère l'ensemble des variables $x \in \text{vars}(\Sigma)$ et sélectionne en fonction de l'opérateur $\otimes$ la variable ayant la valeur optimale $H_w(\Sigma,x)$.

En considérant un opérateur de sélection $\otimes$ et une fonction de pondération w, différentes heuristiques peuvent être dérivées de la formulation générale donnée dans la définition 11. Par exemple, la règle Jeroslow-Wang à deux faces (JW) correspond à $H_w^{\otimes}$ où $w(\alpha) = 2^{-\alpha}$ et $\otimes = max$. De nombreuses variantes de l'heuristique JW ont été proposées. Toutes tentent de choisir les variables ayant le maximum d'occurrences dans les clauses de taille minimale (Maximum Occurrences in clauses of Minimal Size — MOMS) [DUBOIS *et al.* 1993,]. Une autre heuristique simple mais intéressante pourrait être $H_w^{\otimes}$ avec $w(\alpha) = 1$ et $\otimes \in \{min, max\}$. Cette heuristique, avec $\otimes = max$ (respectivement $\otimes = min$), choisit en priorité une variable avec le plus grand (respectivement le plus petit) nombre d'occurrences dans la formule. Intuitivement, c'est en choisissant le littéral apparaissant dans le plus de clauses que l'on réduira le plus grand nombre de clauses, et donc potentiellement déclenchera le plus de propagations unitaires.

Convertir un CSP en problème SAT

Une variable propositionnelle x_v est associée à chaque valeur X_v du CSP (X étant une variable du CSP et $v \in \text{dom}(X)$). La correspondance est la suivante : x_v est **vrai** si la valeur v est assignée à X (i.e. $X = v$) et x_v est **faux** si v est retirée de $\text{dom}(X)$ (i.e. $X \neq v$).

Des clauses *au plus une* ($\neg x_a \vee \neg x_b$, $\neg x_a \vee \neg x_c$, $\neg x_b \vee \neg x_c \ldots$) et *au moins une* ($x_a \vee x_b \vee x_c \ldots$) sont ajoutées afin d'exprimer le fait qu'une valeur et une seule doit être affectée à chaque variable du CSP.

Les clauses suivantes codent les domaines des variables du problème représenté figure 1.11 :

Au moins une :	Au plus une :		
$x_a \vee x_b \vee x_c$	$\neg x_a \vee \neg x_b$	$\neg y_a \vee \neg y_b$	$\neg y_b \vee \neg y_c$
$y_a \vee y_b \vee y_c \vee y_d$	$\neg x_a \vee \neg x_c$	$\neg y_a \vee \neg y_c$	$\neg y_b \vee \neg y_d$
	$\neg x_b \vee \neg x_c$	$\neg y_a \vee \neg y_d$	$\neg y_c \vee \neg y_d$

Il y a plusieurs manières de coder les contraintes sous forme de clauses SAT. La plus directe consiste à *coder les conflits* . Un autre codage plus efficace consiste à *coder les supports*.

Codage direct [DE KLEER 1989**]** Chaque multiplet interdit par chaque contrainte est encodé sous la forme d'une clause. La taille de la clause correspond à l'arité de la contrainte. Une constrainte binaire est encodée par un ensemble de clauses binaires : une contrainte C telle que $\mathrm{vars}(C) = \{X,Y\}$ est codée par l'ensemble :

$$\bigcup_{\{X_v,Y_w\} \in \mathrm{cfl}(C)} \{(\neg x_v \vee \neg y_w)\}.$$

La figure 1.11 représente un réseau constitué d'une contrainte impliquant deux variables. Par codage direct de ce réseau, on obtient l'ensemble de clauses suivant :

$$\begin{array}{lll} \neg x_a \vee \neg y_c & \neg x_b \vee \neg y_a & \neg x_c \vee \neg y_a \\ \neg x_a \vee \neg y_d & \neg x_b \vee \neg y_b & \neg x_c \vee \neg y_b \end{array}$$

Codage des supports [GENT 2002**]** L'idée du codage des supports a été introduite par [KASIF 1990] et étendue par [GENT 2002]. Pour une contrainte C telle que $\mathrm{vars}(C) = \{X,Y\}$ est encodée à l'aide des supports $\mathcal{S}pV(C, X_v)$ pour les valeurs de X et $\mathcal{S}pV(C, Y_w)$ pour les valeurs de Y. Les clauses support sont alors l'union des deux ensembles de clauses suivants :

1. $\displaystyle\bigcup_{v \in \mathrm{dom}(X)} \{(\neg x_v \vee y_a \vee \cdots \vee y_k) \mid \{Y_a, \ldots, Y_k\} = \mathcal{S}pV(C, X_v)\}$
2. $\displaystyle\bigcup_{w \in \mathrm{dom}(Y)} \{(\neg y_w \vee x_b \vee \cdots \vee x_l) \mid \{X_b, \ldots, X_l\} = \mathcal{S}pV(C, Y_w)\}$

À partir de l'exemple représenté par la figure 1.11, on obtient l'ensemble de clauses suivant :

$$\begin{array}{ll} \neg x_a \vee y_a \vee y_b & \neg y_a \vee x_a \\ \neg x_b \vee y_c \vee y_d & \neg y_b \vee x_a \\ \neg x_c \vee y_c \vee y_d & \neg y_c \vee x_b \vee x_c \\ & \neg y_d \vee x_b \vee x_c \end{array}$$

Le codage par supports permet de faire un lien direct entre les méthodes de résolution pour le problème SAT et le problème CSP. Les inférences effectuées par la « propagation unitaire » utilisée par la méthode populaire DPLL sur un CSP binaire encodé par les supports est strictement identique à ce qui est obtenu par la consistance d'arc utilisée par l'algorithme MAC (voir section 1.3.1) sur ce même réseau. Le codage par supports présente l'inconvénient de nécessiter un très grand nombre de clauses quand il est appliqué aux contraintes non binaires. Ce cas ne sera pas étudié ici.

À noter que pour les deux codages, les clauses *au plus une* ne sont pas nécessaires pour résoudre le problème de décision (si une solution existe sans ces clauses, alors il existe une solution avec ces clauses),

mais doivent être prises en compte pour effectuer une correspondance entre les solutions trouvées par SAT vers le CSP [WALSH 2000, GENT 2002].

D'autres conversions CSP vers SAT existent, notamment le *Log Encoding* [WALSH 2000], permettant d'obtenir un nombre restreint de variables propositionnelles dans le problème SAT (logarithmique en la taille du CSP). L'idée est simplement d'utiliser la représentation en base deux (binaire) des valeurs décimales du domaine des variables du CSP : chaque littéral SAT correspond à un « bit ». Cette représentation est cependant moins efficace que les précédentes. En particulier, la propagation unitaire utilisée par DPLL permet moins d'inférence sur la représentation Log que sur la représentation directe, et a fortiori la représentation par supports, d'un même problème.

1.1.4 Complexité algorithmique

Classes de complexité

La théorie de la complexité s'intéresse à l'étude formelle de la difficulté des problèmes (calculables) en informatique, la question étant de savoir s'ils peuvent être résolus efficacement ou pas en se basant sur une estimation (théorique) des temps de calcul et des besoins en mémoire informatique.

Afin d'étudier la complexité des problèmes, on ramène tout ordinateur à un modèle formel. On distingue essentiellement la machine de Turing et la machine de Turing non-déterministe. Une machine de Turing déterministe ne fait qu'un calcul à la fois. Un calcul est constitué d'étapes élémentaires et pour un état donné de la mémoire de la machine, l'action élémentaire effectuée sera toujours la même. Tout ordinateur classique programmé en langage impératif peut être ramené à une machine de Turing.

Une machine non-déterministe est purement théorique et ne peut être construite. À chaque étape de son calcul, cette machine peut effectuer un choix non-déterministe : elle a un choix entre plusieurs actions, et elle en effectue une. Si l'un des choix l'amène à accepter l'entrée, on considère qu'elle a fait ce choix là. En quelque sorte, elle devine toujours juste. Un autre manière de voir leur fonctionnement est de considérer qu'à chaque choix non-déterministe, elles se dédoublent, les clones poursuivent le calcul en parallèle suivant les branches du choix.

La théorie de la complexité repose sur la définition de classes de complexité permettant de classer les problèmes en fonction de la complexité des algorithmes qui existent pour les résoudre. Les classes les plus importantes sont les classes P et NP. Un problème de décision est dans P s'il peut être décidé par un algorithme déterministe en un temps polynomial par rapport à la taille de l'instance. Le problème est alors dit polynomial. Les problèmes dans P correspondent en fait à tous les problèmes facilement solubles.

Un problème NP est non-déterministe polynomial : la classe NP réunit les problèmes de décision pour lesquels la réponse *oui* peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille de l'instance. Étant donné une solution au problème NP (un *certificat*), il existe un algorithme dans P permettant de vérifier ce certificat. Intuitivement, les problèmes dans NP sont tous les problèmes qui peuvent être résolus en énumérant l'ensemble des solutions possibles en les testant avec un algorithme polynomial. La classe Co-NP est similaire à la classe NP, mais avec la réponse *non*.

C étant une classe de complexité, on dit qu'un problème est C-complet si il est dans C, et qu'il est C-difficile. Un problème est C-difficile s'il est au moins aussi dur que tous les problèmes dans C, c'est à dire que tout problème dans C peut être ramené à ce problème [COOK 1971]. Cook démontre en 1971 que le problème SAT est NP-complet. Étant donné qu'un CSP peut être ramené à un problème SAT et vice-versa en temps polynomial, le problème CSP est également NP-complet. Le nom de ces classes est quelque peu sujet à confusion, étant donné que les problèmes NP-difficiles ne sont pas nécessairement dans NP. Les problèmes NP-complets sont les problèmes les plus difficiles *dans* NP. Les problèmes

NP-difficiles sont *au moins* aussi difficiles que NP, mais ne sont pas nécessairement *dans* NP.

On sait que $P \subseteq NP$, mais $P \neq NP$ (i.e. il n'est pas possible de résoudre un problème NP en temps polynomial sur une machine déterministe) est encore au stade de la conjecture, bien qu'elle soit très forte. En pratique, on s'attache donc à améliorer des algorithmes exponentiels pour les problèmes NP.

Comportements asymptotiques

La notation grand O permet de décrire le comportement asymptotique (l'ordre) des fonctions. Elle définit une limite asymptotique supérieure, et est particulièrement importante pour l'étude et la comparaison des algorithmes polynomiaux.

On écrit $f(n) \in O(g(n))$ ssi $\limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$.

Par exemple, la fonction $f(n) = 4n^2 - 2n + 2 \in O(n^2) \in O(n^3)$.... Les problèmes NP-complets nécessitent a priori des algorithmes exponentiels en $O(\exp(n))$ pour être résolus (sauf si $P = NP$!)

La notation grand Ω permet de définir une limite asymptotique inférieure :

$f(n) \in \Omega(g(n))$ ssi $\liminf_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| > 0$.

$f(n) = 4n^2 - 2n + 2 \in \Omega(n^2) \in \Omega(n)$.... On se contente cependant en général de déterminer la limite supérieure par la notation O, bien que le « pire des cas » stigmatisé par ces études ne survienne que rarement en pratique.

Dans certains cas, la notation Θ, qui décrit l'équivalence asymptotique, pourra être particulièrement intéressante :

$f(n) \in \Theta(g(n))$ ssi $f(n) \in O(g(n)) \wedge g(n) \in O(f(n))$,
ou encore $f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n))$.

La notation Θ est la plus précise, puisqu'elle exprime à la fois une borne asymptotique inférieure et supérieure (d'ailleurs, $f(n) = 4n^2 - 2n + 2 \in \Theta(n^2) \notin \Theta(n^3)$ ni $\Theta(n)$).

Les algorithmes systématiques utilisés pour la résolution de CSP ont une complexité exponentielle (et il est peu probable qu'on trouve un jour un algorithme non exponentiel pour résoudre un tel problème, à moins que $P = NP$). L'ordre de ces algorithmes est en $O(d^n)$.

1.2 Méthodes d'inférence

Pour résoudre des CSP (et les autres problèmes NP-complets), on utilise généralement un algorithme exponentiel complet de recherche systématique, ou un algorithme incomplet de recherche locale (voire d'autres techniques incomplètes, comme les algorithmes évolutionnaires ou par colonies de fourmis). Dans les deux cas, on explore un espace de recherche de taille exponentielle pour trouver une solution. Les méthodes d'inférence ont pour but de réduire la taille du problème, et donc de l'espace de recherche, en temps polynomial. Pour ce faire, on a la possibilité d'exploiter des propriétés des graphes appelées des *consistances*. Les consistances permettent d'identifier des valeurs ou plus généralement des instanciations qui ne pourront appartenir à aucune solution (nogoods), et donc de les supprimer du réseau de contraintes [DECHTER 2003].

On utilisera les notations suivantes :

– Étant donné une consistance ϕ, $\phi(P)$ représente le réseau ϕ-consistant $\leq P$ le plus grand. En général, $\forall P$, $\phi(P)$ est unique, bien qu'on puisse concevoir des consistances pour lesquelles ce ne soit pas vrai.

– Si le domaine d'une variable est vide ($\mathrm{dom}(X) = \emptyset$) ou si une contrainte n'autorise aucune instanciation ($\mathrm{rel}(C) = \emptyset$), le réseau est inconsistant. On note alors $P = \bot$.
– $P|_{X=a}$ représente le réseau P dont le domaine de la variable X est réduite à la valeur $\{a\}$.

Les consistances de domaine permettent d'identifier des nogoods de taille 1, c'est à dire des valeurs inconsistantes [DEBRUYNE ET BESSIÈRE 2001]. Ce sont les consistances les plus utiles en pratique, puisque la suppression de valeurs est une opération très élémentaire dans les prouveurs de CSP.

Les consistances de relation permettent d'identifier des nogoods de taille quelconque, c'est à dire des instanciations inconsistantes. Les consistances de relation nécessitent que les contraintes soient exprimées en extension afin de stocker les multiplets supprimés, et la structure du réseau peut être amenée à être modifiée (on ajoute des contraintes). Les consistances de relation sont donc peu utilisées en pratique, à cause des coûts en mémoire, mais aussi parce que la modification de la structure des graphes peut perturber les algorithmes de recherche. De plus, en ajoutant des contraintes, on ralentit les algorithmes de propagation simples comme la consistance d'arc, dont la complexité dans le pire des cas est proportionnelle au nombre de contraintes. Les consistances de relation conservatives ont été introduites afin de limiter ces inconvénients : on ne stocke les multiplets supprimés que dans les contraintes déjà présentes dans le réseau initial. Ces consistances sont bien entendu moins fortes que les consistances de relation « complètes » [DEBRUYNE 1999].

Pour comparer la force des différentes méthodes d'inférence, on utilise les notations suivantes :
– Une consistance ϕ est plus forte qu'une consistance ψ, noté $\phi \succeq \psi$ ssi tout CN ϕ-consistant est aussi ψ-consistant.
– Une consistance ϕ est strictement plus forte qu'une consistance ψ, noté $\phi \succ \psi$ ssi $\phi \succeq \psi$ et il existe au moins un CN ψ-consistant mais pas ϕ-consistant.

Les inférences basées sur les consistances transforment un problème en un autre strictement équivalent : les solutions sont identiques. Les méthodes d'inférence permettant d'éliminer les symétries (la détection de valeurs substituables, par exemple) suppriment une partie des solutions, et le problème n'est alors équivalent que du point de vue de la consistance.

En général, si $\phi \succeq \psi$, alors $\forall P, \phi(P) \leq \psi(P)$. Cependant, pour établir certaines formes d'inférence (comme la chemin consistance PC ou la chemin consistance partielle PPC), on est amené à ajouter des contraintes dans le réseau. Dans ce cas, la relation $\phi(P) \leq \psi(P)$ tient toujours si l'on compare les contraintes ajoutées dans $\phi(P)$ aux contraintes universelles implicites portant sur les mêmes variables dans $\psi(P)$.

On peut être amené à combiner des consistances. On notera $\phi \circ \psi(P) = \phi(\psi(P))$ et $(\phi \circ \psi)^n$ est défini récursivement telle que $(\phi \circ \psi)^0(P) = P$ et $(\phi \circ \psi)^n(P) = \phi \circ \psi \circ (\phi \circ \psi)^{n-1}(P)$.

1.2.1 Consistance d'arc et consistance d'arc généralisée

La consistance d'arc (AC pour Arc Consistency) est clairement la propriété la plus étudiée et la plus utilisée pour effectuer des inférences lors de la résolution de CSP. Applicable sur pratiquement tous les types de CSP discrets de par sa faible complexité théorique et son très bon comportement en pratique, il existe de plus des algorithmes de conception très simple capables d'établir la consistance d'arc de manière optimale. La consistance d'arc est définie sur les réseaux binaires, mais est généralisable aux réseaux quelconques (GAC pour Generalized Arc Consistency).

Définition 12 (Consistance d'arc généralisée)**.** Étant donné un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$, une valeur X_a est arc-consistante (généralisée) ssi X_a a au moins un support dans toutes les contraintes associées à la variable X.

Un réseau P est arc-consistant ssi $\forall X \in \mathscr{X}$, $\forall a \in \mathrm{dom}(X)$, X_a est arc-consistant.

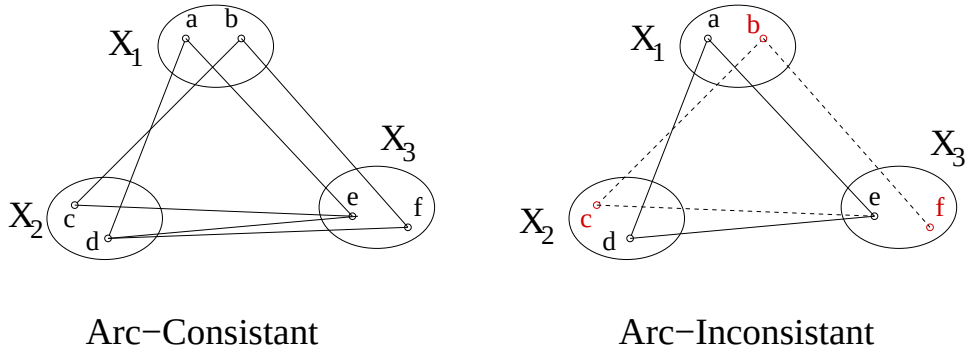


FIGURE 1.12 – Exemple de propagation par consistance d'arc

Les nombreux algorithmes permettant d'établir la consistance d'arc (ou la consistance d'arc généralisée) cherchent à supprimer les valeurs ne vérifiant pas cette propriété le plus rapidement possible. La figure 1.12 donne un exemple de réseau arc-consistant à gauche, et arc-inconsistant à droite. Pour rendre le réseau de droite à nouveau arc-consistant, il faut tout d'abord supprimer la valeur f de X_3 qui n'a pas de support dans X_2, puis, par propagation, la valeur b de X_1 et enfin la valeur c de X_2.

L'algorithme GAC-3

L'algorithme de référence permettant d'établir la consistance d'arc reste l'algorithme AC-3 [MACKWORTH 1977], assez efficace en général et facile à implanter. Il présente une complexité temporelle en $O(ed^3)$ dans le pire des cas et une complexité spatiale limitée (la queue Q a une taille en (n)), en dehors des structures de données nécessaires pour décrire le réseau. L'algorithme GAC-3 généralisant l'algorithme AC-3 aux contraintes d'arité quelconque présente une complexité temporelle dans le pire des cas en $O(ek^3d^{k+1})$.

Les algorithmes 1, 2 et 3 permettent d'établir la consistance d'arc généralisée suivant la méthode GAC-3 orientée variables [MCGREGOR 1979, CHMEISS ET JÉGOU 1998, BOUSSEMART *et al.* 2004B,]. L'algorithme 1 se base sur un ensemble Q des variables « récemment modifiées » dont les voisins doivent être révisés par l'algorithme 2. Les lignes 6, 7, 9, 10 et 15 à 18 sont des optimisations qui seront décrites dans le pargagraphe suivant. En général, on fera appel à GAC-3 en initialisant l'ensemble de variables à réviser Q par $\mathscr{Y} = \mathscr{X}$. Cependant, quand on sait qu'une seule variable X a été modifiée dans un réseau GAC (ce qui est souvent le cas quand on utilise un algorithme GAC comme base d'un autre algorithme comme SAC ou MGAC — cf sections 1.2.4 et 1.3.1), on pourra utiliser $\mathscr{Y} = \{X\}$. On fait appel ligne 11 à une fonction $avoidRevision$ qui permet de détecter avec une complexité $O(k)$ si toutes les valeurs de X ont au moins un support dans C (voir paragraphe suivant sur la détection de révisions inutiles). Si $avoidRevision$ renvoie **faux**, il faut effectuer une révision complète de la variable via l'algorithme 2.

La fonction $seekSupport(C, X_a)$ contrôle l'existence d'un *support*, c'est à dire une instanciation satisfaisant la contrainte C. L'algorithme 3 est un algorithme simple qui réalise cette fonction en $O(d^{k-1})$, mais on verra ensuite que l'on peut améliorer cette complexité. $check(C, I)$ contrôle simplement si la contrainte C est satisfaite par l'instanciation I. Si aucun support pour une valeur dans une contrainte n'est trouvé, alors la valeur peut être supprimée. Il faudra alors contrôler toutes les variables voisines pour voir si d'autres valeurs ne doivent pas être supprimées par propagation (ligne 8 de l'algorithme 1). Il existe d'autres variantes de GAC-3, orienté contraintes, orienté arcs $(Contrainte, Variable)$ ou orienté variables inversées (on stocke dans Q non pas les variables récemment modifiées mais les variables à réviser). L'avantage de la version présentée par l'algorithme 1 est que la file Q contient généralement

Algorithme 1 : GAC-3 ($P = (\mathscr{X},\mathscr{C})$: CN, $\mathscr{Y}$: {Variable}) : CN

```
1  Q ← 𝒴
2  ∀C ∈ 𝒞, ∀X ∈ vars(C), rev[C, X] ← vrai
3  tant que Q ≠ ∅ faire
4      prendre X de Q
5      pour chaque C ∈ 𝒞 | X ∈ vars(C) faire
6          si ¬rev[C, X] alors
7              continuer
8          pour chaque Y ∈ vars(C) faire
9              si Y = X ∧ ∄Z ∈ vars(C) | Z ≠ Y ∧ rev[C, Z] alors
10                 continuer
11             si ¬avoidRevision(C, Y) ∧ revise(C, Y) alors
12                 si dom(Y) = ∅ alors
13                     retourner ⊥
14                 Q ← Q ∪ {Y}
15                 pour chaque C′ ∈ 𝒞 | C′ ≠ C ∧ Y ∈ vars(C′) faire
16                     rev[C′, Y] ← vrai
17         pour chaque Y ∈ vars(C) faire
18             rev[C, Y] ← faux
19 retourner P
```

Algorithme 2 : revise(C : Contrainte, X : Variable) : Booléen

```
1  domainSize ← |dom(X)|
2  pour chaque v ∈ dom(X) faire
3      si ¬seekSupport(C, X_v) alors
4          retirer v de dom(X)
5  retourner domainSize ≠ |dom(X)|
```

Algorithme 3 : seekSupport-3(C : Contrainte, X_v : Variable$_{valeur}$) : Booléen

```
1  Soit ℐ l'ensemble des instanciations possibles des variables vars(C)\X
2  pour chaque I ∈ ℐ faire
3      si check(C, I ∪ X_v) alors
4          retourner vrai
5  retourner faux
```

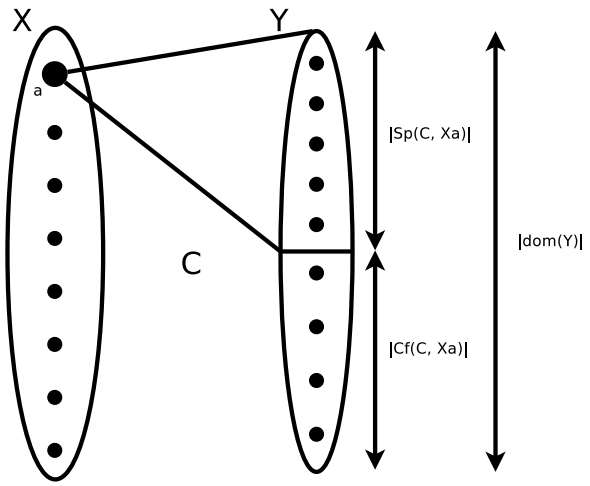


FIGURE 1.13 – Relation entre la taille du domaine, les conflits et les supports

peu d'éléments et il devient possible de la trier suivant un critère quelconque sans alourdir l'algorithme. En révisant d'abord les variables dont les domaines sont les plus petits, on obtient la variante considérée comme la plus rapide de GAC-3. D'autre part, il est possible de limiter les défauts les plus évidents de l'approche orientée-variable en détectant certaines révisions inutiles [BOUSSEMART *et al.* 2004B,].

Détecter des révisions inutiles

Pour ce faire, nous utilisons une légère variante de l'approche proposée par [BOUSSEMART *et al.* 2004B,] : le compteur $ctr(C, X)$ peut en pratique être remplacé par un booléen. L'idée est que si après avoir sélectionné une variable X, on effectue une révision effective de l'arc (C, Y) (c'est à dire qu'on supprime au moins une valeur du domaine de Y), puis l'on sélectionne la variable Y entraînant une révision effective de l'arc (C, X), il est inutile d'effectuer à nouveau la révision de (C, Y) si X est à nouveau sélectionné et n'a pas été modifié ailleurs. En associant un booléen $rev[C, X]$ à chaque arc, il est possible de déterminer quelles révisions sont utiles : cela correspond au test ligne 6 de l'algorithme 1). Initialement, toutes les révisions sont potentiellement utiles et $rev[C, X]$ est initialisé à **vrai** pour tous les arcs (C, X). D'autre part, quand une variable X est sélectionnée et qu'une contrainte C impliquant X est considérée, deux situations peuvent survenir. Si X est la seule variable de $\mathrm{vars}(C)$ telle que $rev[C, X] =$ **vrai**, alors la révision de tous les arcs (C, Y) tels que $X \neq Y$ est effectuée. Sinon, tous les arcs (C, Y), y compris avec $X = Y$, sont révisés (en effet, il est utile de réviser (C, X) puisqu'au moins une autre variable de $\mathrm{vars}(C)$ a été révisée ailleurs). La structure $rev[C, X]$ a une complexité spatiale en $\Theta(ke)$.

Finalement, on peut également détecter une partie des révisions inutiles par une simple opération permettant de déterminer si une valeur a au moins un support [BOUSSEMART *et al.* 2004C,]. Une valeur X_a a au moins un support dans C_{XY} ssi $|\mathcal{S}pI(C, X_a)| > 0$. On peut parfois détecter ce cas de figure sans avoir à rechercher un support par énumération des multiplets. En effet, à tout moment, $|\mathcal{S}pI(C_{XY}, X_a)| = |\mathrm{dom}(Y)| - |\mathcal{C}f(C_{XY}, X_a)|$ (cf figure 1.13). Évidemment, il n'est pas plus facile d'énumérer les conflits que d'énumérer les supports. Cependant, il est envisageable de compter le nombre de conflits et de supports pour chaque valeur au début de la recherche. Si les relations ne sont pas modifiées au cours de la recherche, les supports et les conflits restent toujours inclus dans les ensembles de supports $\mathcal{S}p^{init}(C_{XY}, X_a)$ et de conflits $\mathcal{C}f^{init}(C_{XY}, X_a)$ au début de la recherche : pendant une recherche par MGAC, on va supprimer des valeurs, c'est à dire soit des conflits, soit des supports.

Algorithme 4 : avoidRevision(C : Contrainte, X : Variable)

```
1 size ← 1
2 pour chaque Y ∈ vars(C) | Y ≠ X faire
3 |  size ← size × |dom(Y)|
4 retourner size > |Cf^max(C, X)|
```

On a donc $\mathcal{C}f(C_{XY}, X_a) \subseteq \mathcal{C}f^{init}(C_{XY}, X_a)$, soit $|\mathcal{C}f(C_{XY}, X_a)| \leq \left|\mathcal{C}f^{init}(C_{XY}, X_a)\right|$ et donc $|\mathcal{S}pI(C_{XY}, X_a)| \geq |\mathrm{dom}(Y)| - \left|\mathcal{C}f^{init}(C_{XY}, X_a)\right|$. Si $|\mathrm{dom}(Y)| > \left|\mathcal{C}f^{init}(C_{XY}, X_a)\right|$, alors $|SpI(C_{XY}, X_a)| > 0$ et il est inutile de rechercher un support de X_a dans C_{XY} : on sait qu'il en existe au moins un.

On peut aussi constater que si $|\mathrm{dom}(Y)| > \max_{\forall v \in \mathrm{dom}(X)} \left|Cf^{init}(C_{XY}, X_v)\right|$, alors toutes les valeurs du domaine de X ont au moins un support dans Y et il est inutile de réviser la variable X par rapport à Y.

Lorsque l'on utilise un algorithme comme GAC-2001 ou GAC-3^{rm}, décrits ci-dessous, tester la première propriété prend presque autant de temps que de contrôler l'existence d'un support. On se contente donc en pratique de la seconde propriété, capable d'identifier un grand nombre de révisions inutiles simultanément. Cette propriété est généralisée aux contraintes d'arité quelconque (algorithme 4). On compte au début de la recherche le nombre de supports et de conflits de chaque valeur (sauf si les arités des contraintes dépassent un certain seuil), et on recherche le maximum de conflits de chaque variable, que l'on stocke dans la structure $|Cf^{max}(C, X)| = \max_{\forall v \in \mathrm{dom}(X)} \left|Cf^{init}(C, X_v)\right|$.

Algorithmes optimaux

AC-3 souffre cependant de « cas pathologiques », des cas particuliers de réseaux où sa complexité dans le pire des cas est exhibée, par exemple les instances de type Domino [ZHANG ET YAP 2001]. L'algorithme AC-4 a été introduit pour améliorer la complexité dans le pire des cas et atteint $O(ed^2)$. Il a été prouvé qu'il s'agissait de la complexité optimale pour établir la consistance d'arc [MOHR ET HENDERSON 1986]. La complexité moyenne de AC-4 est cependant très mauvaise (pratiquement identique à la complexité dans le pire des cas, puisqu'elle nécessite systématiquement de tester tous les multiplets de toutes les contraintes pour maintenir à jour des structures de données) et l'algorithme fonctionne mal en pratique. D'autres algorithmes ont été proposés par la suite : AC-5 [PERLIN 1991, VAN HENTENRYCK *et al.* 1992,], AC-6 [BESSIÈRE 1994], AC-7 [BESSIÈRE *et al.* 1999,] ou AC-8 [CHMEISS ET JÉGOU 1998]. Les algorithmes AC-4 à AC-7 sont des algorithmes à « grain fin », par opposition aux algorithmes à « gros grain » que sont AC-3 et ses variantes. Les algorithmes à grain fin travaillent sur des triplets $(Contrainte, Variable, Valeur)$ alors que les algorithmes à gros grain travaillent sur des arcs $(Contrainte, Variable)$. Cependant, les algorithmes les plus utilisés restent les améliorations d'AC-3, et en particulier AC-2001 et AC-3^{rm}. [WALLACE 1993], par exemple, montre que AC-3 est en pratique presque toujours plus efficace qu'AC-4. AC-2001 consiste en une simple amélioration d'AC-3 et présente la complexité optimale en $O(ed^2)$ [BESSIÈRE *et al.* 2005,].

La complexité optimale pour établir GAC est de $O(ekd^k)$ et est atteinte par GAC-4 [MOHR ET MASINI 1988] et GAC-schema [BESSIÈRE ET RÉGIN 1997]. La généralisation de AC-2001 aux contraintes d'arité quelconque, GAC-2001, présente une complexité dans le pire des cas de $O(ek^2d^k)$. Malgré le facteur k par rapport à la complexité optimale, GAC-2001 reste très efficace en pratique. GAC-2001 améliore GAC-3 en enregistrant pour chaque triplet le dernier multiplet support trouvé $last[Contrainte, Variable_{valeur}]$. Quand on cherche un multiplet valide, on reprend la recherche d'un multiplet valide au

Algorithme 5 : seekSupport-2001(C : Contrainte, X_v : Variable$_{valeur}$) : Booléen

```
1 Soit 𝒥 l'ensemble ordonné des instanciations possibles des variables vars(C)\X
2 pour chaque I ∈ 𝒥 | I ≥ last[C,X_v] faire
3     si check(C, I ∪ X_v) alors
4         last[C,X_v] ← I
5         retourner vrai
6 retourner faux
```

Algorithme 6 : seekSupport-rm(C : Contrainte, X_v : Variable$_{valeur}$) : Booléen

```
1  si check(C, res[C, X_v]) alors
2      retourner vrai
3  Soit 𝒥 l'ensemble des instanciations possibles des variables vars(C)\X
4  pour chaque I ∈ 𝒥 faire
5      I' ← I ∪ X_v
6      si check(C, I') alors
7          pour chaque Y_w ∈ I' faire
8              res[C, Y_w] ← I'
9          retourner vrai
10 retourner faux
```

dernier multiplet trouvé (cf algorithme 5). Cependant, à chaque nouvel appel à GAC-2001 (si au moins une valeur a été restaurée entre deux appels), il faut réinitialiser $last$, puisque entre deux appels le dernier multiplet support trouvé n'est peut être plus valide. Outre le temps de réinitialisation, non négligeable en pratique, il arrive fréquemment qu'un multiplet trouvé reste valide d'un appel sur l'autre à GAC-2001. Une des dernières améliorations de GAC-3, GAC-3^{rm} n'atteint pas la complexité optimale dans le pire des cas mais évite les réinitialisations de la structure $last$ (remplacée ici par une structure res pour « résidus », de conception similaire). L'idée est simplement, si le multiplet enregistré n'est plus valide, de reprendre la recherche d'un multiplet au début. L'algorithme 6 exploite de plus la multidirectionnalité des contraintes en enregistrant le dernier multiplet trouvé pour toutes les valeurs du multiplet simultanément (lignes 6-7). Il s'agit d'un algorithme extrêmement efficace en pratique [LECOUTRE ET HEMERY 2007]. Les structures $last$ ou rev admettent une complexité spatiale en $O(k^2ed)$.

Incrémentalité

Une propriété intéressante de tous les algorithmes de consistance d'arc est leur incrémentalité : un algorithme d'AC est dit incrémental si sa complexité dans le pire des cas est identique quand il est appliqué une fois sur un réseau donnée P et quand il est appliqué jusqu'à nd fois sur P où, entre deux exécutions successives, au moins une valeur a été supprimée. On peut ainsi construire des algorithmes autour d'AC qui exploitent cette propriété et obtenir des complexités dans le pire des cas relativement faibles (cf [BESSIÈRE ET DEBRUYNE 2005], par exemple).

Alternatives à la consistance d'arc

PIC et PWIC : Les alternatives à la consistance d'arc généralisée sont peu nombreuses et peu d'entre elles peuvent être efficacement maintenues pendant la recherche. Deux consistances prometteuses a été récemment mis en avant par [STERGIOU ET WALSH 2006] :

- la consistance de chemin inverse (PIC pour Path Inverse Consistency) relationnelle. La définition de PIC correspond à la définition de K-SAC que l'on trouve dans la section 1.2.4 avec $K = 2$.
- la consistance inverse par paires (PWIC pour PairWise Inverse Consistency), plus forte que rel PIC. PWIC a été développée à partir de la consistance par paires (PWC [JANSSEN *et al.* 1989,], une consistance de relations) qui spécifie que toute instanciation consistante dans une contrainte C doit pouvoir être étendue à toute autre contrainte qui partage au moins une variable avec C.

Ce sont deux consistances de domaine, applicable aux contraintes non-binaires, plus fortes que GAC et pouvant être établies en un temps limité. Les complexités des algorithmes proposés sont assez faibles : $O(e^2k^2d^k)$ pour l'algorithme PWIC-2, ce qui ne représente qu'un facteur $O(e)$ par rapport à GAC-2001. [STERGIOU ET WALSH 2006] montrent que maintenir PWIC pendant la recherche fonctionne particulièrement bien sur les réseaux de contraintes épars (c'est à dire avec un nombre de contraintes relativement faible, où le facteur $O(e)$ a le moins d'impact sur les performances de l'algorithme).

Watched Literals : Les Watched Literals sont issus des algorithmes utilisés pour effectuer la propagation unitaire pour le problème SAT (dans le cadre d'une procédure DPLL). [GENT *et al.* 2006B,] ont récemment étudié une adaptation du mécanisme des Watched Literals pour réaliser GAC lors de la résolution d'un CSP. En pratique, le mécanisme est très proche de celui utilisé par GAC-3^{rm}. L'idée est de « surveiller » (*watch*) des couples (Variable, Valeur) (les « littéraux » pour un CSP). En effet, ce n'est que lorsque le dernier support d'une valeur dans une contrainte est supprimé que la valeur elle-même doit être supprimée. Il est donc inutile d'effectuer des propagations dans tous les cas. On décide donc de « surveiller » un des supports de chaque valeur. Quand celui-ci est supprimé et seulement dans ce cas, on cherche après un autre support. Si aucun support ne peut être trouvé, la valeur peut être supprimée. Les supports ainsis surveillés peuvent être ramenés aux « résidus » utilisés par GAC-3^{rm} et fonctionnent de fait de la même façon.

1.2.2 Consistance de chemin et consistance de chemin conservative

La consistance de chemin a été introduite très tôt dans la littérature, et n'est définie que pour les réseaux binaires. La consistance d'arc a été dans un premier temps étendue à la k-consistance :

Définition 13 (k-consistance)**.** Une instanciation consistante de taille $k-1$ (portant sur $k-1$ variables) est k-consistante si elle peut être étendue à une k^{e} variable, c'est à dire que pour toute k^{e} variable non instanciée X, il existe une valeur a telle que l'instanciation étendue à cette valeur X_a est consistante.

La consistance d'arc est alors la 2-consistance et la consistance de chemin la 3-consistance. Pour définir la consistance de chemin, on introduit la notion d'arc-consistance d'un couple de valeurs dans un réseau binaire :

Définition 14 (Consistance de chemin (PC))**.** Étant donné un CN binaire $P = (\mathscr{X}, \mathscr{C})$, une instanciation de longueur 2 $\{X_a, Y_b\}$ est chemin-consistante (PC) ssi $\forall Z \in \mathscr{X} \mid Z \notin \{X, Y\}$, $\exists c \in \text{dom}(Z)$ tel que les instanciations $\{X_a, Z_c\}$ et $\{Y_b, Z_c\}$ sont consistantes.

P est PC ssi toutes les instanciations consistantes de longueur 2 de P sont PC.

P est fortement chemin-consistant (sPC) ssi il est à la fois AC et PC.

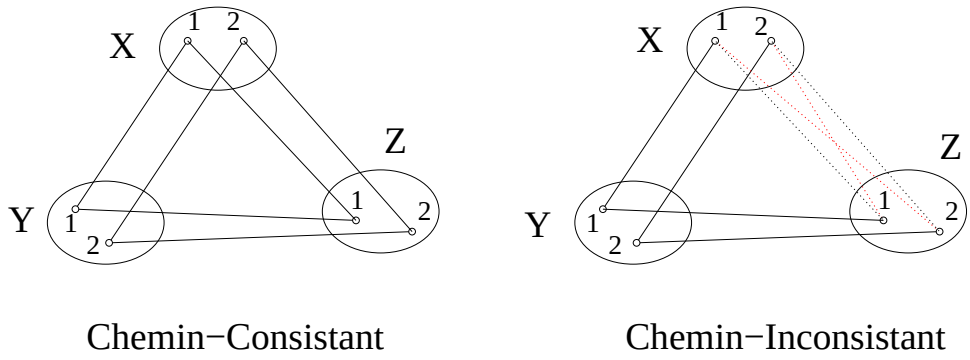


FIGURE 1.14 – Exemple de propagation par consistance de chemin.

Définition 15 (Consistance de chemin conservative (CPC) [DEBRUYNE 1999])**.** Étant donné un CN binaire $P = (\mathscr{X}, \mathscr{C})$, une instanciation de longueur 2 $\{X_a, Y_b\}$ est chemin-consistant conservative (CPC) ssi :

- soit $\nexists C \in \mathscr{C} \mid \text{vars}(C) = \{X, Y\}$,
- soit $\forall Z \in \mathscr{X} \mid \exists (C, C') \in \mathscr{C}^2 \mid \text{vars}(C) = \{X, Z\} \wedge \text{vars}(C') = \{Y, Z\} \wedge \exists c \in \text{dom}(Z)$ tel que les instanciations $\{X_a, Z_c\}$ et $\{Y_b, Z_c\}$ sont consistantes.

P est CPC ssi toutes les instanciations consistantes de longueur 2 de P sont aussi CPC.
P est fortement chemin-consistant conservatif (sCPC) ssi il est à la fois AC et CPC.

La figure 1.14 montre un exemple de propagation par consistance de chemin. La figure de gauche est chemin consistante. Sur la figure de droite, la contrainte universelle entre X et Z doit tout d'abord être introduite. Les instanciations $\{X_1, Z_2\}$ et $\{X_2, Z_1\}$ sont alors chemin-inconsistantes : il n'existe pas de valeur dans Y compatible à la fois avec X_1 et Z_2, ni avec X_2 et Z_1. On obtient donc le réseau de gauche après application de la chemin consistance au réseau de droite. On peut remarquer dans cet exemple que l'application de la consistance de chemin au système $X = Y \wedge Y = Z$ revient à introduire la contrainte implicite $X = Z$. La consistance de chemin conservative a été introduite afin de limiter l'inconvénient d'introduire de nombreuses contraintes pour établir la consistance de chemin (comme indiqué en début de cette section 1.2) : on ignore les instanciations inconsistantes de deux variables si il n'y a pas de contrainte dans le réseau dont les variables correspondent à l'instanciation.

Plusieurs algorithmes ont été introduits permettant d'établir la consistance de chemin. L'algorithme proposé par [MCGREGOR 1979] pour établir la consistance de chemin forte se base sur la consistance d'arc. Il présente une complexité temporelle en $O(n^5 d^5)$ et une complexité spatiale en $O(n^2 d^2)$. Les algorithmes PC-5 [SINGH 1995] et PC-2001 [BESSIÈRE *et al.* 2005,] presentent une complexité temporelle en $O(n^3 d^3)$ (il s'agit de la complexité optimale) et une complexité spatiale en $O(n^3 d^2)$. Finalement, l'algorithme PC-8 [CHMEISS ET JÉGOU 1998], malgré une complexité temporelle non-optimale dans le pire des cas $O(n^3 d^4)$ (et une complexité spatiale en $O(n^2 d^2)$) est considéré comme le plus efficace en pratique.

Théorème 2 ([DEBRUYNE 1999])**.** $\text{PC} \succ \text{CPC}, \text{sPC} \succ \text{sCPC}$

Bien que cette relation signifie a priori que pour tout CN P, $\text{PC}(P) \leq \text{CPC}(P)$, mais PC nécessite généralement pour être établie d'ajouter des contraintes. La relation d'ordre $\leq$ tient cependant si on ajoute dans le réseau $\text{CPC}(P)$ des contraintes universelles implicites, portant sur les mêmes variables, pour remplacer les contraintes manquantes.

On trouve également une autre définition de la chemin consistance dans [MONTANARI 1974] et [MACKWORTH 1977]. Un chemin est une séquence de k variables $(X_0, \ldots, X_k)$ (la même variable

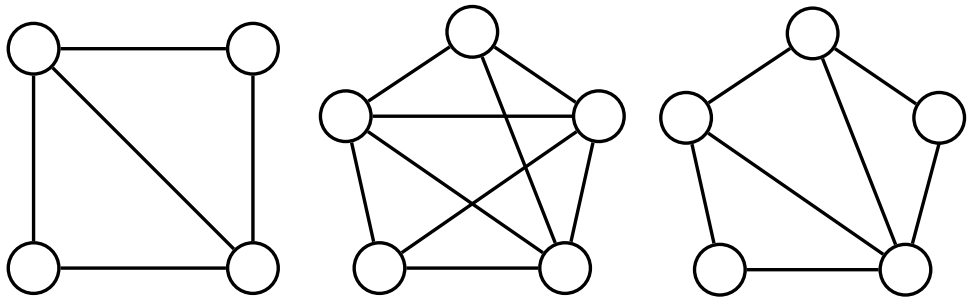


FIGURE 1.15 – Trois graphes triangulés

peut apparaître plusieurs fois dans la séquence). Un chemin est PC ssi pour chaque couple de variables (X_i, X_{i+i}) du chemin on peut trouver une valeur a_i et a_{i+1} dans le domaine de chaque variable telle que l'instanciation $\{(X_i, a_i), (X_{i+1}, a_{i+1})\}$ soit consistante. Un réseau est PC si tous les chemins possibles sont PC. Le nombre de chemins d'un CN est cependant très important, on se contente donc d'établir la chemin consistance sur un graphe complet. Dans ce cas, il suffit de prendre en compte les chemins de longueur 2 (c'est à dire les séquences d'exactement trois variables) pour obtenir un réseau chemin-consistant [MONTANARI 1974].

La consistance de chemin partielle (PPC) ne prend en compte que les chemins effectivement présents dans le graphe, c'est à dire que pour chaque couple de variables (X_i, X_{i+1}) du chemin, il existe une contrainte telle que $\mathrm{vars}(C) = \{X_i, X_{i+1}\}$. Il s'agit en quelque sorte d'une variante plus forte de CPC : CPC ne va contrôler que les chemins de longueur 2. En pratique il reste très difficile d'établir PPC sur un graphe quelconque d'après cette définition, puisque le nombre de chemins dans un graphe reste extrêmement important.

[BLIEK ET SAM-HAROUD 1999] montrent cependant que si un CN est triangulé et chaque chemin de longueur 2 (du graphe triangulé, c'est à dire les chemins liés par des contraintes existantes) est PC (c'est à dire que le CN triangulé est CPC), alors il est PPC. Un CN triangulé est un CN où chaque cycle (un chemin du graphe commençant et terminant par la même variable) de longueur strictement supérieure à 3 comporte une corde, c'est à dire une arête (une contrainte) reliant deux nœuds (deux variables) du graphe. La figure 1.15 donne des exemples de graphes triangulés. Pour trianguler un graphe, comme pour le compléter, on ajoute des contraintes universelles. Cependant, il faut ajouter bien moins de contraintes pour trianguler un graphe que pour le compléter.

Théorème 3. Soit $\mathcal{T}$ un opérateur de triangulation [2] et $\mathcal{G}$ un opérateur de complétion de graphe. $\mathcal{T}$ et $\mathcal{G}$ réalisent respectivent la triangulation et la complétion en ajoutant des contraintes universelles au graphe. L'opérateur de chemin consistance PC suppose qu'on réalise en fait la complétion d'un graphe avant de supprimer les multiplets correspondant aux instanciations chemin-inconsistantes :

$\mathrm{PC} = \mathrm{CPC} \circ \mathcal{G} = \mathrm{PPC} \circ \mathcal{G}$

On a :

$\mathrm{PC} \succ \mathrm{PPC} \succ \mathrm{CPC}$ (Mais PC nécessite de compléter le graphe et la PPC sur un graphe quelconque est difficile à établir)

$\mathrm{PPC} \circ \mathcal{T} = \mathrm{CPC} \circ \mathcal{T}$

De même pour les variantes fortes de PC, PPC et CPC (sPC, sPPC et sCPC).

2. On notera que la triangulation d'un graphe n'est pas unique

1.2.3 Consistance de chemin restreinte

L'objectif des propriétés de consistance de chemin restreinte (à ne pas confondre avec les consistances de chemin partielle ou limitée) est de supprimer plus de valeurs que AC tout en évitant les principaux inconvénients de PC. Même la complexité optimale de PC peut être prohibitive. La propriété RPC (*Restricted PC*) ne vérifie la consistance des relations seulement quand une relation est la dernière d'une valeur. De cette sorte, si la relation est inconsistante, la valeur peut être supprimée [BERLANDIER 1995].

Cette propriété a été étendue à la k-RPC : on ne cherche après un support chemin-consistant dans une contrainte que pour les valeurs qui ont au moins k supports dans cette contrainte. AC correspond à la 0-RPC et RPC à la 1-RPC. Finalement, un CN est Max-RPC si toutes les valeurs ont *au moins un* support chemin-consistant dans toutes les contraintes, quel que soit le nombre de supports [DEBRUYNE ET BESSIÈRE 1997A]. La propriété Max-RPC semble atteindre un excellent rapport « valeurs supprimées / temps » avec des algorithmes admettant une complexité temporelle dans le pire des cas optimale en $O(en + ed^2 + Kd^3)$ et une complexité spatiale dans le pire des cas en $O(ed + kd)$.

Théorème 4 ([DEBRUYNE ET BESSIÈRE 1997A])**.** $\mathrm{sPC} \succ \text{Max-RPC} \succ k\text{-RPC} \succ \mathrm{RPC} \succ \mathrm{AC}$

1.2.4 Singleton consistance d'arc (SAC)

Un certain intérêt a été porté au développement de consistances plus fortes que GAC. L'objectif est toujours de diminuer le plus possible la taille des instances en un minimum de temps. La singleton consistance d'arc (SAC pour Singleton Arc Consistency), introduite par [DEBRUYNE ET BESSIÈRE 1997B], a été particulièrement étudiée ces dernières années, celle ci semblant présenter un rapport valeurs supprimées / temps intéressant.

SAC simple

On peut remarquer que SAC est tout à fait applicable aux réseaux non-binaires, puisque on peut aussi bien se baser sur AC que sur GAC. La littérature ne fait en pratique jamais la distinction entre SAC et SGAC. La singleton consistance d'arc est définie ici en se basant sur GAC.

Définition 16 (Singleton consistance d'arc)**.** Étant donné un CN $P = (\mathscr{X}, \mathscr{C})$, une valeur X_a avec $X \in \mathscr{X}$ est singleton arc-consistante (SAC) ssi $\mathrm{GAC}(P|_{X=a}) \neq \bot$.

Un réseau P est SAC ssi $\forall X \in \mathscr{X}$, $\forall a \in \mathrm{dom}(X)$, X_a est SAC.

Théorème 5 ([DEBRUYNE ET BESSIÈRE 1997A])**.** $\mathrm{sPC} \succ \mathrm{SAC} \succ \text{Max-RPC}$

Les complexités données ici sont obtenues dans le cadre de réseaux binaires, bien que la définition et les algorithmes s'appliquent également aux réseaux non-binaires. L'algorithme de base SAC-1 introduit dans [DEBRUYNE ET BESSIÈRE 1997B] (algorithmes 7 et 8) présente une complexité dans le pire des cas de $O(en^2d^4)$. Nous apportons ici une légère amélioration, introduite dans [LECOUTRE *et al.* 2007A,] : en supposant que $\mathscr{X}$ est ordonné, $first(\mathscr{X})$ renvoie la première variable de $\mathscr{X}$, et $next\text{-}modulo(\mathscr{X}, X)$ renvoie la variable qui suit X dans $\mathscr{X}$ si elle existe, ou $first(\mathscr{X})$ sinon. Ainsi, l'algorithme se termine quand on a fait un tour complet de toutes les variables sans effectuer aucune modification. Une étude montre que la complexité optimale pour un algorithme de singleton consistance d'arc est en $O(end^3)$ et est atteinte par l'algorithme SAC-OPT au prix d'une complexité spatiale en $O(end^2)$. L'algorithme SAC-SDS relâche la complexité optimale ($O(end^4)$) pour une meilleure complexité spatiale ($O(n^2d^2)$) et un meilleur comportement en pratique dans la plupart des cas [BESSIÈRE ET DEBRUYNE 2005].

Algorithme 7 : SAC-1($P = (\mathscr{X},\mathscr{C})$: CN) : CN

```
1  P ← GAC(P,𝒳)
2  X ← first(𝒳)
3  marque ← X
4  répéter
5      si checkSAC(X) alors
6          P ← GAC(P,{X})
7          marque ← X
8          si dom(X) = ∅ alors retourner ⊥
9      X ← next-modulo(X,𝒳)
10 jusqu'à X = marque
11 retourner P
```

Algorithme 8 : checkSAC(P : CN, X : Variable) : Booléen

```
1  modif ← faux
2  pour chaque a ∈ dom(X) faire
3      si GAC(P|_{X=a},{X}) = ⊥ alors
4          retirer a de dom(X)
5          modif ← vrai
6  retourner modif
```

L'algorithme SAC-3 mêle recherche et inférence pour obtenir une complexité identique à celle de SAC-1 mais a généralement un meilleur comportement en pratique que SAC-SDS (sauf sur les problèmes aléatoires) et la possibilité de découvrir des solutions au CN pendant l'établissement de la Singleton consistance d'arc [LECOUTRE ET CARDON 2005].

Finalement, [BESSIÈRE ET DEBRUYNE 2008] ont proposé une extension de SAC nommée la singleton consistance d'arc propagée (SPAC), basée sur l'observation suivante : si $Y_b \notin \mathrm{AC}(P|_{X=a})$, cela correspond à la détection du nogood $\neg X = a \vee \neg Y = b$ et on peut en déduire que $X_a \notin \mathrm{AC}(P|_{Y=b})$. Il est alors possible d'exploiter cette inférence lorsque l'on teste la singleton arc-consistance de Y_b.

K-SAC et K-SAC faible [VAN DONGEN 2006]

On peut généraliser la propriété de singleton consistance d'arc aux instanciations :

Définition 17 (K-singleton consistance d'arc)**.** Soit un CN $P = \{\mathscr{X},\mathscr{C}\}$. Une valeur X_v de P est K-SAC ssi pour tout sous-ensemble variables (distinctes) de $\mathscr{X}$ $\{X_1, X_2, \ldots, X_{K-1}\}$ de longueur $K-1$, il existe une instanciation portant sur ces variables $\{(X_1, v_1), (X_2, v_2), \ldots, (X_{K-1}, v_{K-1})\}$ de longueur $K-1$ telle que $\mathrm{AC}(P|_{X=v \wedge X_1=v_1 \wedge X_2=v_2 \wedge \cdots \wedge X_{K-1}=v_{K-1}}) \neq \bot$.

P est K-singleton arc-consistant ssi toutes les valeurs de P sont K-singleton arc-consistantes.

0-SAC revient à la consistance d'arc, 1-SAC à la singleton consistance d'arc simple. 2-SAC est une consistance de relations plus forte que la consistance de chemin. [VAN DONGEN 2006] propose une version « faible » de la K-singleton consistance d'arc, plus facile à établir. K-SAC faible est une consistance de domaine et peut être définie comme suit :

Définition 18 (K-singleton consistance d'arc faible)**.** Soit un CN $P = \{\mathscr{X},\mathscr{C}\}$. Une valeur X_v de P

est K-SAC ssi il existe une instanciation $\{(X_1, v_1), (X_2, v_2), \ldots, (X_{K-1}, v_{K-1})\}$ de longueur $K-1$ telle que $\mathrm{AC}(P|_{X=v \wedge X_1=v_1 \wedge X_2=v_2 \wedge \cdots \wedge X_{K-1}=v_{K-1}}) \neq \bot$.

P est faiblement K-singleton arc-consistant ssi toutes les valeurs de P sont faiblement K-singleton arc-consistantes.

La 0-SAC faible reste équivalente à AC et 1-SAC faible à SAC classique. Pour $K \geq 2$, on obtient des consistances de domaine plus fortes que SAC. [VAN DONGEN 2006] propose un algorithme établissant K-SAC faible sur des réseaux binaires avec une complexité dans le pire des cas en $O\left(en^2d^{K+3} + (n-K)en^2d^4\right)$ (en supposant l'utilisation d'un algorithme de consistance d'arc sous-jacent optimal comme AC-2001). L'algorithme semble efficace en pratique, puisque l'auteur indique avoir montré pour la première fois, en moins d'une heure de calcul avec une machine cadencée à 2,8 GHz, l'inconsistance du problème très difficile $scen11$-$f1$ (le problème n'est pas faiblement 8-SAC).

1.2.5 Consistances aux bornes

Les consistances aux bornes ont été introduites pour les CSP continus, les CSP dont les variables peuvent prendre des valeurs dans le domaine des réels $\mathbb{R}$ (ou tout du moins des flottants $\mathbb{F}$). Les domaines sont alors stockés sous forme d'intervalles convexes $[a,b]$, indiquant la valeur minimale et maximale des valeurs du domaine. Comme dans le cas des CSP discrets, il existe de nombreux algorithmes plus ou moins forts de propagation de contraintes [HYVONEN 1992, FALTINGS 1994].

Le plus simple, la 2B Consistance (ou Hull Consistance), est une adaptation de la consistance d'arc, appliquée uniquement aux bornes des domaines [LHOMME 1993, BENHAMOU *et al.* 1994,]. Cette consistance nécessite cependant l'existence pour chaque paire (Contrainte, Variable) (C, X) de deux fonctions permettant le calcul des bornes min et max de l'ensemble des valeurs obtenues par projection de X sur l'ensemble des supports de C. Si de telles fonctions ne peuvent être obtenues, il faut décomposer le réseau de contraintes. La Box Consistance [BENHAMOU *et al.* 1994,] exploite l'arithmétique des intervalles ($[a,b] + [c,d] = [a+c, b+d]$, $[a,b] \times [c,d] = [\min(ac,ad,bd,bd), \max(ac,ad,bc,bd)]$...) et ne nécessite pas de décomposer les contraintes. La 2B et la Box consistances sont identiques quand aucune variable n'apparaît plusieurs fois dans l'expression d'une contrainte [COLLAVIZZA *et al.* 1999,].

On peut définir des consistances plus fortes que la 2B ou la Box consistance en affectant une variable à une de ses bornes et en contrôlant si le réseau obtenu est toujours consistant. Il s'agit de la 3B [LHOMME 1993] ou Bound Consistance. La définition de la 3B consistance peut être rapportée à la définition de SAC pour les CSP discrets. On peut finalement introduire une définition récursive de la kB Consistance avec $k \geq 2$ [LHOMME 1994].

1.3 Méthodes de recherche

On regroupe généralement les méthodes de recherche pour la résolution de problèmes combinatoires en général et CSP en particulier en deux grandes familles : algorithmes systématiques et algorithmes non-systématiques (parfois appelés algorithmes heuristiques). Les algorithmes non-systématiques sont des algorithmes généralement incomplets, adaptés aux problèmes satisfiables de très grande taille qui restent intraitables par des méthodes systématiques. Ils regroupent notamment la recherche locale, les algorithmes évolutionnaires (« génétiques ») ou basés sur les « colonies de fourmis ». Nous nous intéresserons ici aux algorithmes de recherche systématiques et aux algorithmes de recherche locale.

Algorithme 9 : MGAC-d(P = $(\mathscr{X},\mathscr{C})$: CN) : Booléen

```
1 si 𝒳 = ∅ alors retourner vrai
2 sélectionner X ∈ 𝒳
3 pour chaque a ∈ dom(X) faire
4 |   P' ← GAC(P|_{X=a}, {X})
5 |   si P' ≠ ⊥ ∧ MGAC(P'\X) alors
6 |   |   retourner vrai
7 retourner faux
```

1.3.1 Recherche systématique

La recherche systématique travaille sur des instanciations partielles. On part de l'instanciation vide, et on effectue des hypothèses, c'est à dire qu'on tente d'affecter une valeur à chaque variable. Quand on détecte une inconsistance (généralement, un domaine vide), on effectue un retour-arrière (*backtrack*), c'est à dire qu'on annule la dernière hypothèse (on désaffecte une variable) pour en essayer une autre. De cette manière, on construit un arbre de recherche. Chaque branche de l'arbre correspond à une hypothèse, ou a sa réfutation. Si on parvient de cette façon à affecter toutes les variables, on a trouvé une solution. Si toutes les hypothèses possibles ont été envisagées sans succès, on a prouvé que le problème n'avait pas de solution. L'ordre dans lequel les hypothèses (le choix d'une variable et d'une valeur à affecter) sont faites est extrêmement important. Des heuristiques sont amenées à effectuer ce choix à chaque nœud de l'arbre.

Pendant longtemps, les algorithmes systématiques de référence ont été le *Forward Checking* (FC) et sa généralisation aux contraintes n-aires nFC (n pour *non-binary*). Après chaque affectation de variable, on supprime les valeurs dans les variables voisines incompatibles avec la valeur choisie. L'inférence effectuée à chaque nœud d'une recherche par FC peut être vue comme une méthode d'inférence moins forte que AC. Au début des années 1990, l'algorithme FC avait été amélioré à plusieurs reprises par des notions de *retour-arrière intelligent*, et en particulier par le *retour-arrière dirigé par les conflits* (*Conflict Directed Backjumping* ou CBJ). FC-CBJ a longtemps été considéré comme le meilleur algorithme pour résoudre un CSP. L'idée est, à chaque fois qu'on rencontre une inconsistance au cours de la recherche, d'identifier la provenance de cette inconsistance et de revenir directement à la source du problème. Cependant, avec le développement d'heuristiques dynamiques et d'algorithmes de consistance d'arc plus performants que GAC-3, on s'est rendu compte que l'algorithme MGAC, qui maintient la consistance d'arc généralisée à chaque branche de l'arbre, était généralement plus efficace [SABIN ET FREUDER 1994, BESSIÈRE ET RÉGIN 1996].

Les premiers algorithmes MGAC étaient à *branchement d-aire* : chaque nœud de l'arbre correspond au choix d'une variable à instancier, puis on teste toutes les valeurs possibles pour cette variable, une par une. L'algorithme 9 donne un exemple d'un tel algorithme. $P\backslash X$ représente le CN obtenu de P en retirant la variable X. Cependant, il a été montré récemment qu'un branchement binaire était bien plus performant que le branchement d-aire [HWANG ET MITCHELL 2005]. Le branchement binaire consiste à tester une hypothèse d'affectation de type $X = a$, et, si l'hypothèse est réfutée, de supprimer a du domaine de X ($X \neq a$) avant d'établir à nouveau GAC. La réfutation consiste en la deuxième branche du nœud. Au nœud suivant, on peut choisir une nouvelle hypothèse, sur une variable quelconque. Cet variante de MGAC permet une bien meilleure flexibilité dans le choix des hypothèses de branchement, en particulier associé à des heuristiques adaptatives comme $wdeg$ (voir section 1.3.2 ci-dessous).

Un algorithme MGAC récursif à branchement binaire est décrit par l'algorithme 10. À chaque nœud de l'arbre, une hypothèse est considérée (ligne 3). L'hypothèse (lignes 4-5) et sa réfutation (lignes 7-8)

Algorithme 10 : MGAC-2(P = $(\mathscr{X},\mathscr{C})$: CN, $maxBT$: Entier) : Booléen

```
1 si maxBT < 0 alors lever Expiration
2 si 𝒳 = ∅ alors retourner vrai
3 sélectionner X_a tel que X ∈ 𝒳 et a ∈ dom(x)
4 P' ← GAC(P|_{X=a}, {X})
5 si P' ≠ ⊥ ∧ MGAC(P'\X, maxBT) alors
6     retourner vrai
7 P' ← GAC(P|_{X≠a}, {X})
8 retourner P' ≠ ⊥ ∧ MGAC(P', maxBT − 1)
```

forment alors les deux branches du nœud. On maintient systématiquement GAC (lignes 4 et 7). L'algorithme étant exponentiel ($O(d^n)$), on fixe un critère d'arrêt pour contrôler son exécution, ici le nombre maximal de retour-arrières autorisés $maxBT$. Si ce nombre est atteint, l'algorithme est interrompu sans qu'on sache si le problème est satisfiable ou non (ligne 1). Si toutes les variables ont été affectées, on a trouvé une solution : le problème est satisfiable (ligne 2).

À l'heure actuelle, maintenir des consistances plus fortes que GAC pendant la recherche n'est pas considéré comme compétitif dans le cas général. Ces consistances fortes sont donc généralement appliquées pendant une phase de prétraitement. De cette sorte, elles peuvent bénéficier à tout type d'algorithme de recherche.

1.3.2 Heuristiques pour la recherche systématique

On a indiqué plus haut que l'ordre dans lequel les hypothèses sont effectuées est très important. Les heuristiques chargées d'effectuer le choix des variables et des valeurs à affecter sont cruciales pour résoudre efficacement les CSP. En particulier, les algorithmes de choix de variable peuvent changer du tout au tout l'efficacité d'un algorithme MGAC. Ces algorithmes ont en effet le défaut de dépendre fortement des premières hypothèses effectuées. Si on commence par faire des hypothèses évidentes, on finira forcément par buter sur les parties les plus difficiles du réseau de contraintes. Il faudra alors revenir sur ces hypothèses faciles et de prouver à nouveau l'inconsistence de sous-problèmes difficiles. C'est ce qu'on appelle le *thrashing*. Le fait de faire beaucoup d'inférences à chaque nœud de l'arbre permet de limiter le thrashing (on détecte l'inconsistence de sous-problèmes beaucoup plus rapidement), mais des heuristiques évoluées vont aussi permettre à l'algorithme de se focaliser sur ces sous-problèmes difficiles et de les résoudre en priorité.

Plusieurs heuristiques de choix de variables ont été proposées. On peut en distinguer trois types :

- Heuristiques statiques : deg (*max-degree*) et sa variante $ddeg$ (*dynamic degree*) [3]
- Heuristique dynamiques : dom (*min-domain*), $brelaz$, $dom/ddeg$.
- Heuristiques adaptatives : $wdeg$ (*weighted degree*), $dom/wdeg$

Les heuristiques statiques peuvent être calculées *a priori*, avant la recherche. L'ordre n'évoluera pas pendant la recherche. Les heuristiques dynamiques sont recalculées à chaque nœud, tandis que les heuristiques adaptatives se basent sur des statistiques a priori indépendantes de l'arbre de recherche.

La première heuristique proposée fut dom (*min-domain*) [HARALICK ET ELLIOTT 1980]. L'idée est de choisir en priorité les variables les plus contraintes, c'est à dire les variables dont le domaine est le plus petit. L'espace de recherche total d'un réseau correspond au produit du domaine de toutes les variables. C'est en supprimant une valeur du domaine le plus petit que l'on réduit le plus l'espace de recherche. On

3. Bien que $ddeg$ soit généralement estampillée « dynamique », il est en pratique très facile de calculer avant la recherche l'implication du choix de chaque variable en termes de degré et de fixer alors l'ordre de manière statique.

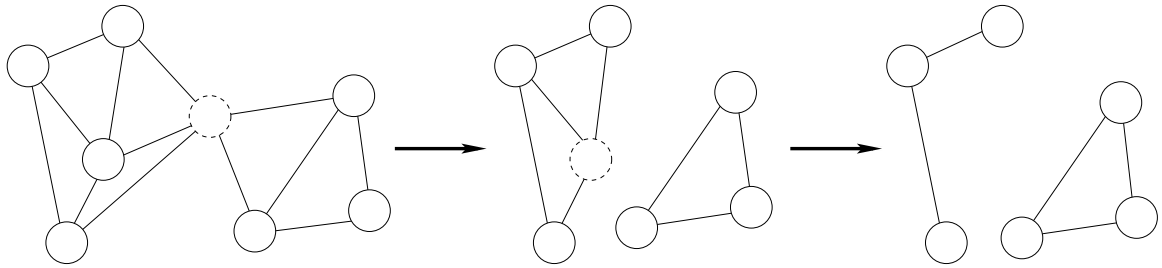


FIGURE 1.16 – Impact du choix d'une variable sur le graphe de contraintes.

Algorithme 11 : revise-wdeg(C : Contrainte, X : Variable) : Booléen

```
1 domainSize ← |dom(X)|
2 pour chaque v ∈ dom(X) faire
3     si ¬seekSupport(C, X_v) alors
4         retirer v de dom(X)
5 si |dom(X)| = 0 alors
6     wght[C] ← wght[C] + 1
7 retourner domainSize ≠ |dom(X)|
```

peut estimer que la taille de l'espace de recherche correspond au produit de la taille du domaine de toutes les variables : $R = \prod_{X \in \mathscr{X}} |\mathrm{dom}(X)| \in O(d^n)$. Si on supprime une valeur du domaine d'une variable X, la taille de l'espace de recherche $R' = R \times \frac{|\mathrm{dom}(X)|-1}{|\mathrm{dom}(X)|} = R \times (1 - \frac{1}{|\mathrm{dom}(X)|})$. On minimise donc R' en minimisant $|\mathrm{dom}(X)|$.

Dans le même ordre d'idée (choisir en priorité les variables les plus contraintes), il est efficace de commencer par les variables affectées par le plus grand nombre possible de contraintes. On peut également prendre en compte l'effet sur le réseau de contraintes de l'affectation d'une variable. En effet, une variable singleton peut tout simplement être ignorée dans le cadre de MGAC, puisque toutes les valeurs incompatibles avec ce singleton dans les autres variables ont été filtrées par GAC. On peut donc retirer cette variable du réseau, ainsi que toutes les contraintes ne présentant alors plus qu'une variable non affectée, ce qui a pour effet de faire évoluer le degré de chaque variable. De cette sorte, on peut « éclater » le réseau de contraintes en plusieurs sous-graphes qu'il est plus facile de traiter (voir figure 1.16). De plus, il suffit qu'un seul de ces sous-graphes soit inconsistant pour que l'ensemble du problème soit inconsistant. D'autre part, un ensemble arc-consistant de variables reliées par des contraintes de telle sorte que les contraintes ne forment pas de cycle (elles forment un arbre) est obligatoirement consistant.

On peut donc considérer l'heuristique statique deg, où le degré est calculé pour chaque variable au début de la recherche, ou l'heuristique dynamique $ddeg$. deg ou $ddeg$ ne sont cependant pas très efficaces seules, mais se combinent très bien avec l'heuristique dom, soit pour trancher les égalités (c'est le principe de l'heuristique $brelaz$ [BRELAZ 1979]), soit via un simple rapport $dom/ddeg$ [BESSIÈRE ET RÉGIN 1996].

De nombreux travaux ont montré que la dernière heuristique, $dom/wdeg$, était l'heuristique de choix de variables la plus robuste à l'heure actuelle [BOUSSEMART *et al.* 2004A, LECOUTRE *et al.* 2004, HULUBEI ET O'SULLIVAN 2005, VAN DONGEN 2005,]. L'idée est d'associer un compteur $wght[C]$ à chaque contrainte, et d'incrémenter ce compteur à chaque fois que la contrainte permet de vider un domaine (cf la version modifiée de $revise$ algorithme 11, lignes 5 et 6). Le *score* d'une variable est alors le résultat du rapport entre la taille du domaine et le *degré pondéré* de la variable. Le degré pondéré

Algorithme 12 : initP($P = (\mathscr{X}, \mathscr{C})$: CN) : Integer

```
1 pour chaque X ∈ 𝒳 faire
2     sélectionner v ∈ dom(X) | countConflicts(P|_{X=v}) est minimal
3     P ← P|_{X=v}
4 retourner countConflicts(P)
```

d'une variable est la somme du poids des contraintes associées à la variable, de sorte que plus d'une des variables associées à la contrainte ne soit pas encore assignée. L'heuristique $dom/wdeg$ choisit alors en priorité une variable dont le résultat de ce rapport est minimal. Le fait de pondérer les contraintes souvent falsifiées a tendance à exacerber les contraintes les plus *difficiles* du problème, et à identifier les sous-réseaux difficiles.

Résoudre plusieurs fois de suite le même problème avec une légère modification aléatoire des heuristiques peut engendrer d'importantes disparités dans le temps de résolution du problème. C'est le phénomène *Heavy-Tail* [GOMES *et al.* 2000, HULUBEI ET O'SULLIVAN 2005,]. Un problème relativement facile peut alors mettre beaucoup de temps à être résolu en raison de mauvais choix initiaux. Outre l'utilisation d'heuristiques adaptatives comme $dom/wdeg$, l'utilisation des redémarrages permet de limiter fortement ce phénomène. On fixe un critère d'arrêt à MGAC ($maxBT$). Si le problème n'a pu être résolu après $maxBT$ backtracks, on recommence avec de nouveaux choix initiaux. Dans le cas de $dom/wdeg$, le fait que les contraintes n'aient pas le même poids d'un redémarrage à l'autre suffit à modifier les choix initiaux. On fait progresser $maxBT$ géométriquement à chaque redémarrage de telle sorte que l'algorithme reste théoriquement complet.

1.3.3 Recherche locale

La recherche locale a déjà été envisagée pour résoudre les CSP [MINTON *et al.* 1992, GALINIER ET HAO 1997, GALINIER ET HAO 2004,], mais la littérature sur le sujet est bien plus maigre que sur la recherche systématique, bien que la recherche locale ait été longuement étudiée en recherche opérationnelle ou même pour résoudre le problème SAT. Contrairement aux algorithmes de recherche systématique comme MGAC, les techniques de recherche locale sont par nature incomplètes : si une solution existe, il n'existe aucune garantie de la trouver en temps fini, et l'absence de solution ne peut être prouvée. Cependant, sur les instances de très grande taille, la recherche locale reste en pratique la meilleure alternative.

Un algorithme de recherche locale fonctionne sur des *instanciations complètes* : une valeur est affectée à chaque variable, puis l'instanciation est itérativement *réparée* jusqu'à la découverte d'une solution. Une réparation implique en général le changement d'une valeur affectée à une variable, de telle sorte que le moins de contraintes possibles soient violées [MINTON *et al.* 1992,]. L'instanciation initiale des variables peut être générée aléatoirement. Cependant, pour que les premières réparations soient plus intéressantes, on utilise généralement une instanciation gloutonne telle que présentée par l'algorithme 12 : on cherche à minimiser le nombre de conflits dès l'initialisation. $countConflicts(P)$ renvoie le nombre de contraintes falsifiées par des variables instanciées.

La conception d'algorithmes de recherche locale efficaces nécessite la mise au point de structures de données évoluées permettant le développement d'algorithmes incrémentaux puissants, capables de garder une trace de la qualité de chaque réparation. On utilise par exemple une structure de données $\gamma(X, v)$ qui contient à tout moment le nombre conflits qu'amènerait la réparation consistant à affecter la valeur v à X [GALINIER ET HAO 1997]. Les algorithmes 13 et 14 décrivent la gestion de γ ($check(C)$ contrôle si C est satisfaite par l'instanciation actuelle des variables $\mathrm{vars}(C)$). Comme chaque réparation n'a un

Algorithme 13 : initγ(P = ($\mathscr{X}$,$\mathscr{C}$) : CN)

```
1 γ(X, v) ← 0 ∀X ∈ 𝒳 ∀v ∈ dom(X)
2 pour chaque C ∈ 𝒞 faire
3     pour chaque X ∈ vars(C) faire
4         pour chaque v ∈ dom(X) faire
5             si ¬check(C|X=v) alors
6                 γ(X, v) ← γ(X, v) + wght[C]
```

Algorithme 14 : updateγ(X : Variable, v_{old} : Value)

```
1 pour chaque C ∈ 𝒞 | X ∈ vars(C) faire
2     pour chaque Y ∈ vars(C) | X ≠ Y faire
3         pour chaque v_y ∈ dom(Y) faire
4             si check(C|Y=v_y) ≠ check(C|Y=v_y∧X=v_old) alors
5                 si check(C|Y=v_y) alors
6                     γ(Y,v_y) ← γ(Y,v_y) − wght[C]
7                 sinon
8                     γ(Y,v_y) ← γ(Y,v_y) + wght[C]
```

impact que sur les contraintes impliquant la variable modifiée X, il devient possible de sélectionner la réparation à effectuer et mettre à jour γ en $O(\Gamma(X)kd) \in O(n^{k-1}kd)$: c'est la complexité de l'algorithme 14 (l'initialisation de γ décrite par l'algorithme 13 se fait en $O(ekd)$).

Il arrive fréquemment qu'aucune réparation simple ne puisse améliorer l'affectation courante en termes de satisfaction de contraintes. Dans ce cas, un *minimum local* a été atteint. Le principal défi de la recherche locale consiste à trouver le meilleur moyen de sortir ou d'éviter les minima locaux afin de poursuivre la recherche. Diverses techniques ont été étudiées dans de précédents travaux :

Mouvements aléatoires Avec une probabilité p, une réparation est effectuée aléatoirement au lieu d'être choisie parmi les meilleures réparations possibles [MINTON *et al.* 1992,]. L'algorithme 15, appelé MCRW (*Min-Conflicts with Random Walks*), effectue une telle recherche locale. À chaque itération, une variable en conflit est sélectionnée aléatoirement (ligne 3 ; une variable en conflit est une variable telle qu'au moins une contrainte impliquant cette variable est en conflit). Avec une probabilité p, une valeur est alors sélectionné aléatoirement (ligne 5), ou, avec une probabilité $1 - p$ (ligne 7), la meilleure valeur possible est sélectionnée. Il arrive très souvent que la meilleure valeur possible était déjà celle qui était sélectionnée. Dans ce cas, il est inutile de mettre à jour les structures de données. Le coût en termes de temps de calcul de ces itérations est négligeable. Dans tous les cas, le choix de la variable à réparer à chaque itération se fait aléatoirement (ligne 3).

Liste Tabu Les réparations précédentes (un couple variable, valeur) sont enregistrées afin d'éviter les réparations pouvant mener à des affectations déjà visitées [GALINIER ET HAO 1997]. Un nombre limité de réparations (correspondant à la taille de la liste Tabu) est gardé en mémoire, et les plus anciens sont oubliés, de sorte qu'un grand nombre de réparations reste disponible à chaque itération. Le *critère d'aspiration* permet de choisir une réparation présente dans la liste Tabu si celle-ci permet d'obtenir une

Algorithme 15 : MCRW($P = (\mathscr{X},\mathscr{C})$: CN, $maxIterations$: Integer) : Booléen

```
1  nbConflicts ← initP(P) ; initγ(P) ; nbIterations ← 0
2  tant que nbConflicts > 0 faire
3      sélectionner X aléatoirement | X est en conflit
4      si random[0,1] < p alors
5          sélectionner v ∈ dom(X) aléatoirement
6      sinon
7          sélectionner v ∈ dom(X) | γ(X,v) est minimal
8      v_old ← valeur actuelle de X
9      si v ≠ v_old alors
10         P ← P|_{X=v}
11         nbConflicts ← γ(X,v)
12         updateγ(X,v_old)
13         si nbIterations++ > maxIterations alors lever Expiration
14 retourner vrai
```

Algorithme 16 : Tabu($P = (\mathscr{X},\mathscr{C})$: CN, $maxIterations$: Integer) : Booléen

```
1  nbConflicts ← initP(P) ; initγ(P) ; nbIterations ← 0
2  initialiser TABU aléatoirement
3  tant que nbConflicts > 0 faire
4      sélectionner X_v ∉ TABU∨ valide le critère d'aspiration | γ(X,v) est minimal
5      v_old ← valeur courante de X
6      insérer (X_{v_old}) dans TABU et effacer le plus ancien élément de TABU
7      P ← P|_{X=v}
8      nbConflicts ← γ(X,v)
9      updateγ(X,v_old)
10     si nbIterations++ > maxIterations alors lever Expiration
11 retourner vrai
```

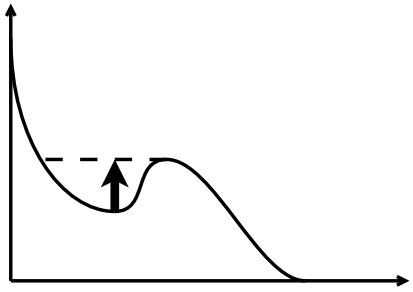


FIGURE 1.17 – Sortir d'un minimum local

affectation meilleure que toutes celles rencontrées jusqu'à présent. Cet algorithme semble particulièrement efficace sur les problèmes d'optimisation de type Max-CSP. L'algorithme 16 effectue une recherche Tabu.

Ces deux techniques nécessitent la mise au point d'importants paramètres, respectivement p et la taille de la liste Tabu.

Pondération de Contraintes La pondération des contraintes est issue de la *Breakout Method* proposée par [MORRIS 1993]. Quand un minimum local est rencontré, toutes les contraintes non satisfaites sont pondérées. Cette méthode permet de « lisser » les minima locaux afin de les éliminer efficacement et durablement. La figure 1.17 montre schématiquement le résultat attendu en représentant l'évolution du nombre de conflits à chaque itération.

Les problèmes structurés, particulièrement les problèmes industriels, sont généralement hétérogènes. Ils comportent des sous-réseaux comportant un très grand nombre de solutions et faciles à résoudre, et d'autres parties beaucoup plus difficiles, s'apparentant à un problème aléatoire au seuil. Pour résoudre ces instances, généralement de très grande taille, il convient d'identifier ces sous-réseaux difficiles. En pondérant les contraintes souvent falsifiées (et donc par intuition plus difficiles à satisfaire), elles seront exacerbées et l'algorithme cherchera à les satisfaire en priorité [EISENBERG ET FALTINGS 2003]. L'algorithme de recherche locale par pondération de contraintes, WMC (*Weighted Min-Conflicts*), est décrit par l'algorithme 17. La ligne 7 détecte un minimum local, et les lignes 8 à 12 incrémentent le poids des contraintes tout en maintenant γ, en $O(ekd)$ opérations. e est ici en fait borné par le nombre de contraintes en conflit au minimum local : la complexité est plus importante que pour une itération « normale », mais comme seules les contraintes en conflit sont prises en compte, les calculs à effectuer restent limités. Pour les problèmes de décision difficiles, à proximité du seuil, le nombre de variables en conflit à un moment donné est généralement très faible. On peut remarquer que WMC ne nécessite pas de paramètre particulier, si ce n'est le nombre maximal d'itérations avant un redémarrage.

Redémarrages Une limite au nombre d'itérations, $maxIterations$, est paramétrée, à la manière du paramètre $maxFlip$ de GSAT [SELMAN *et al.* 1992,] (une itération consiste soit à réparer l'affectation courante ou à pondérer les contraintes dans un minimum local), de sorte que l'algorithme puisse être exécuté plusieurs fois sur des affectations initiales différentes.

Les redémarrages restent très utiles pour éviter que de très mauvais choix initiaux n'orientent la recherche vers de larges zones de l'espace sans solution. Il arrive fréquemment, en particulier sur les problèmes structurés, qu'il faille faire un grand nombre de réparations pour passer d'un minimum local à l'autre, et il peut s'avérer très difficile d'en sortir. Le meilleur moyen reste parfois de « réparer » toutes les variables simultanément tout simplement en reprenant la recherche du début.

Algorithme 17 : WMC(P = ($\mathscr{X}$,$\mathscr{C}$) : CN, $maxIterations$: Entier) : Booléen

```
1  nbConflicts ← initP(P)
2  initγ(P)
3  nbIterations ← 0
4  tant que nbConflicts > 0 faire
5      sélectionner X_v | γ(X,v) est minimal
6      v_old ← valeur actuelle de X
7      si γ(X,v) ≥ γ(X,v_old) alors
8          pour chaque C ∈ 𝒞 | C est en conflit faire
9              wght[C]++ ; nbConflicts++
10             pour chaque Y ∈ vars(C) faire
11                 pour chaque w ∈ dom(Y) | ¬check(C|_{Y=w}) faire
12                     γ(Y,w)++
13     sinon
14         P ← P|_{X=v}
15         nbConflicts ← γ(X,v)
16         updateγ(P,v_old)
17     si nbIterations++ > maxIterations alors
18         lever Expiration
19 retourner vrai
```

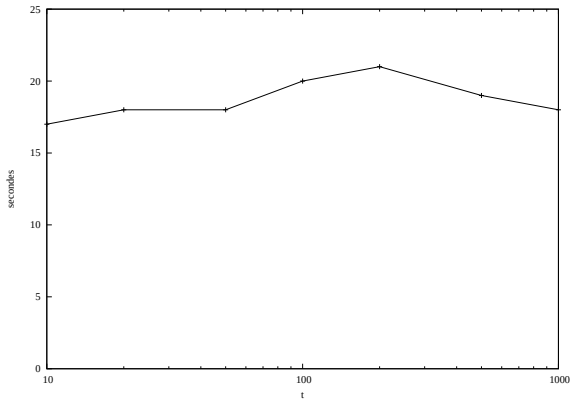


FIGURE 1.18 – Nombre d'instances RLFAP résolues en moins de 100 secondes par la recherche Tabu en fonction de la taille de la liste Tabu

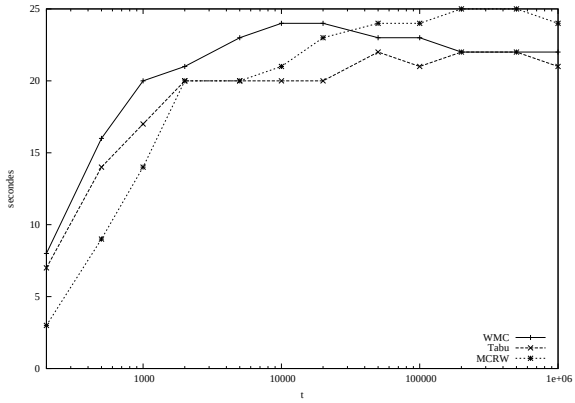


FIGURE 1.19 – Nombre d'instances RLFAP résolues en moins de 100 secondes en fonction du paramètre $maxIterations$

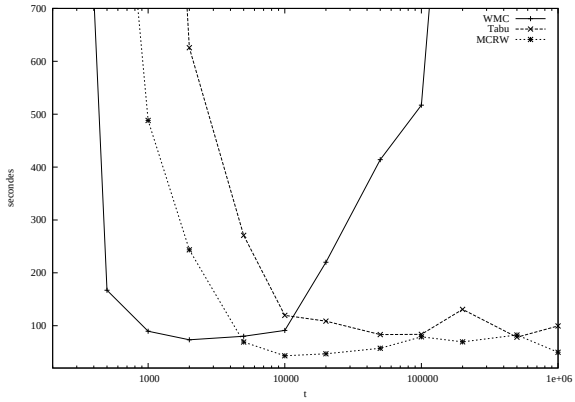


FIGURE 1.20 – Temps de résolution d'instances aléatoires en fonction du paramètre $maxIterations$

Pour mettre en avant l'influence des paramètres sur les algorithmes de recherche locale, nous montrons quelques figures obtenues lors du paramétrage des algorithmes implantés dans notre prouveur CSP4J, présenté dans le chapitre 4 de ce document.

La figure 1.18 indique le nombre d'instances de problèmes RLFAP qui sont résolues par la recherche Tabu en moins de 100 secondes avec notre prouveur sur une machine virtuelle Sun Java 6 fonctionnant sur un processeur AMD 64bits cadencé à 2 GHz, en fonction de la taille de la liste Tabu. Les problèmes testés sont les 25 problèmes satisfiables de la bibliothèque de problèmes RLFAP disponible [LECOUTRE 2006]. On observe un « plus haut » pour une taille de liste Tabu égale à 200. Sur des problèmes aléatoires de la classe $\langle 2; 50; 23; 262; 0{,}45\rangle$, la taille optimale de la liste Tabu est de 30 éléments. On procède de même pour paramétrer le paramètre p de l'algorithme MCRW, avec un comportement similaire.

Bien que WMC ne prenne pas de paramètre supplémentaire, le paramètre $maxIterations$ est évidemment important et l'efficacité de la résolution en dépend fortement. Si la valeur est trop faible, la recherche recommencera à chaque fois juste avant la découverte d'une solution. Si la valeur est trop forte, on risque de perdre beaucoup de temps dans de larges minima locaux. La figure 1.19 montre le nombre d'instances résolues sur les problèmes RLFAP en moins de 100 secondes en fonction du paramètre $maxIterations$ pour les trois algorithmes ($p = 0{,}4$ et $Tabu = 200$), et la figure 1.20 donne sur des problèmes aléatoires forcés à être satisfiables de la classe $\langle 2; 50; 23; 262; 0{,}45\rangle$ (ces problèmes sont au seuil, $p = 0{,}4$ et $Tabu = 30$). WMC semble plus sensible au paramètre $maxIterations$ que les deux autres algorithmes, en particulier sur les problèmes aléatoires.

Les algorithmes MCRW et Tabu sont particulièrement adaptés pour donner des solutions au problème Max-CSP. Les algorithmes restent évidemment incomplets et ne peuvent indiquer si une solution falsifiant un certain nombre de contraintes est la solution optimale (c'est à dire qu'il n'existe pas d'autre solution falsifiant moins de contraintes), mais ils peuvent rapidement atteindre des solutions proches ou égales à la solution optimale. WMC fonctionne mal sur les problèmes Max-CSP : bien qu'il puisse trouver rapidement des solutions d'assez bonne qualité, il va avoir tendance à éviter des solutions falsifiant un faible nombre de contraintes si celles-ci sont fortement pondérées.

1.3.4 Méthodes de recherche hybrides

Malgré l'utilisation de méthode d'inférence et d'heuristiques évoluées pour MGAC, il y a de nombreux cas où la recherche locale et les méthodes heuristiques donne de meilleurs résultats que la recherche systématique. Cependant, l'absence de garantie quant au résultat pour la recherche locale, l'impossibilité de prouver l'absence de solution ou d'utiliser des méthodes d'inférence puissantes comme GAC font que la recherche locale n'est pas non plus la panacée.

Considérant la dualité existant entre les deux méthodes, de nombreux efforts de recherche ont été effectués pour développer des algorithmes hybrides, tendant de profiter des avantages des deux principes. C'est un défi important dans le domaine des problèmes combinatoires [SELMAN *et al.* 1997,].

On peut à ce jour distinguer trois grandes méthodes d'hybridation entre la recherche systématique et la recherche locale :

1. Effectuer une recherche locale avant ou après une recherche systématique
2. Effectuer une recherche systématique améliorée par une recherche locale à un point ou un autre de la recherche, c'est à dire sur des assignations partielles
3. Effectuer une recherche locale globale tout en utilisant des méthodes de recherche systématique pour sélectionner un voisin ou filtrer l'espace de recherche

La manière la plus simple d'hybrider deux algorithmes quelconques reste de les lancer en parallèle, et de s'arrêter dès qu'un algorithme trouve une solution. Bien qu'efficace, cette méthode d'hybridation n'est pas particulièrement satisfaisante : on cherche avant tout à développer des méthodes qui permettent de

résoudre des problèmes au delà de la portée de l'une et l'autre méthode. Cette méthode peut s'apparenter à la première catégorie d'hybridation.

Deux travaux importants tombent également dans la première catégorie [MAZURE *et al.* 1998, EISENBERG ET FALTINGS 2003,]. L'idée consiste à utiliser des informations obtenues pendant une première phase de recherche locale pour guider les heuristiques de la recherche systématique.

Pour illustrer la seconde catégorie, on peut citer [SCHAERF 1997], où on effectue une recherche locale à chaque fois qu'une recherche systématique arrive dans une impasse.

La troisième catégorie semble particulièrement prometteuse, avec de bons résultats obtenus par les algorithmes *decision-repair* [JUSSIEN ET LHOMME 2000] ou encore IDB [PRESTWICH 2001]. Les algorithmes obtenus ont toutes les caractéristiques de la recherche locale (en particulier, ils sont incomplets), mais bénéficient de la performance des algorithmes de propagation de contraintes (GAC et FC, respectivement). Ces algorithmes s'inspirent en partie du retour-arrière intelligent : les algorithmes fonctionnent comme une recherche systématique de type MGAC ou FC, exploitant donc les méthodes d'inférence, mais on s'autorise ici à annuler n'importe quelle hypothèse à n'importe quel moment, et pas nécessairement la dernière hypothèse faite (même si c'est possible : en cela ces algorithmes généralisent MGAC et CBJ et peuvent donc être complets dans certains cas). Annuler une hypothèse quelconque et donc remettre en cause toutes les inférences précédentes, ou encore la conception de nouvelles heuristiques restent cependant problématiques.

Chapitre 2

Inférence autour de la consistance d'arc

La consistance d'arc (généralisée) joue un rôle central dans la résolution des CSP, tant dans son utilisation pendant la recherche avec un algorithme de type M(G)AC (cf section 1.3.1) que dans son utilisation dans des algorithmes d'inférence plus forts comme SAC (cf section 1.2.4). Parmi tous les algorithmes proposés pour établir (G)AC (cf section 1.2.1), les algorithmes à gros grain basés sur (G)AC-3 sont à l'heure actuelle considérés comme plus compétitifs. En particulier, l'algorithme (G)AC-3^{rm} semble à ce jour le plus efficace en pratique [LECOUTRE ET HEMERY 2007].

Dans cette section, nous nous intéressons à trois problématiques autour de la consistance d'arc : dans un premier temps, on cherche à améliorer les performances « brutes » des algorithmes de consistance d'arc, en particulier en cherchant à améliorer la complexité moyenne ou dans le meilleur des cas de l'algorithme AC-3^{rm}. On peut notamment chercher à améliorer la valeur des constantes qui n'apparaissent généralement pas dans les études de complexité et ne sont pas prises en compte par la notation O [GENT *et al.* 2006A,], c'est à dire la « vitesse brute » des algorithmes. C'est l'approche qui sera étudiée dans la section 2.1.

Une autre approche intéressante consiste à limiter l'impact de la consistance d'arc, à développer des algorithmes d'inférence moins forts, mais capables de se concentrer sur les inférences les plus utiles pour la résolution du problème. C'est l'approche étudiée dans la section 2.2, où on limite l'effort d'inférence aux bornes des domaines des variables.

Finalement, on peut aussi chercher à construire des consistances plus fortes qu'AC construites sur la consistance d'arc elle-même. Plusieurs méthodes d'inférence très populaires sont construites de cette manière, en particulier la consistance de chemin (l'algorithme proposé par [MCGREGOR 1979] utilise la consistance d'arc) ou la singleton consistance d'arc (SAC). Les sections 2.3 et 2.4 étudient de nouvelles consistances de relation nommées consistance duale (DC) et consistance duale conservative (CDC) qui utilisent la consistance d'arc au cœur de leurs définitions et algorithmes.

2.1 Établir la consistance d'arc par des opérations bit-à-bit

La contrainte binaire *max-supports* est construite de telle sorte que la valeur maximale de chaque variable impliquée supporte toutes les valeurs de l'autre variable (cf figure 2.1). Les problèmes basés sur cette contrainte mettent en évidence une des limitations d'AC-3 : pour prouver qu'un réseau de n variables de d valeurs, liées par e contraintes *max-supports* est AC en utilisant AC-3 ou une de ses variantes, il faut exécuter exactement $2e.(d^2 - d + 1)$ tests de contraintes. Si l'on considère maintenant que le domaine courant de chaque variable est représenté par un ensemble de *bits* : chaque bit est associé à une valeur du domaine et indique si cette valeur est présente ou non. On peut alors représenter le domaine d'une variable sous forme d'un *vecteur de bits*. On peut également représenter les contraintes

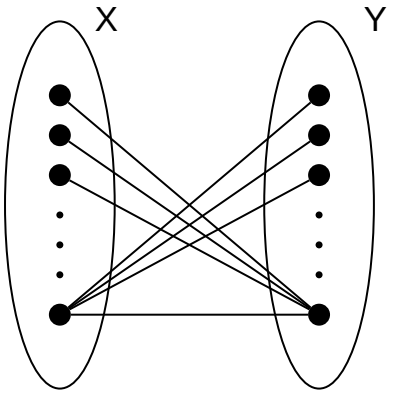


FIGURE 2.1 – Une contrainte *max-support* : la plus grande valeur de X supporte toutes les valeurs de Y et vice-versa.

$Instances$		$AC3$	$AC3^{rm}$	$AC2001$	$AC3^{bit}$
$\langle 250,50,5000 \rangle$	cpu	1,58	1,56	1,61	0,05
	$\#ops$	$24{,}5M$	$24{,}3M$	$24{,}5M$	$0{,}5M$
$\langle 250,100,5000 \rangle$	cpu	6,17	6,15	6,26	0,10
	$\#ops$	$99{,}0M$	$98{,}5M$	$99{,}0M$	$2{,}0M$
$\langle 500,50,10000 \rangle$	cpu	3,11	3,11	3,21	0,11
	$\#ops$	$49{,}0M$	$48{,}5M$	$49{,}0M$	$1{,}0M$
$\langle 500,100,10000 \rangle$	cpu	12,29	12,27	12,48	0,19
	$\#ops$	$198{,}0M$	$197{,}0M$	$198{,}0M$	$4{,}0M$

TABLE 2.1 – Établir la consistance d'arc sur des instances *max-supports*

binaires en extension sous forme de vecteurs de bits. On associe à chaque triplet (C_{XY}, X, a) un vecteur de bits indiquant si chaque valeur de Y est compatible avec a (a est une valeur du domaine de X et C_{XY} une contrainte impliquant X et Y). Pour chercher un support de (C_{XY}, X, a), il suffit alors d'appliquer l'opérateur *bit-à-bit* ET sur deux vecteurs. Si le résultat de l'opération diffère de ZERO (un vecteur dont tous les bits sont à 0), on sait qu'un support existe.

On peut considérer que les vecteurs de bits sont équivalents à un tableau de *mots CPU*, l'unité de base de l'architecture d'un ordinateur. Chaque opération bit-à-bit entre deux mots CPU représente x tests de contraintes, x étant la taille d'un mot CPU (généralement, 32 ou 64 sur un ordinateur moderne). Pour l'exemple ci-dessus, il faut donc x fois moins d'opérations pour établir la consistance d'arc en utilisant ce principe, que nous nommerons AC-3^{bit}, qu'avec un algorithme AC-3, AC-2001 ou AC-3^{rm} classique. Le tableau 2.1 montre quelques résultats expérimentaux obtenus pour le problème *max-supports* (chaque instance étant décrite sous la forme $\langle n, d, e \rangle$) sur un processeur 64-bits. $\#ops$ indique le nombre d'opérations bit-à-bit effectuées par AC-3^{bit} ou le nombre de tests de contrainte effectués par AC-3, AC-3^{rm} et AC-2001. Comme on pouvait s'y attendre, AC-3^{bit} est environ 60 fois plus efficace, bien que AC-3^{bit}, et AC-3 aient la même complexité $O(ed^3)$ (AC-3^{rm} atteint une complexité en $O(ed^2)$ sur ce problème).

L'idée d'exploiter les opérations bit-à-bit pour accélérer les calculs n'est pas nouvelle. En particulier, [MCGREGOR 1979] indiquait que les vecteurs de bits pouvaient être utilisés pour représenter les domaines et les ensembles de supports comme décrit ci-dessus. Des optimisations similaires étaient également mentionnées par [ULLMANN 1976] ou [BLIEK 1997]. Notre principale contribution est donner une description précise de l'exploitation des opérateurs bit-à-bit pour établir la consistance d'arc et pour

montrer à partir d'une vaste campagne d'expérimentations, que cette approche est réellement efficace.

Les travaux présentés dans cette section ont été réalisés en coopération avec Christophe Lecoutre et Lakhdar Saïs et ont été soumis au journal Constraint Programming Letters [LECOUTRE ET VION 2008].

2.1.1 Représentation binaire

Dans cette section, nous utiliserons la notion de cn-valeurs :

Définition 19 (cn-valeur)**.** Soit $P = (\mathscr{X},\mathscr{C})$ un CN. Une cn-valeur de P est un triplet de la forme (C, X, a), où $C \in \mathscr{C}$, $X \in \mathrm{vars}(C)$ et $a \in \mathrm{dom}(X)$.

Cette section introduit quelques détails sur la représentation binaire des domaines et des contraintes. Nous considérons que les vecteurs de bits sont représentés sous la forme d'un tableau de mots CPU (l'unité de base de l'architecture d'un ordinateur). Tous les langages de programmation ne permettent pas l'utilisation directe de vecteurs de bits comme structure de données. D'autre part, il reste plus efficace d'effectuer les calculs utilisant les opérateurs bit-à-bit sur des tableaux de mots CPU que sur des vecteurs de bits.

Sans perte de généralité, nous considérerons ici que l'ordinateur est équipé avec un processeur 64-bits. Cela signifie que la déclaration d'un tableau en langage Java serait $long[]$, puisque qu'un $long$ correspond à un mot de 64 bits.

Représentation des domaines

En utilisant un mécanisme de copie [SCHULTE 1999] pour gérer les domaines pendant une recherche systématique par retour-arrière, on peut associer un simple bit à chaque valeur de chaque domaine. Plus précisément, un bit peut être associé avec l'index (en commençant à 1) de n'importe quelle valeur d'un domaine. Quand ce bit est mis à 1 (respectivement 0), cela signifie que la valeur correspondante est présente dans le domaine (respectivement absente de celui-ci). En utilisant un tableau de mots CPU, il est alors possible de représenter un domaine de manière compacte. Nous appellerons ces tableaux la *représentation binaire des domaines*. Pour tout variable X, la complexité spatiale est alors en $\Theta(|\mathrm{dom}(X)|)$, ce qui est optimal.

Un autre mécanisme utilisé dans de nombreux logiciels de programmation par contraintes est appelé *trailing*, plus précisément décrit dans [VAN HENTENRYCK *et al.* 1992, LECOUTRE ET SZYMANEK 2006,]. La complexité spatiale de cette représentation est également en $\Theta(|\mathrm{dom}(X)|)$ et permet d'effectuer toutes les opérations élémentaires (déterminer si une valeur est présente, supprimer une valeur, ajouter une valeur, etc.) en $O(1)$. Ajouter et maintenir les structures pour la représentation binaire des domaines ne modifie pas la complexité dans le pire des cas en espace ni en temps, comme indiqué ci-dessous.

Pour représenter les domaines, nous introduisons un nouveau tableau à deux dimensions nommé $bitDom$ qui associe à chaque variable X la représentation binaire $bitDom[X]$ de $\mathrm{dom}(X)$.

- Pour supprimer la i^{e} valeur de $\mathrm{dom}(X)$, la seule opération nécessaire sur la structure $bitDom$ est la suivante :

$$bitDom[X][i \ \mathrm{div}\, 64] \leftarrow bitDom[X][i \ \mathrm{div}\, 64] \,\mathrm{ET}\, masks0[i \ \mathrm{mod}\, 64]$$

- Pour restaurer la i^{e} valeur de $\mathrm{dom}(X)$, l'opération est la suivante :

$$bitDom[X][i \ \mathrm{div}\, 64] \leftarrow bitDom[X][i \ \mathrm{div}\, 64] \,\mathrm{OU}\, masks1[i \ \mathrm{mod}\, 64]$$

L'opérateur div réalise la division entière, mod calcule le reste de la division, OU (respectivement ET) est l'opérateur bit-à-bit qui réalise l'opération logique de disjonction $\vee$ (respectivement la conjonction $\wedge$). La structure constante $masks1$ (respectivement $masks0$) est un tableau prédéfini de 64 mots

$masks1$

1	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001
2	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010
3	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0100
...	
63	0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
64	1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

$masks0$

1	1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1110
2	1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1101
3	1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1011
...	
63	1011 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
64	0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111

$ZERO$

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

TABLE 2.2 – $masks1$, $masks0$ et $ZERO$

CPU dont la i^{e} case contient une séquence de 64 bits à 0 (respectivement, 1), à l'exception du i^{e} bit, qui est forcé à 1 (respectivement 0) : voir table 2.2.

Représentation des contraintes

Nous ne considérerons que les contraintes binaires (bien qu'il soit envisageable de représenter de cette façon des contraintes d'arité relativement faible). On peut représenter une contrainte binaire en extension en utilisant soit un tableau de booléens à deux dimensions, soit une liste (ensemble) de multiplets, soit en intention par une formule prédicat.

Nous utilisons ici un tableau à deux dimensions nommé $bitSup$. Pour chaque cn-valeur (C, X, a), $bitSup[C, X, a]$ contient la représentation binaire des supports (du problème initial) de X_a dans C. Pour simplifier la présentation et sans perte de généralité, on peut supposer que les index et les valeurs correspondent (i.e. la i^{e} valeur du domaine de chaque variable est égale à i). Si C est telle que $\text{vars}(C) = \{X, Y\}$, alors $(a, b) \in \text{rel}(C)$ ssi le b^{e} bit de $bitSup[C, X, a]$ et mis à 1.

Si les contraintes sont initialement données au prouveur en extension, construire le tableau $bitSup$ ne présente pas de difficulté particulière. Par contre, si les contraintes sont données en intention, il faut alors effectuer chaque test de contrainte possible en évaluant le prédicat pour les instanciations correspondant à chaque multiplet de la contrainte afin de construire $bitSup$. En supposant que chaque test de contrainte est effectué en temps constant, cela représente une initialisation en $O(ed^2)$. Cependant, si plusieurs prédicats et signatures de contraintes sont identiques (i.e. les produits cartésiens des domaines des variables impliquées par la contrainte sont identiques), certaines sous-parties de $bitSup$ peuvent être partagées, ce qui représente des gains en espace et en temps d'initialisation importants.

La complexité en espace dans le pire des cas de la représentation binaire des contraintes est de $O(ed^2)$, et est de $O(d^2)$ dans le meilleur des cas (si toutes les contraintes sont identiques et les représentations peuvent être partagées). Le pire des cas correspond généralement à des problèmes aléatoires sans structure, alors que le meilleur des cas pourra se rencontrer sur des instances fortement structurées (académiques ou réelles). Par exemple, dans un problème de coloration de graphes, toutes les contraintes et tous les domaines sont identiques.

2.1.2 Exploiter les représentations binaires

$\mathrm{dom}^{init}(X)$ représente le domaine initial de la variable X (c'est à dire, avant toute opération de recherche et d'inférence). Pour tout tableau t, $t[1]$ indique le premier élément et $t.length$ sa taille. Nous pouvons maintenant utiliser ces structures de données pour effectuer efficacement plusieurs types de calculs en utilisant des opérateurs bit-à-bit, et en particulier dans les trois contextes suivants :

Détection de sous-domaines Les instructions suivantes permettent de déterminer si le domaine d'une variable X est inclus dans le domaine d'une autre variable Y (telles que $\left|\mathrm{dom}^{init}(X)\right| = \left|\mathrm{dom}^{init}(Y)\right|$) :

pour chaque $i \in \{1, \ldots, bitDom[X].length\}$ **faire**
  si $bitDom[X][i]$ OU $bitDom[Y][i] \neq bitDom[Y][i]$ **alors**
    retourner faux
retourner vrai

Ce type de calcul peut être particulièrement intéressant, par exemple pour implanter une méthode de cassage de symétries par détection de dominance [FOCACCI ET MILANO 2001, FAHLE *et al.* 2001,]. Dans ce cas, on compare le domaine courant d'une variable avec un domaine enregistré précédemment (de la même variable ou non). On peut alors efficacement déterminer si un état est dominé par un autre.

Substitution La séquence d'instructions suivante peut être utilisée pour déterminer si une valeur X_a est substituable par une valeur X_b par rapport à une contrainte C (impliquant X). Une valeur est substituable par une autre si ses supports sont un sous-ensemble des supports de l'autre valeur. Par exemple, dans une contrainte $max\text{-}supports$ représentée figure 2.1, toutes les valeurs sont substituables par la dernière valeur de chaque domaine. Si cette propriété de substituabilité est vraie pour toutes les contraintes impliquant la variable X, et s'il n'existe pas de solution avec la valeur substituante, il n'y aura pas non plus de solution avec les valeurs substituables. Cette technique permet de réduire substantiellement l'espace de recherche [FREUDER 1991, BELLICHA *et al.* 1994, COOPER 1997,].

pour chaque $i \in \{1, \ldots, bitDom[X].length\}$ **faire**
  si $bitSup[C,X,a][i]$ OU $bitSup[C,X,b][i] \neq bitSup[C,X,b][i]$ **alors**
    retourner faux
retourner vrai

Détection de supports Finalement, la séquence d'instructions suivante permet de déterminer si une valeur X_a admet au moins un support dans une contrainte C (impliquant X et une seconde variable Y) :

pour chaque $i \in \{1, \ldots, bitDom[Y].length$ **faire**
  si $bitSup[C,X,a][i]$ ET $bitDom[Y][i] \neq$ ZERO **alors**
    retourner vrai
retourner faux

ZERO représente un mot CPU dont tous les bits sont mis à 0 (cf table 2.2). Cette manière de rechercher un support est mentionnée par [MCGREGOR 1979]. Nous proposons ici une description détaillée de cette approche.

Il est intéressant de remarquer que pour toutes les opérations décrites ci dessus, il est parfois possible de renvoyer une réponse booléenne même si tous les éléments des domaines n'ont pas été passés en revue. Par exemple, pour les trois calculs indiqués ci-dessus, il est possible d'obtenir un résultat dès la

Algorithme 18 : seekSupport-bit(C : Contrainte, X_v : Variable$_{valeur}$) : Booléen

```
1 Soit Y la variable telle que vars(C) = {X,Y}
2 pour chaque i ∈ {1,...,bitDom[Y].length} faire
3     si bitSup[C,X,a][i] ET bitDom[Y][i] ≠ ZERO alors
4         retourner vrai
5 retourner faux
```

première utilisation d'un opérateur bit-à-bit (pour $i = 1$). Cela semble naturel, mais il ne faut pas perdre de vue qu'effectuer une opération sur un vecteur de bits, puis comparer le résultat avec un autre vecteur de bits peut être bien plus coûteux.

2.1.3 AC-3^{bit} : une optimisation simple de AC-3

Dans cette section, nous montrons comment adapter l'algorithme AC-3 pour exploiter les opérations bit-à-bit. L'objectif du nouvel algorithme, nommé AC-3^{bit}, est d'économiser une grande quantité d'opérations (tests de contraintes), et donc de temps CPU.

L'algorithme AC-3 de base a été présenté dans la section 1.2.1 page 19. Les algorithmes AC-3 optimisés conservent la même « base » présentée par l'algorithme 1 (page 21), seule la fonction $seekSupport$ change (et se limite aux contraintes binaires).

L'algorithme 18 présente la recherche d'un support utilisant les opérateurs bit-à-bit. Il utilise simplement le code présenté dans la section 2.1.2 ci-dessus.

Proposition 1. La complexité temporelle dans le pire des cas de AC-3^{bit} est en $O(ed^3)$

La preuve est immédiate. On peut malgré tout observer qu'en pratique, AC-3^{bit} peut être bien plus efficace que les autres variantes d'AC-3.

Observation. Le nombre d'opérations bit-à-bit effectuées par AC-3^{bit} peut être jusqu'à x fois moindre que le nombre de tests de contraintes effectués par AC-3, AC-2001 et AC-3^{rm}, où x est la taille d'un mot CPU dans l'architecture de l'ordinateur considéré.

2.1.4 Expérimentations

Pour montrer l'intérêt de l'algorithme introduit dans cette section (et plus généralement, l'intérêt pratique de l'utilisation des opérations bit-à-bit), nous avons effectué une vaste expérimentation sur une machine virtuelle Sun Java 5.0 pour Linux fonctionnant sur un CPU Intel i686 cadencé à 2,4 GHz et équipé de 512 Mio de RAM sur un ensemble de problèmes aléatoires, académiques et issus de problèmes réels [LECOUTRE 2006]. Les performances [4] ont été mesurées en termes de temps CPU (cpu, exprimé en secondes) et la mémoire en Mebibytes (mem).

Nous avons implanté les différents algorithmes de consistance d'arc AC-3, AC-2001, AC-3^{rm} et AC-3^{bit} dans la plate-forme Abscon [LECOUTRE *et al.* 2005,] et comparé ceux-ci sur l'algorithme de recherche MAC. Tous les algorithmes de consistance d'arc bénéficient de la propriété de *condition support* correspondant à la Proposition 1 de [BOUSSEMART *et al.* 2004C,] et à l'Équation 1 de [MEHTA ET VAN DONGEN 2005B]. L'heuristique de choix de variable choisie pour MAC est $dom/wdeg$, et l'heuristique de choix de valeurs $min\text{-}conflicts$ statique ([MEHTA ET VAN DONGEN 2005A]). Aucun mécanisme de redémarrage n'a été utilisé.

4. Dans notre expérimentation, tous les tests de contraintes sont effectués en temps constant et sont aussi efficaces que possible puisque les contraintes sont représentées en extension sous forme de matrices.

		MAC avec			
		AC-2001	AC-3	AC-3^{rm}	AC-3^{bit}
$\langle 40; 8; 753; 0{,}1\rangle$	cpu	13,8	9,8	10,4	7,7
	mem	11	9,5	10	9,5
$\langle 40; 11; 414; 0{,}2\rangle$	cpu	19,6	15,0	14,5	10,0
	mem	8,8	8,0	8,4	8,0
$\langle 40; 16; 250; 0{,}35\rangle$	cpu	21,6	18,5	16,1	9,7
	mem	8,5	7,9	8,2	7,9
$\langle 40; 25; 180; 0{,}5\rangle$	cpu	28,9	27,8	21,2	11,5
	mem	8,4	7,9	8,2	7,9
$\langle 40; 40; 135; 0{,}65\rangle$	cpu	21,1	22,0	15,4	7,8
	mem	8,5	8,0	8,2	8,1
$\langle 40; 80; 103; 0{,}8\rangle$	cpu	16,6	19,5	12,2	5,0
	mem	10	9,5	9,8	9,6
$\langle 40; 180; 84; 0{,}9\rangle$	cpu	24,3	36,6	18,4	6,7
	mem	15	14	14	14

TABLE 2.3 – Résultats moyens sur des instances aléatoires : 100 instances par classe, temps cpu en secondes et mem(oire) en Mio.

Nous avons tout d'abord considéré 7 classes d'instances aléatoires binaires situées au seuil (environ la moitié des instances sont satisfiables). Pour chaque classe $\langle 2, n, d, e, t\rangle$, le nombre de variables n a été fixé à 40, la taille des domaines d entre 8 et 180, le nombre de contraintes e entre 753 et 84 (et donc la densité entre 0,1 et 0,96 pour que les instances soient au seuil) et la dureté t entre 0,1 et 0,9.

La première classe $\langle 40; 8; 753; 0{,}1\rangle$ correspond à des instances denses impliquant des contraintes de dureté faible, alors que la septième et dernière classe $\langle 40; 180; 84; 0{,}9\rangle$ correspond à des instances éparses impliquant des contraintes de dureté forte. Dans la table 2.3, on peut observer que même avec des domaines réduits (par exemple $d = 8$), MAC-3^{bit} est l'algorithme le plus rapide. MAC-3^{bit} est de 2 à 4 fois plus rapide que MAC-2001 et de 50% à 3 fois plus efficace que MAC-3^{rm}.

Le bon comportement de MAC-3^{bit} est confirmé sur différentes séries d'instances structurées. En effet, la table 2.4 permet de constater qu'une fois encore, MAC-3^{bit} est plus efficace que les autres algorithmes. C'est particulièrement vrai pour les instances de Job Shop des séries $enddr1$ et $enddr2$. Ceci peut être expliqué par le fait que la taille moyenne des domaines pour ces instances est d'environ 120 valeurs, ce qui signifie que sur un processeur à 64 bits, seulement deux opérations sont nécessaires pour rechercher un support.

Finalement, nous présentons les résultats obtenus sur plusieurs instances académiques ou issues de problèmes réels difficiles (table 2.5). L'intérêt d'utiliser AC-3^{bit} apparaît clairement sur une instance comme $knights$-50-25. Il est également intéressant d'observer que l'écart entre AC-3^{bit} et les autres algorithmes augmente avec la difficulté des instances des séries $scen11$-fi. En effet, là où les algorithmes se comportent de manière similaire sur l'instance plus facile $scen11$-$f10$, AC-3^{bit} est deux fois plus rapide que les autres algorithmes de consistance d'arc sur l'instance plus difficile $scen11$-$f4$. La tendance apparaît clairement en regardant attentivement les résultats obtenus sur les instances intermédiaires $scen11$-$f8$ et $scen11$-$f6$.

		MAC avec			
		AC-2001	AC-3	AC-3^{rm}	AC-3^{bit}
blackHole-4-4 (10 instances)	cpu	1,5	1,4	1,4	0,9
	mem	8,6	7,9	8,7	7,9
driver (7 instances)	cpu	3,9	3,0	3,1	2,8
	mem	35	24	56	24
ehi-85 (100 instances)	cpu	1,8	0,9	1,1	0,7
	mem	30	19	38	19
ehi-90 (100 instances)	cpu	1,7	0,9	1,1	0,7
	mem	31	20	39	20
jobshop enddr1 (10 instances)	cpu	1 616,0	1 694,0	1 218,0	453,0
	mem	14	13	14	13
jobshop enddr2 (6 instances)	cpu	1 734,0	2 818,0	1 491,0	568,0
	mem	15	14	15	14
geom (100 instances)	cpu	12,4	10,8	8,9	5,8
	mem	11	10	11	10
hanoi (5 instances)	cpu	1,0	1,2	1,1	0,5
	mem	13	11	12	12
qwh-20 (10 instances)	cpu	266,0	183,0	242,0	153,0
	mem	33	21	44	21

TABLE 2.4 – Résultats moyens sur des instances structurées : temps cpu en secondes et mem(oire) en Mio.

Algorithme 19 : seekSupport-bit+rm(C : Contrainte, X_a : Variable$_{valeur}$) : Booléen

```
1 Soit Y la variable telle que vars(C) = {X,Y}
2 i ← res[C, X_a]
3 si bitSup[C, X, a][i] ET bitDom[Y][i] ≠ ZERO alors
4     retourner vrai
5 pour chaque i ∈ {1,...,bitDom[Y].length} faire
6     si bitSup[C, X, a][i] ET bitDom[Y][i] ≠ ZERO alors
7         res[C, X_a] ← i
8         retourner vrai
9 retourner faux
```

		MAC avec			
		AC-2001	AC-3	AC-3^{rm}	AC-3^{bit}
Instances académiques					
knights-50-9	cpu	85	1 148	109	36
	mem	27	23	23	23
knights-50-25	cpu	$> 1\,200$	$> 1\,200$	$> 1\,200$	211
	mem				28
pigeons-11	cpu	55	53	57	44
	mem	21	21	21	21
pigeons-12	cpu	656	547	591	484
	mem	21	21	21	21
queenAttacking-6	cpu	123	125	128	79
	mem	21	21	25	21
queenAttacking-7	cpu	407	436	381	263
	mem	25	22	25	22
Instances issues de problèmes réels					
e0ddr2-10-by-5-1	cpu	257	316	177	68
	mem	23	23	23	23
enddr2-10-by-5-1	cpu	178	263	143	61
	mem	23	23	23	23
scen11-f10	cpu	5	5	6	6
	mem	33	29	45	29
scen11-f8	cpu	11	11	12	9
	mem	33	29	45	29
scen11-f6	cpu	82	76	75	47
	mem	33	29	45	29
scen11-f4	cpu	1 250	1 233	1 106	670
	mem	33	29	45	29

TABLE 2.5 – Résultats sur des instances structurées difficiles : temps cpu en secondes et mem(oire) en Mio.

$Instances$		AC-2001	AC-3	AC-3^{rm}	AC-3^{bit}	AC-3^{bit+rm}
domino-500-500	cpu	12,7	403,0	9,4	4,3	3,7
	mem	$27M$	$23M$	$27M$	$23M$	$23M$
domino-800-800	cpu	48,4	2 437,0	34,5	13,4	8,7
	mem	$49M$	$33M$	$41M$	$33M$	$33M$
domino-1000-1000	cpu	89,5	5 911,0	62,4	25,1	14,3
	mem	$66M$	$42M$	$54M$	$42M$	$46M$
domino-2000-2000	cpu	678,0	$>$ 18 000,0	443,0	289,0	91,0
	mem	$210M$		$156M$	$117M$	$132M$
domino-3000-3000	cpu	2 349,0	$>$ 18 000,0	1 564,0	1 274,0	278,0
	mem	$454M$		$322M$	$240M$	$275M$

TABLE 2.6 – Établir la consistance d'arc sur des instances Domino

2.1.5 Et les résidus ?

On peut ici se demander si l'exploitation des résidus présente toujours un intérêt pour les instances binaires. En effet, pour des domaines jusqu'à 300 valeurs, déterminer si une cn-valeur admet un support nécessite moins de 5 opérations (sur une architecture 64 bits). C'était bien le cas sur la plupart des séries et instances présentées ci dessus. En conséquence, AC-3^{bit} était toujours plus rapide que AC-3^{rm}. Cependant, quand les domaines deviennent plus importants, il peut devenir pénalisant d'exploiter des opérations bit-à-bit seules. C'est pourquoi nous proposons de combiner celles-ci avec l'exploitation des résidus. Le principe est le suivant : quand un support est détecté, sa position dans la représentation binaire de la contraintes est enregistré. On utilise alors la matrice d'entiers à trois dimensions res (initialisée par une matrice de 0), on peut alors utiliser l'algorithme 19. Lorsqu'on cherche un support, la position résiduelle est testée en premier (ligne 3), et quand une autre position est trouvée, elle est enregistrée (ligne 7).

Pour illustrer l'importance de la combinaison des opérations bit-à-bit avec les résidus sur des grands domaines, nous montrons dans la table 2.6 les résultats obtenus sur des instances du problème Domino, introduit par [ZHANG ET YAP 2001] pour exhiber la sous-optimalité de AC-3. Sur les instances les plus difficiles, où les domaines contiennent plus de 3 000 valeurs, AC-3^{bit+rm} est environ 5 fois plus rapide que AC-3^{bit} et AC-3^{rm}, et 9 fois plus efficace que AC-2001.

2.1.6 En résumé

Dans cette section, nous avons introduit une description précise de l'exploitation d'opérations bit-à-bit pour améliorer l'algorithme de consistance d'arc de base AC-3. L'algorithme obtenu, AC3^{bit}, apparaît être environ deux fois plus efficace que AC-3^{rm}, lui même manifestement plus rapide (et plus simple à intégrer dans MAC) que l'algorithme optimal AC-2001. Nous avons également montré comment combiner les opérations bit-à-bit avec les résidus, qui apparaissent très utiles quand la taille des domaines devient importante (plus de 300 valeurs). Nous pensons que, pour résoudre des instances binaires, quand les contraintes sont données en extension ou peuvent être efficacement converties en extension, l'algorithme générique MAC équipé de AC-3^{bit} ou AC-3^{bit+rm} est l'approche la plus efficace. Comme pour MAC-3^{rm}, il n'y a aucune structure de données à maintenir, en particulier après un retour-arrière.

Finalement, notez que MAC-3^{bit+rm} était l'algorithme utilisé par les prouveurs Abscon 109 et CSP4J qui ont participé à la Seconde Compétition Internationale de Prouveurs CSP [VAN DONGEN *et al.* 2006A,]. Plus précisément, l'algorithme a été utilisé sur les instances binaires impliquant des

contraintes en extension et les contraintes en intention pouvant facilement être converties en contraintes en extension. Par exemple, toutes les contraintes des instances RLFAP peuvent être converties en moins de 0,5 secondes. Les bons résultats obtenus par les deux prouveurs Java, et tout particulièrement Abscon, pendant cette compétition, confirment indirectement les résultats présentés dans cette section.

2.2 Consistances aux bornes

Les consistances plus fortes que la consistance d'arc, comme la Max-consistance de chemin Restreinte (Max-RPC) ou la singleton consistance d'arc, semblent prometteuses, mais il paraît encore difficile à l'heure actuelle de maintenir ces consistances au cours de la recherche, et elles sont surtout utilisées pendant une phase de préparation, avant de commencer la recherche elle même, par exemple avec un algorithme de type MGAC. Une autre approche possible semble être de considérer une forme partielle de ces consistances fortes.

Dans le cadre des CSP continus, c'est à dire les CSP dont les variables peuvent prendre des valeurs réelles ($\mathrm{dom}(X) \subseteq \mathbb{R}$), ou tout du moins flottantes ($\mathrm{dom}(X) \subseteq \mathbb{F}$), il faut raisonner en termes d'intervalles continus. Les algorithmes d'inférence utilisés dans ce cadre sont adaptés de la consistance d'arc : la consistance d'arc simple pourrait entraîner des itérations infinies (ou presque) [HYVONEN 1992, FALTINGS 1994]. Les domaines sont considérés convexes (constitués d'un seul intervalle). En limitant la consistance d'arc aux bornes de chaque intervalle (convexe), on peut introduire de nouvelles formes de consistances. La consistance basée sur l'approximation (afin de maintenir les domaines convexes) de la projection de fonctions afin de réduire les domaines est appelées la 2B consistance [LHOMME 1993] ou la Hull consistance [BENHAMOU *et al.* 1994,]. Des consistances plus fortes que la 2B consistance ont également été introduites, et en particulier la 3B consistance, qui contrôle si en assignant les variables à chacune de leurs bornes, on obtient un réseau toujours 2B-consistant [LHOMME 1993]. La 3B consistance peut être mise en parallèle avec la singleton consistance d'arc.

Nous introduisons dans cette section plusieurs consistances limitées aux bornes des domaines. L'objectif est de déterminer en pratique si l'exploitation de plusieurs types de consistances aux bornes sur des CSP discrets, comme proposé par [CODOGNET ET DIAZ 1996, VAN HENTENRYCK *et al.* 1998,], se révèle efficace.

Les travaux présentés dans cette section ont été réalisés en coopération avec Christophe Lecoutre et ont fait l'objet d'un article publié lors du 2e Workshop on Constraint Propagation and Implementation (CPAI'05) [LECOUTRE ET VION 2005].

2.2.1 Consistances de domaine aux bornes

Pour toute variable X, $\min(X)$ et $\max(X)$ représentent respectivement la plus petite et la plus grande valeur de $\mathrm{dom}(X)$. Le domaine de chaque variable sera représenté par un intervalle : $\mathrm{dom}(X) = [\min(X), \max(X)]$. On peut définir une version limitée aux bornes de n'importe quelle consistance de domaine Φ comme suit :

Définition 20 (Consistance aux bornes)**.** Soit $P = (\mathscr{X}, \mathscr{C})$ un CN et Φ une consistance de domaine. P est Φ-consistant aux bornes ssi $\forall X \in \mathscr{X}$, $\mathrm{dom}(X) \neq \emptyset$ et $\min(X)$ et $\max(X)$ sont tous deux Φ-consistants.

La 2B et la 3B consistances ont été introduites dans le cadre des CSP continus [LHOMME 1993].

Définition 21 (2B, 3B consistances)**.** $P = (\mathscr{X}, \mathscr{C})$ est :
- 2B-consistant ssi $\forall X \in \mathscr{X}$, $\mathrm{dom}(X) \neq \emptyset$ et $\min(X)$ et $\max(X)$ sont tous deux arc-consistants.
- 3B-consistant ssi $\forall X \in \mathscr{X}$, $\mathrm{dom}(X) \neq \emptyset$, $\mathrm{2B}(P|_{X=\min(X)}) \neq \bot$ et $\mathrm{2B}(P|_{X=max(X)}) \neq \bot$.

Algorithme 20 : 2B ($P = (\mathscr{X}, \mathscr{C})$: CN, $\mathscr{Y}$: {Variable}) : CN

```
1 Q ← 𝒴
2 tant que Q ≠ ∅ faire
3     prendre X de Q
4     pour chaque C ∈ 𝒞 | X ∈ vars(C) faire
5         pour chaque Y ∈ vars(C) faire
6             si revise-2B(C, Y) alors
7                 si dom(Y) = ∅ alors retourner ⊥
8                 Q ← Q ∪ {Y}
9 retourner P
```

Algorithme 21 : revise-2B(C : Contrainte, X : Variable) : Booléen

```
1 domainSize ← |dom(X)|
2 tant que |dom(X)| > 0 ∧ ¬seekSupport(C,X_min(X)) faire
3     retirer min(X) de dom(X)
4 tant que |dom(X)| > 1 ∧ ¬seekSupport(C,X_max(X)) faire
5     retirer max(X) de dom(X)
6 retourner domainSize ≠ |dom(X)|
```

La 2B consistance correspond clairement à la consistance d'arc aux bornes et la 3B consistance à une relaxation de SAC aux bornes (SAC aux bornes nécessiterait que $\mathrm{AC}(P|_{X=\min(X)}) \neq \bot$ et $\mathrm{AC}(P|_{X=max(X)}) \neq \bot$). On peut observer (voir ci-après) qu'une consistance et sa restriction aux bornes admettent la même complexité dans le pire des cas : par exemple, établir AC ou 2B se fait en $O(ed^2)$ et établir SAC, SAC aux bornes ou même la 3B consistance sont en $O(end^3)$. Bien que cela semble contredire les complexités optimales obtenues pour la 2B consistance ($O(ed)$ d'après [LHOMME 1993]) et la 3B consistance ($O(end^2)$ d'après [BORDEAUX *et al.* 2001,]), ces travaux considéraient que les contraintes étaient « basiques », c'est à dire qu'il existait pour chaque contrainte C des fonctions capables de calculer en temps constant les bornes min et max de domaine de toute variable impliquée par C, ce qui explique le facteur d.

Cependant, un avantage lié à l'exploitation des consistances aux bornes reste la complexité spatiale très limitée : en effet, il est possible de réduire l'espace requis par certains algorithmes d'un facteur d ou même d^2, puisque toutes les structures de données peuvent être limitées au stockage de deux bornes. De plus, si on se limite à des domaines convexes (toutes les valeurs entre les bornes min et max appartiennent au domaine), un réseau de contraintes peut être représenté avec une complexité spatiale en $O(n + e)$. Ce peut être très utile pour des réseaux impliquant des variables aux grands domaines, comme par exemple les réseaux représentant des problèmes d'ordonnancement.

2.2.2 2B consistance (consistance d'arc aux bornes)

Pour adapter un algorithme de consistance d'arc de type AC-3 aux bornes, on peut simplement utiliser l'algorithme $revise$-$2B$ présenté par l'algorithme 21 en lieu et place du $revise$ classique (algorithme 2 page 21). Les propriétés qui permettaient d'éviter des révisions inutiles pour l'algorithme GAC-3 orienté variables ne sont cependant plus valables : si une révision d'un arc (C, X) est effectuée, il faut obligatoirement contrôler tous les arcs (C', X) (avec $C' \neq C$), puisque la consistance des nouvelles

bornes de $\mathrm{dom}(X)$ n'a jamais été contrôlée. L'algorithme permettant d'établir la 2B consistance est alors l'algorithme 20.

Les algorithmes présentés par [LHOMME 1993] et [BORDEAUX *et al.* 2001,] utilisent une fonction de révision spécifique à chaque contrainte, et supposent que pour toute contrainte, une telle fonction en $O(1)$ existe.

Proposition 2. Appliqué à un réseau de contraintes binaire, l'algorithme de 2B consistance (l'algorithme 1 page 21 utilisant la fonction de révision $revise\text{-}2B$) admet des complexités temporelle et spatiale dans le pire des cas en $O(ed^2)$ et $O(n)$, respectivement.

Démonstration. Chaque arc $(Contrainte, Variable)$ peut être révisé d fois dans le pire des cas [MACKWORTH 1977, BORDEAUX *et al.* 2001, BESSIÈRE *et al.* 2005,]. Si une révision ne supprime aucune valeur, au plus $2 \times d$ tests de contraintes sont effectués par $revise\text{-}2B$. Si des valeurs sont été supprimées des dommaines, il peut y avoir jusqu'à d tests de contraintes par valeur supprimée (dans le cas binaire). Cependant, pour chaque révision utile effectuée sur un arc donné, le nombre maximal de révisions possibles sur cet arc est réduit d'autant : seulement d révisions utiles peuvent être effectuées sur chaque arc. Comme il existe $2 \times e$ arcs dans le cas binaire, on obtient une complexité temporelle dans le pire des cas en $O(ed^2)$.

D'autre part, la seule structure utilisée par l'algorithme est la queue Q, dont la complexité spatiale est en $O(n)$. □

On peut remarquer que même si on exploite l'enregistrement des supports comme le ferait AC-2001, la complexité dans le pire des cas reste en $O(ed^2)$, bien qu'on aurait pu s'attendre à une meilleure complexité, les consistances aux bornes ne considérant que les min et le max de chaque domaine.

On peut constater que dans le cas de la 2B consistance, la complexité dans le pire des cas correspond directement à des situations où un grand nombre de valeurs sont supprimées. Appliqué à un réseau déjà 2B-consistant, un algorithme de 2B consistance admet une complexité dans le pire des cas en $O(ed)$ alors qu'un algorithme de type AC-3 ou AC-2001 appliqué à un réseau arc-consistant garde une complexité dans le pire des cas en $O(ed^2)$.

2.2.3 2B+ consistance (Max-consistance de chemin restreinte aux bornes)

La max-consistance de chemin restreinte (Max-RPC) [DEBRUYNE ET BESSIÈRE 1997A] est plus forte que la consistance d'arc, la consistance de chemin restreinte ou la k-consistance de chemin restreinte, mais plus faible que la singleton consistance d'arc. Dans cette section, nous proposons d'adapter Max-RPC en la limitant aux bornes, tout en utilisant un algorithme de consistance d'arc à gros grain sous-jacent. En fait, cette adaptation, notée 2B+, ne garantit pas que chaque borne ait un support chemin-consistant dans chaque contrainte. Il faut le considérer comme un algorithme « opportuniste », simple à définir et à implanter.

On obtient l'algorithme 2B+ en appelant la fonction $seekSupport\text{-}Path$ (algorithme 22) en lieu et place du $seekSupport$ classique dans $revise\text{-}2B$. Cette fonction renvoie **vrai** ssi la valeur donnée a un support chemin-consistant dans la contrainte donnée.

2.2.4 3B consistance (Singleton consistance d'arc aux bornes)

De nombreux travaux ont récemment étudié la singleton consistance d'arc (cf section 1.2.4 page 28). Bien qu'il soit possible de proposer un algorithme établissant SAC aux bornes, il semble plus approprié d'établir en fait une 3B consistance, afin de pouvoir continuer à représenter les domaines des variables sous forme d'intervalles. En effet, effectuer SAC aux bornes nécessiterait d'appliquer la consistance

Algorithme 22 : seekSupport-Path(C_{XY} : Contrainte, X_a : Variable$_{valeur}$) : Booléen

```
1  Soit Y la variable telle que vars(C_XY) = {X,Y}
2  pour chaque b ∈ dom(Y) | (X_a, Y_b) ∈ rel(C) faire
3      pour chaque Z t.q. (X, Y, Z) forme une 3-clique faire
4          pour chaque c ∈ dom(Z) faire
5              Soient C_XZ et C_YZ telles que vars(C_XZ) = {X, Z} et vars(C_YZ) = {Y, Z}
6              si (X_a, Z_c) ∈ C_XZ ∧ (Y_b, Z_c) ∈ C_YZ alors
7                  continuer boucle ligne 3
8          continuer boucle ligne 2
9      retourner vrai
10 retourner faux
```

Algorithme 23 : 3B-1($P = (\mathscr{X},\mathscr{C})$: CN) : CN

```
1  P ← 2B(P, 𝒳)
2  X ← first(𝒳)
3  marque ← X
4  répéter
5      domainSize ← |dom(X)|
6      tant que |dom(X)| > 0 ∧ 2B(P|_{X=min(X)}, {X}) ≠ ⊥ faire
7          retirer min(X) de dom(X)
8      tant que |dom(X)| > 1 ∧ 2B(P|_{X=max(X)}, {X}) ≠ ⊥ faire
9          retirer max(X) de dom(X)
10     si |dom(X)| < domainSize alors
11         si 2B(P, {X}) = ⊥ alors retourner ⊥
12         marque ← X
13     X ← next-modulo(X, 𝒳)
14 jusqu'à X = marque
15 retourner P
```

d'arc sur le réseau, susceptible de supprimer des valeurs à l'intérieur du domaine et pas seulement au niveau des bornes.

3B-1

L'algorithme 23, est l'adaptation aux bornes d'un algorithme de singleton consistance d'arc de base (type SAC-1). 3B-1 établit d'abord la 2B consistance sur le réseau donné (ligne 1). Puis, on teste si chaque borne du domaine de chaque variable est 2B-consistant (lignes 6 et 8). Les bornes non consistantes sont supprimées (lignes 7 et 9). Quand le domaine d'une variable est modifié, la 2B consistance est maintenue (lignes 10 et 11). Le processus est poursuivi jusqu'à l'obtention d'un point fixe : la terminaison est gérée de la même manière que pour un algorithme SAC classique.

Proposition 3. Appliqué à un réseau de contraintes binaires, l'algorithme 3B-1 admet des complexités temporelle et spatiale en $O(en^2d^3)$ et $O(n)$, respectivement.

Démonstration. Chaque variable du réseau peut être révisée au plus nd fois : dans le pire des cas, après avoir révisé toutes les variables, on a supprimé une seule valeur, nécessitant de réviser à nouveau toutes les variables du réseau. Dans ce cas, à chacun de ces « tours de variable », on fait appel $2 \times n + 1$ fois à l'algorithme de 2B consistance ($2 \times n$ pour contrôler les deux bornes de chaque variable, plus un après avoir supprimé une borne). Si on supprime plusieurs valeurs par tour, on réduit d'autant le nombre maximal de tours. On obtient donc une complexité temporelle dans le pire des cas en $O(en^2d^3)$. Comme l'algorithme 3B-1 ne nécessite pas de structure de données, sa complexité spatiale est identique à celle de 2B, c'est à dire $O(n)$. □

Encore une fois, la complexité de 3B-1 appliquée à un réseau déjà 3B-consistant est très limitée : un seul tour de boucle, dans lequel on effectue seulement $2 \times n$ appels à l'algorithme 2B en $O(ed^2)$: bien qu'un réseau 3B consistant soit aussi 2B consistant, on n'applique pas ici la 2B consistance sur des réseaux 2B consistants mais sur des réseaux $P|_{X=a}$ qui ne sont probablement pas 2B-consistants. On obtient donc une complexité en $O(end^2)$ seulement.

3B-1d

Les algorithmes travaillant sur des domaines continus ne peuvent se contenter de tester une valeur réelle (ou même flottante) à la fois. Il faut donc utiliser des fonctions de projection pour réaliser la 2B consistance. Pour réaliser la 3B consistance, on utilise en général plutôt une approche dichotomique, susceptible de détecter l'inconsistance d'un grand nombre de valeurs simultanément [LHOMME 1993]. Au lieu de tester la consistance de chaque valeur une par une, on teste d'abord la moitié inférieure de l'intervalle ($X = [\min(X), m]$, m étant la valeur médiane de l'intervalle). Si cette moitié est inconsistante, on peut la supprimer immédiatement. Sinon, on teste la moitié inférieure de cet intervalle, puis éventuellement la moitié supérieure si la moitié inférieure est inconsistante, et ce récursivement, jusqu'à ce que la borne inférieure soit validée, puis on procède de même pour la valeur supérieure. L'algorithme 3B-1d est une adaptation dichotomique de l'algorithme 3B-1 de base, et dispose d'une complexité temporelle dans le pire des cas de $O\left(en^2d^3\log(d)\right)$: on peut être amené à ne détecter des valeurs inconsistantes qu'une par une, et les décompositions dichotomiques sont dans ce cas inutiles.

Pour garder la bonne complexité $O(end^2)$ quand 3B-1d est appliqué à un réseau de contraintes déjà 3B-consistant, on commence par tester les bornes min et max individuellement avant de commencer la décomposition dichotomique s'il y a lieu. (Sans ce petit test supplémentaire, la complexité serait de $O\left(end^2\log(d)\right)$.)

3B-2

Il est également possible de chercher à améliorer l'algorithme 3B-1 dans l'esprit des travaux qui ont été effectuées sur SAC (SAC-OPT, SAC-SDS, SAC-3) et obtenir une complexité dans le pire des cas améliorée ($O(end^3)$, qui correspond à la complexité optimale pour établir SAC).

On obtient cette complexité en enregistrant et en exploitant un certain nombre d'informations au cours de l'établissement de la 3B consistance. L'algorithme 3B-2 est décrit par les algorithmes 24, 25, 26 et 27.

Quand la consistance d'une valeur doit être testée pour la deuxième fois, il est inutile d'établir la 2B consistance à partir du début [BORDEAUX *et al.* 2001, BESSIÈRE ET DEBRUYNE 2005,]. Nous introduisons donc trois structures de données :

- $initialized[X, bound]$ est un tableau indiquant pour chacune des bornes min ou max de X si la 2B consistance a déjà été établie.
- $minInferences[X, bound, Y]$ est un tableau permettant d'enregistrer pour chaque borne de X et chaque variable Y la valeur $\min(Y)$ dans $\mathrm{2B}(P|_{X=\min(X)})$ ou $\mathrm{2B}(P|_{X=\max(X)})$.

Algorithme 24 : 3B-2($P = (\mathscr{X},\mathscr{C})$: CN) : CN

```
1  P ← 2B(P, 𝒳)
2  X ← first(𝒳)
3  marque ← X
4  répéter
5      domainSize ← |dom(X)|
6      tant que |dom(X)| > 0 ∧ check2B(P, X, min) faire
7          retirer min(X) de dom(X)
8      tant que |dom(X)| > 1 ∧ check2B(P, X, max) faire
9          retirer max(X) de dom(X)
10     si |dom(X)| < domainSize alors
11         P' ← 2B(P, {X})
12         si P' = ⊥ alors retourner ⊥
13         foreach X ∈ 𝒳 do
14             si min^P(X) ≠ min^P'(X) alors initialized[X, min] ← faux
15             si max^P(X) ≠ max^P'(X) alors initialized[X, max] ← faux
16         P ← P'
17         marque ← X
18     X ← next-modulo(X, 𝒳)
19 jusqu'à X = marque
20 retourner P
```

Algorithme 25 : check2B($P = (\mathscr{X},\mathscr{C})$: CN, X : Variable, $bound$: $min|max$) : Booléen

```
1  P_store ← P
2  S ← exploitInferences(X, bound)
3  si S = ∅ alors
4      consistent ← vrai
5  sinon
6      si bound = min alors
7          consistent ← 2B(P|_{X=min(X)}, S) ≠ ⊥
8      sinon
9          consistent ← 2B(P|_{X=max(X)}, S) ≠ ⊥
10     si consistent alors
11         recordInferences(X, bound)
12     sinon
13         initialized[X, bound] ← faux
14 P ← P_store
15 retourner consistent
```

Algorithme 26 : exploitInferences(X : Variable, $bound$: $min|max$) : {Variable}

```
1 si ¬initialized[X, bound] alors retourner {X}
2 S ← ∅
3 pour chaque Y ∈ 𝒳 faire
4     min(Y) ← max(min(Y), minInferences[X, bound, Y])
5     max(Y) ← min(max(Y), maxInferences[X, bound, Y])
6     si min(Y) > minInferences[X_a, Y] ∨ max(Y) < maxInferences[X_a, Y] alors
7         S ← S ∪ {Y}
8 retourner S
```

Algorithme 27 : recordInferences(X : Variable, $bound$: $min|max$)

```
1 initialized[X, bound] ← vrai
2 pour chaque Y ∈ 𝒳 faire
3     minInferences[X, bound, Y] ← min(Y)
4     maxInferences[X, bound, Y] ← max(Y)
```

– $maxInferences[X, bound, Y]$ permet d'obtenir pour chaque valeur borne de X et chaque variable Y la valeur $\max(Y)$ dans $2\mathrm{B}(P|_{X=\min(X)})$ ou $2\mathrm{B}(P|_{X=\max(X)})$.

On suppose que $initialized$ est un tableau dont les éléments sont initialisés à **faux**. Les inférences pour une borne X ne sont possibles que si $initialized[X, bound]$ vaut **vrai**. Par exemple, on peut imaginer qu'après avoir établi $2\mathrm{B}(P|_{X=\min(X)})$, on obtienne un réseau tel que $\min(Y) = c$ et $\max(Y) = d$ (et donc, $\mathrm{dom}(Y) = \{c, \ldots, d\}$). $initialized[X, min]$ est alors mis à **vrai**, $minInferences[X, min, Y]$ à c et $maxInferences[X, min, Y]$ à d. En exécutant $check2B$ (algorithme 25), les informations enregistrées sont exploitées (ligne 2) par un appel à $exploitInferences$. Après l'exploitation des valeurs enregistrées, soit la 2B consistance de la borne considérée de X est toujours valide (ligne 4), puisque l'ensemble vide $\emptyset$ est renvoyé par $exploitInferences$ (aucune valeur dans $2\mathrm{B}(P|_{X=\min|\max(X)})$ n'a été supprimée), soit il faut contrôler la 2B consistance de la borne considérée de X à partir de l'ensemble de variables S dont le domaine a été réduit (lignes 6 à 9). Les inférences sont alors mises à jour (ligne 11) par un appel à la fonction $recordInferences$. Si la borne est inconsistante, elle va être modifiée, et les valeurs courantes de $minInferences$ et $maxInferences$ ne sont plus valables. Elles devront donc être recalculées, et $initialized[X, bound]$ est réinitialisé. De même dans l'algorithme 24, si des variables sont modifiées quand la 2B consistance est maintenue (lignes 13-15). $\min^P(X)$ et $\min^{P'}(X)$ représentent la valeur minimale de X dans les réseaux P et P', respectivement (et idem pour $\max(X)$).

La fonction $exploitInferences$ (algorithme 26) retourne soit le singleton $\{X\}$ (ligne 2) si la 2B consistance de la borne considérée de X n'a jamais été contrôlée, ou l'ensemble de variables dont le domaine a perdu au moins une valeur en plus de la dernière exécution de 2B sur cette borne (ligne 8). La fonction $recordInferences$ (algorithme 27) se contente de mettre à jour les structures de données.

L'algorithme décrit ici peut être vu comme une adaptation aux bornes de l'algorithme SAC-OPT proposé par [BESSIÈRE ET DEBRUYNE 2005] ou encore une alternative à l'algorithme de 3B consistance optimisé décrit dans [BORDEAUX *et al.* 2001,].

Proposition 4. Appliqué à un réseau de contraintes binaires, l'algorithme 3B-2 admet des complexités temporelle et spatiale dans le pire des cas en $O(end^3)$ et $O(n^2)$, respectivement.

Démonstration. En enregistrant des informations et en évitant ainsi d'effectuer plusieurs fois le même calcul inutilement [BORDEAUX *et al.* 2001,], l'algorithme 3B-2 exploite l'incrémentalité de la consis-

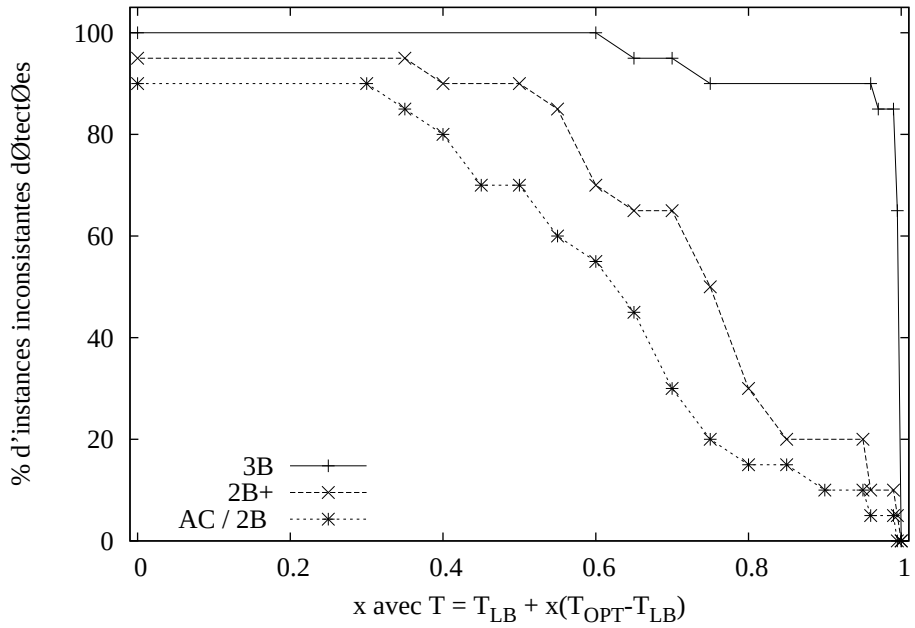


FIGURE 2.2 – Proportion d'instances incohérentes prouvées sans recherche

tance d'arc [BESSIÈRE ET DEBRUYNE 2005]. Cela signifie que $2 \times n$ appels successifs potentiels à $check2B$ par rapport à une valeur X_a est en $O(ed^2)$. Les fonctions $exploitInferences$ et $recordInferences$ sont en $O(n)$ et peuvent être ignorées. Comme il y a nd valeurs, la complexité temporelle dans le pire des cas est en $O(end^3)$. Les structures de données ne stockent que deux bornes par variable. On obtient alors $2 \times n$ valeurs à enregistrer pour $2 \times n$ bornes. On obtient donc une complexité spatiale en $O(n^2)$. □

2.2.5 Expérimentations

Pour montrer l'intérêt pratique des propriétés introduites dans cette section, nous avons implanté les différents algorithmes décrits dans les sections précédentes et conduit une expérimentation sur des instances d'ordonnancement (Job Shop et Open Shop) et d'assignation de fréquences (RLFAP). Les algorithmes 2B, 2B+, 3B-1, 3B-1d et 3B-2 correspondent aux descriptions données dans les paragraphes précédents.

Nous avons tout d'abord établi une comparaison entre tous les algorithmes sur un ensemble de 20 problèmes de Job Shop générés à partir du modèle de Taillard [TAILLARD 1993], en fixant la taille de ceux-ci à 8 jobs et 8 machines. Pour chaque instance, il existe un temps optimal T_{OPT} minimal dans lequel on peut réaliser tous les jobs en respectant l'utilisation des ressources. Calculer cette valeur optimale nécessite plusieurs appels à un algorithme NP-Complet. Une estimation basse $T_{LB} \leq T_{OPT}$ en additionnant les temps de chaque opération pour chaque job un par un d'une part, et les temps de chaque opération pour une même machine d'autre part. La limite basse est la valeur maximale de ces sommes. Pour chacune des 20 instances générées ici, on a calculé la valeur T_{OPT} grâce à un algorithme MGAC classique.

Nous avons alors cherché à estimer le pouvoir filtrant de chaque algorithme en fixant une limite T comprise entre T_{LB} et T_{OPT}. Quand T est proche de T_{LB}, certains algorithmes sont capables de détecter l'inconsistance du problème sans même effectuer de recherche (évidemment, tous les problèmes avec

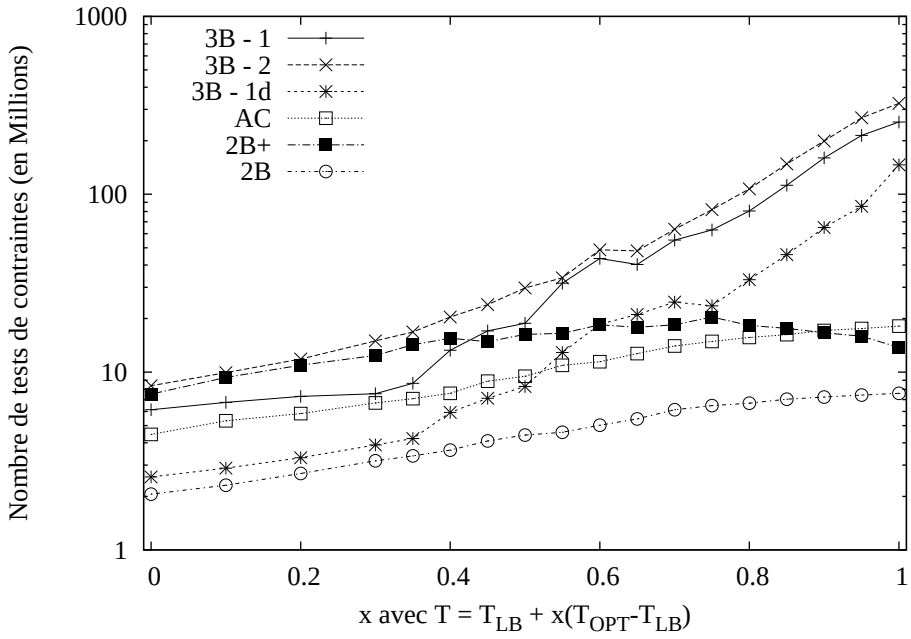


FIGURE 2.3 – Tests de contraintes effectués pour établir les consistances

$T < T_{OPT}$ sont inconsistants). La figure 2.2 montre la proportion d'instances détectées inconsistances sans recherche en faisant varier T entre T_{LB} et T_{OPT} par la formule $T = T_{LB} + x(T_{OPT} - T_{LB})$ et en faisant varier x entre 0 ($T = T_{LB}$) et 1 ($T = T_{OPT}$). On peut observer que la 3B consistance permet un niveau de filtrage suffisant pour détecter l'inconsistance de la plupart de ces instances, contrairement à 2B+ et tout particulièrement 2B et AC, qui ont un comportement identique. La figure 2.3 montre l'effort, en terme de tests de contraintes (échelle logarithmique), nécessaire pour l'établir les différentes consistances avec les algorithmes présentés (des résultats similaires sont obtenus en considérant le temps CPU). Naturellement, plus la capacité de filtrage est importante, plus il faut de tests de contraintes pour l'établir. On peut constater que 3B-1d semble être le plus efficace sur ces instances.

Nous avons ensuite considéré les instances RLFAP. La table 2.7 montre les résultats obtenus sur quelques instances sélectionnées. Il apparaît ici que AC reste la meilleure approche sur ces instances, tant en temps CPU qu'en termes de filtrage (#rmvs indique le nombre de valeurs inconsistantes supprimées). Ceci peut s'expliquer de par le fait que contrairement aux problèmes d'ordonnancement qui utilisent beaucoup de contraintes de précédence, les contraintes apparaissant dans les problèmes RLFAP ne se concentrent pas sur les bornes des domaines.

Dans un second temps, nous avons cherché à maintenir toutes les consistances pendant la recherche d'une solution. Nous avons étudié tout d'abord les 10 instances d'Open Shop de Taillard avec 7 Jobs et 7 Machines [TAILLARD 1993]. Pour chaque instance, nous avons essayé de calculer T_{OPT} en utilisant un algorithme de branchement exploitant la propagation de contraintes. La table 2.8 donne la moyenne et l'écart-type sur les 10 instances de l'écart entre la borne supérieure obtenue en 300 secondes de calcul avec les différents algorithmes de propagation par rapport à la valeur optimale pour le problème. Les résultats indiquent clairement que si la 2B consistance n'est pas une approche très performante, la 3B consistance, en particulier avec les algorithmes 3B-1d et 3B-2, est particulièrement efficace sur ces problèmes.

Nous avons également considéré à nouveau les 20 instances de Job Shop décrites précédemment. La table 2.9 présente le temps moyen nécessaire pour prouver que le problème est consistant pour $T =$

		AC	2B	2B+	3B-1	3B-1d	3B-2
$graph4$	cpu	0,47	0,10	2,23	8,29	10,25	34,26
	#rmvs	776	0	187	411	411	411
$graph10$	cpu	0,86	0,15	4,15	11,87	13,42	32,75
	#rmvs	386	0	46	122	122	122
$graph14$-$f27$	cpu	0,43	0,16	1,93	3,25	2,48	3,32
	#rmvs	2 314	0	0	0	0	0
$graph14$-$f28$	cpu	0,43	0,16	2,10	5,81	4,72	4,56
	#rmvs	3 230	0	0	2	2	2
$scen02$-$f25$	cpu	0,14	0,07	0,55	0,58	0,52	0,58
	#rmvs	106	0	0	0	0	0
$scen11$-$f8$	cpu	0,55	0,13	2,70	3,27	2,95	3,33
	#rmvs	4 992	0	0	0	0	0
$scen11$-$f10$	cpu	0,51	0,21	4,11	2,98	2,66	3,12
	#rmvs	6 324	3 024	3 024	3 024	3 024	3 024

TABLE 2.7 – Établir les consistances sur des instances RLFAP

	AC	2B	3B - 1	3B - 1d	3B - 2
Écart moyen	4,79%	28,6%	4,28%	3,00%	3,32%
Écart-type	3,5%	8,2%	2,4%	2,2%	2,4%

TABLE 2.8 – Maintenir les consistances sur les instances d'Open Shop 7×7 de Taillard

	AC	2B	2B+	3B - 1	3B - 1d	3B - 2
Temps CPU Moyen ($T = T_{OPT}$)	212	216	282	88	73	117
Inconsistances prouvées en 300s ($T = T_{OPT} - 1$)	45%	55%	30%	85%	85%	85%

TABLE 2.9 – Maintenir les consistances sur des instances de Job Shop 8×8

	AC	2B	2B+	3B - 1	3B - 1d	3B - 2
$graph04$	2,1	*timeout*	*timeout*	260,0	314,0	391,0
$graph10$	8,2	*timeout*	14,3	*timeout*	*timeout*	*timeout*
$graph14$-$f27$	6,3	*timeout*	*timeout*	*timeout*	847,0	*timeout*
$graph14$-$f28$	40,7	8,1	20,5	347,6	435,9	*timeout*
$scen02$-$f25$	4,1	2,9	43,5	47,8	43,6	50,0
$scen11$-$f8$	115,3	74,6	98,9	*timeout*	*timeout*	*timeout*
$scen11$-$f10$	6,7	6,4	15,6	388,3	372,7	652,3

TABLE 2.10 – Maintenir les consistances sur des instances de RLFAP

T_{OPT} ainsi que la proportion de problèmes pour lesquels on a pu prouver qu'ils étaient inconsistants en moins de 300 secondes pour $T = T_{OPT} - 1$. Une fois encore, maintenir 3B est la meilleure approche.

La table 2.10 montre les résultats ($timeout$ a été fixé à 900 secondes) obtenus pour résoudre les instances RLFAP. Sur certaines instances difficiles, 2B donne de bons résultats, mais maintenir des consistances plus fortes est plutôt pénalisant.

Finalement, n'oublions pas que tous les algorithmes sont basés sur AC-3. L'utilisation d'algorithmes de base plus sophistiqués comme AC-2001 ou AC-3^{rm} pour établir la 2B ou la 3B consistance ne devrait pas changer les résultats : on a vu précédemment que la complexité pour établir la 2B consistance restait en $O(ed^2)$, que l'algorithme soit basé sur AC-3 ou AC-2001. Cependant, l'algorithme 2B+ pourrait être amélioré dans ce sens : beaucoup de tests de contraintes pourraient être évités.

2.2.6 En résumé

L'objectif des travaux décrits dans cette section était d'établir une connexion plus claire entre les consistances de domaines établies sur le modèle des CSP discrets, et les consistances aux bornes établies sur le modèle des CSP continus. En particulier, nous avons étudié des versions limitées aux bornes de consistances de domaine bien connues.

Le principal avantage des consistances aux bornes réside dans les besoins en espace qui peuvent être très limités, en particulier quand les domaines sont convexes. Pour certaines instances de CSP discrets avec de très grands domaines, ce peut être la seule approche réaliste. D'autre part, si l'économie d'espace n'est pas indispensable, la complexité dans le pire des cas pour établir les consistances aux bornes (pour lesquelles aucune sémantique des contraintes n'est disponible) sont assez décevantes, bien que sur des instances peu contraintes beaucoup de travail est évité. Par exemple, dans ce contexte, la complexité optimale dans le pire des cas pour établir la 2B consistance (c'est à dire la consistance d'arc aux bornes) est identique à celle pour établir la consistance d'arc, bien que la complexité des algorithmes de 2B consistance appliqués à des réseaux déjà 2B-consistants est meilleure ($O(ed)$ contre $O(ed^2)$). D'un point de vue pratique, l'utilisation de consistances aux bornes est une bonne approche sur des problèmes impliquant des contraintes « travaillant aux les bornes », comme les contraintes de précédence. Dans ce cas, il est de plus souvent possible d'adopter un filtrage spécifique qui exploite la sémantique des contraintes et ainsi d'obtenir une meilleure complexité. Dans un contexte moins favorable, nos résultats expérimentaux sur les problèmes d'assignation de fréquences ne montre pas un réel avantage à l'utilisation de consistances aux bornes par rapport à la consistance d'arc classique. Nous pensons cependant que les consistances aux bornes pourraient jouer un rôle dans le développement de méthodes destinées à contrôler l'effort nécessaire pour maintenir des consistances fortes pendant la recherche.

2.3 Consistance duale et consistance de chemin

Si les consistances de domaine [DEBRUYNE ET BESSIÈRE 2001] permettent d'identifier des valeurs inconsistances, les consistances de relations permettent d'identifier des instanciations inconsistantes (de longueur 2 quand les relations sont binaires). La consistance de relations la plus connue est la consistance de chemin (PC). De nombreux algorithmes ont été proposés pour établir PC (cf section 1.2.2).

Nous définissons ici une nouvelle forme de consistance de relations, nommée la consistance duale. Le principe est d'enregistrer des couples de valeurs inconsistantes identifiées après l'affectation d'une valeur à une variable puis l'établissement de la consistance d'arc. Comme pour la singleton consistance d'arc, la consistance duale est donc construite autour de la consistance d'arc. Nous montrons d'après l'algorithme de McGregor [MCGREGOR 1979] que la consistance duale est équivalente à la consistance de chemin quand elle est appliquée à un réseau de contraintes binaires complet.

Nous introduisons également plusieurs algorithmes permettant de réaliser une consistance de chemin forte en utilisant le principe introduit par la consistance duale, c'est à dire en effectuant des tests de singletons successifs. sDC-1 est l'algorithme le plus simple, et le deux autres, sDC-2 et sDC-3, améliorent sDC-1 en exploitant partiellement ou totalement l'incrémentalité de l'algorithme de consistance d'arc. En termes de complexité, sDC-1 admet une complexité temporelle dans le pire des cas [5] en $O(\lambda n^3 d^3) \in O(n^5 d^5)$ et une complexité spatiale en $O(n^2 d^2)$. sDC-2 améliore l'algorithme sDC-1 en pratique mais admet les même complexités temporelle et spatiale. En exploitant totalement l'incrémentalité d'AC, l'algorithme sDC-3 atteint une complexité temporelle dans le pire des cas en $O(n^3 d^4)$ tout en conservant la complexité spatiale en $O(n^2 d^2)$. sDC-3 a donc la même complexité que l'algorithme PC-8, qui est considéré comme l'algorithme le plus rapide pour établir la consistance de chemin [CHMEISS ET JÉGOU 1998].

Les travaux présentés dans cette section ont été réalisés en coopération avec Stéphane Cardon et Christophe Lecoutre et ont fait l'objet d'un article publié lors de la 13e Conférence on Principles and Practice of Constraint Programming (CP'2007) [LECOUTRE *et al.* 2007B,].

2.3.1 Consistance duale : définition et équivalence avec la consistance de chemin

La consistance duale, tout comme la consistance de chemin, n'est définie que dans le cadre d'un réseau de contraintes binaires normalisé.

Définition 22 (Consistance duale)**.** Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, (X, Y) deux variables distinctes de $\mathscr{X}$, $a \in \mathrm{dom}(X)$ et $b \in \mathrm{dom}(Y)$.

L'instanciation (de longueur 2) $\{X_a, Y_b\}$ est dual-consistante (DC) ssi $Y_b \in \mathrm{AC}(P|_{X=a})$ et $X_a \in \mathrm{AC}(P|_{Y=b})$.

(X, Y) est DC ssi toutes les instanciations de $\{X, Y\}$ sont DC.

P est DC ssi $\forall (X, Y) \in \mathscr{X}^2 \mid X \neq Y$, (X, Y) est DC.

On peut montrer que DC est équivalent à PC, ce que l'on pouvait prévoir puisque l'algorithme proposé par [MCGREGOR 1979] pour établir la consistance de chemin forte était basé sur la consistance d'arc. Nous en fournissons une preuve dans notre contexte :

Proposition 5. $\mathrm{DC} = \mathrm{PC}$

Démonstration. Montrons que pour tout CN P, $\mathrm{DC}(P) = \mathrm{PC}(P)$. Soit $\{X_a, Y_b\}$ une instanciation de P.

5. λ représente le nombre d'instanciations consistantes dans les contraintes du problème et est en $O(ed^2)$ dans le cas des réseaux binaires et $O(n^2 d^2)$ si le graphe est complet

Algorithme 28 : FC ($P = (\mathscr{X},\mathscr{C})$: CN, X : Variable) : CN

```
pour chaque C ∈ 𝒞 | X ∈ vars(C) faire
    soit Y la variable telle que vars(C) = {X, Y}
    si revise(C, Y) ∧ dom(Y) = ∅ alors retourner ⊥
retourner P
```

– Si $\{X_a, Y_b\}$ n'est pas PC, cela signifie que soit $\{X_a, Y_b\}$ est inconsistante et donc que $\{X_a, Y_b\}$ n'est pas DC, soit $\exists Z \in \mathscr{X} \mid \forall c \in \mathrm{dom}(Z)$, $\{X_a, Z_c\}$ ou (Y_b, Z_c) sont inconsistantes. Dans ce cas, on sait que $Y_b \notin \mathrm{AC}(P|_{X=a})$ puisque après avoir établi AC sur $P|_{X=a}$, toutes les valeurs c de $\mathrm{dom}(Z)$ sont telles que $\{X_a, Z_c\}$ est consistante. Nécessairement et par hypothèse, toutes ces valeurs sont incompatibles avec Y_b, ce qui entraîne la suppression de b de $\mathrm{dom}(Y)$ lors de l'établissement de AC. $\{X_a, Y_b\}$ n'est alors pas DC.
– Si $\{X_a, Y_b\}$ n'est pas DC, cela signifie que $Y_b \notin \mathrm{AC}(P|_{X=a})$ (ou $X_a \notin \mathrm{AC}(P|_{Y=b})$). Soit $H(n)$ l'hypothèse de récurrence suivante : si le nombre de révisions (c'est à dire le nombre d'étapes dans un algorithme tel que AC-3) pour retirer b de $\mathrm{dom}(Y)$ lors de l'établissement de AC sur $P|_{X=a}$ est inférieure ou égale à n, alors $\{X_a, Y_b\}$ est inconsistante dans $\mathrm{PC}(P)$.
$H(1)$ est valide, puisque dans ce cas, cela signifie simplement que $\{X_a, Y_b\}$ est inconsistante. Supposons que $H(n)$ est vraie et montrons que $H(n+1)$ est valide :
Si b est supprimée de $\mathrm{dom}(Y)$ après $n+1$ révisions en établissant AC sur $P|_{X=a}$, alors cela signifie que la dernière révision portait sur une contrainte impliquant Y et une autre variable Z. Toute valeur c de $\mathrm{dom}(Z)$ qui supportait Y_b avait été supprimée après au plus n révisions. Par hypothèse, cela signifie que pour toutes ces valeurs c, $\{X_a, Z_c\}$ n'est pas consistante dans $\mathrm{PC}(P)$. On peut donc déduire ici que $\{X_a, Y_b\}$ n'est pas PC.

□

Pour mémoire, un réseau est fortement chemin-consistant (sPC) ssi il est à la fois chemin-consistant et arc-consistant. De même, un réseau fortement dual-consistant (sDC) est à la fois dual-consistant et arc-consistant.

2.3.2 De nouveaux algorithmes pour établir la consistance de chemin forte

Les algorithmes que nous proposons pour établir sPC sont appelés sDC-1, sDC-2 et sDC-3. Avant de les décrire, il nous faut introduire quelques notions sur le Forward Checking (FC). Le principe fondamental de tous ces algorithmes est d'effectuer des tests singleton successifs jusqu'à atteindre un point fixe.

Consistance d'arc et Forward Checking

Les algorithmes de sDC proposés pour établir la sPC sont donnés dans le cadre de l'utilisation d'un algorithme de consistance d'arc à gros grain (AC-3, AC-2001, AC-3^{rm}, AC-3^{bit}...) orientés variables.

FC est un algorithme de recherche systématique, comme MAC, qui maintient une forme de consistance d'arc limitée à chaque nœud [HARALICK ET ELLIOTT 1980]. Après avoir assigné une valeur à une variable, seules les variables non assignées qui sont connectées à celle-ci sont révisées (voir algorithme 28 : l'algorithme FC ici présenté n'effectue que la forme limitée d'inférence utilisée par l'algorithme de recherche FC). Notons que la complexité dans le pire des cas d'un appel à FC est $O(nd)$, puisqu'il peut y avoir au plus $n-1$ révisions et la révision d'une variable par rapport à une variable assignée est en $O(d)$.

Algorithme 29 : sDC-1($P = (\mathscr{X},\mathscr{C})$: CN) : CN

```
1  P ← AC(P, 𝒳)
2  X ← first(𝒳)
3  marque ← X
4  répéter
5      si |dom(X)| > 1 alors
6          si checkVar-1(P, X) alors
7              P ← AC(P, {X})
8              si P = ⊥ alors retourner ⊥
9              marque ← X
10     X ← next-modulo(𝒳, X)
11 jusqu'à X = marque
12 retourner P
```

Algorithme 30 : checkVar-1($P = (\mathscr{X},\mathscr{C})$: CN, X : Variable) : Booléen

```
1  modif ← faux
2  pour chaque a ∈ dom^P(X) faire
3      P' ← AC(P|_{X=a}, {X})
4      si P' = ⊥ alors
5          retirer a de dom^P(X)
6          modif ← vrai
7      sinon
8          pour chaque C ∈ 𝒞 | X ∈ vars(C) faire
9              soit Y ∈ 𝒳 | vars(C) = {X, Y}
10             pour chaque b ∈ dom^P(Y) | b ∉ dom^{P'}(Y) faire
11                 retirer {X_a, Y_b} de rel^P(C)
12                 modif ← vrai
13 retourner modif
```

Algorithme sDC-1

Les algorithmes 29 et 30 établissent la consistance duale forte (et donc la consistance de chemin forte). L'algorithme est très proche de celui proposé dans [McGregor 1979] pour établir la consistance de chemin forte. Il y a cependant d'importantes améliorations qui seront décrites à la fin de cette sous-section. Dans l'algorithme 29, on commence par établir la consistance d'arc (ligne 1), puis, à chaque itération de la boucle principale, on considère une variable que l'on appellera la « variable courante ». On gère la terminaison de l'algorithme de la même manière que pour un algorithme de SAC classique. L'appel à $checkVar$-1 à la ligne 6 permet d'effectuer toutes les inférences possibles pour X comme décrit ci-dessous. Si une inférence est effectuée, $checkVar$-1 renvoie **vrai** et la consistance d'arc est ré-établie (ligne 7). On peut remarquer que quand le domaine d'une variable est constitué d'un singleton (une et une seule valeur), aucune inférence ne peut plus être faite, puisque le réseau est toujours maintenu arc-consistant (d'où le test à la ligne 5).

Pour effectuer toutes les inférences à partir d'une variable X, on fait appel à la fonction $checkVar$-1

(algorithme 30). Pour chaque valeur a du domaine de X, AC est établie sur $P|_{X=a}$. Si a n'est pas SAC, alors a est supprimée du domaine de X (ligne 5). Sinon (lignes 8 à 12), pour toute valeur Y_b présente dans P et absente de P', l'instanciation $\{X_a, Y_b\}$ est retirée de $\mathrm{rel}(C)$.

Notre algorithme peut être considéré comme une amélioration de l'algorithme de Mac Gregor [MC-GREGOR 1979]. Nous apportons deux améliorations importantes : le mécanisme de terminaison amélioré d'une part, et le fait de maintenir AC pendant l'exécution afin de démarrer les tests de singleton à partir d'une seule variable, ce qui permet d'éviter un grand nombre de révisions inutiles, notamment sur des réseaux de contraintes épars. Pour finir, cet algorithme est capable d'établir directement la consistance duale conservative, qui sera étudiée dans la section suivante (2.4).

Proposition 6. Appliqué à un graphe de contraintes complet, l'algorithme sDC-1 établit sDC.

Démonstration. Tout d'abord, la preuve que toute inférence effectuée par sDC-1 est correcte est immédiate. À la ligne 3 de l'algorithme 30, l'exécution de $P' \leftarrow \mathrm{AC}(P|_{X=a}, \{X\})$ donne un résultat équivalent à $P' \leftarrow \mathrm{AC}(P|_{X=a}, \mathscr{X})$, ce qui garantit que l'algorithme est complet. En effet, le réseau est maintenu AC à chaque fois qu'une modification est apportée (ligne 6 de l'algorithme 29) et toute inférence effectuée sur une valeur X_a n'a pas d'impact sur $P|_{X=b}$, b étant n'importe quelle autre valeur du domaine de la variable X. □

Proposition 7. La complexité dans le pire des cas de sDC-1 est en $O(\lambda end^3)$ et sa complexité spatiale est en $O(ed^2)$.

Démonstration. Dans le pire des cas, la fonction $checkVar$-1 peut être appelée λ fois pour une variable donnée, puisque entre deux appels successifs, au moins un couple de valeurs doit être supprimé d'une relation. Un algorithme d'AC optimal en $O(ed^2)$ comme AC-2001 peut être utilisé en ligne 3, et supprimer les couples de valeurs inconsistants (lignes 8 à 12) est en $O(nd)$. Comme on suppose que $n < e$, un appel à $checkVar$-1 est donc en $O(d(ed^2 + nd)) \in O(ed^3)$. On obtient donc $O(\lambda end^3)$ pour sDC-1.

En termes d'espace, il faut tout d'abord enregistrer les domaines et les relations ($O(nd + ed^2) \in O(ed^2)$). Les seules structures de données utilisées par sDC-1 sont celles utilisées par l'algorithme d'AC sous-jacent, soit $O(ed)$ pour AC-2001 ou AC-3^{rm}. La complexité spatiale est donc en $O(ed^2)$. □

Dans le cas d'un graphe complet, $e = (n^2 - n)/2 \in O(n^2)$, d'où des complexités temporelle en $O(\lambda n^3 d^3)$ et spatiale en $O(n^2 d^2)$. $\lambda \in O(ed^2)$, donc sDC-1 peut atteindre une complexité temporelle en $O(n^5 d^5)$. Cela peut sembler élevé, mais en pratique sDC-1 atteint très vite un point fixe (le nombre d'appels à la fonction $checkVar$-1 pour une variable donnée est très faible) parce que les inférences sur des valeurs inconsistances, et particulièrement les couples de valeurs inconsistantes, peuvent être immédiatement prises en compte. Le corollaire suivant indique également que le temps perdu pour appliquer sDC-1 sur un réseau déjà sDC est plutôt limité pour autant que la taille des domaines ne soit pas trop importante :

Corollaire 1. Appliqué à un réseau sDC, la complexité dans le pire des cas pour sDC-1 est en $O(end^3)$

Le meilleur des cas (extrême), survient quand tous les couples de valeurs sont autorisés. Dans ce cas, le coût pour établir AC à partir d'une seule variable (ligne 3) est identique à celui pour établir FC. La complexité dans le meilleur des cas de sDC-1 est donc seulement en $O(ed^2)$ ($\in O(n^2 d^2)$ pour un graphe complet).

Algorithme sDC-2

L'algorithme sDC-2 améliore l'algorithme sDC-1 en se basant sur l'idée de limiter le coût de l'établissement de AC à chaque test de singleton. Dans sDC-1, on applique $\mathrm{AC}(P|_{X=a}, \{X\})$. Cette opération est strictement identique à $\mathrm{AC}(\mathrm{FC}(P|_{X=a}, X), Q)$, où Q indique l'ensemble de variables de P dont

Algorithme 31 : sDC-2($P = (\mathscr{X}, \mathscr{C})$: CN) : CN

```
1  P ← AC(P, 𝒳)
2  X ← first(𝒳)
3  marque ← X
4  cnt ← 0
5  répéter
6      cnt ← cnt + 1
7      si |dom(X)| > 1 alors
8          nbValuesBefore ← nbValues(P)
9          si checkVar-2(P, X, cnt) alors
10             lastModif[X] ← cnt
11             P ← AC(P, {X})
12             si P = ⊥ alors retourner ⊥
13             si nbValues(P) ≠ nbValuesBefore alors
14                 lastModif[Y] ← cnt, ∀Y ∈ 𝒳
15             marque ← X
16     X ← next-modulo(𝒳, X)
17 jusqu'à X = marque
18 retourner P
```

le domaine a été réduit par $\mathrm{FC}(P|_{X=a}, X)$. En effet, en appliquant $\mathrm{AC}(P|_{X=a}, \{X\})$, on commence par réviser chaque variable $Y \neq X$ par rapport à X, et on ajoute ces variables dans la queue de propagation si une ou plusieurs valeurs on été supprimées de leurs domaines respectifs. La première itération de l'algorithme d'AC est équivalente à un FC. En dehors du premier test de singleton de X_a, dans sDC-2, on appliquera $\mathrm{AC}(\mathrm{FC}(P|_{X=a}, X), Q')$, où Q' est un ensemble de variables construit à partir d'informations enregistrées lors de la propagation. L'objectif est d'obtenir une file telle que $Q' \subseteq Q$, de sorte que sDC-2 soit moins coûteux que sDC-1 (un grand nombre de révisions inutiles peuvent être évitées, et, comme nous le verrons, le coût de gestion des informations nécessaires est négligeable). On exploite ainsi en partie l'incrémentalité de l'algorithme d'AC sous-jacent.

Pour établir sPC sur un réseau donné P, on peut utiliser notre second algorithme, sDC-2 (algorithme 31). Cette algorithme diffère de sDC-1 de par l'introduction d'un compteur et une structure de données nommée $lastModif$, constituée d'un tableau d'entiers. Le compteur est utilisé pour dénombrer le nombre de tours de la boucle principale (lignes 4 et 6). L'utilisation de $lastModif$ est définie comme suit : pour chaque variable X, $lastModif[X]$ indique le numéro du dernier tour où une inférence a été faite sur X. Une telle inférence peut consister en la suppression d'une valeur de $\mathrm{dom}(X)$ ou d'un couple de valeurs dans la relation associée à une contrainte impliquant X. Quand la fonction $checkVar$-2 renvoie **vrai**, cela signifie qu'au moins une inférence sur X a été effectuée, et $lastModif[X]$ est actualisée (ligne 10). On maintient ensuite AC (ligne 11), et si au moins une valeur a été supprimée (au cours de $checkVar$-2 ou de l'établissement de AC), on considère que toutes les variables ont été altérées pendant le tour actuel. On peut bien sûr chercher à ajuster $lastModif$ de manière plus subtile, en cherchant quelles variables sont réellement concernées par les inférences faites en maintenant AC, mais de par notre expérience, cela alourdit l'algorithme sans bénéfice notable.

Toutes les inférences applicables sur une variable donnée X sont effectuées par un appel à la fonction $checkVar$-2 (algorithme 32). Pour chaque valeur $a \in \mathrm{dom}(X)$, s'il s'agit d'un premier appel à $checkVar$-2 pour X (ligne 3), on procède comme d'habitude. Sinon, l'incrémentalité est partiellement

Algorithme 32 : checkVar-2($P = (\mathscr{X}, \mathscr{C})$: CN, X : Variable, cnt : Entier) : Booléen

```
1  modif ← faux
2  pour chaque a ∈ dom^P(X) faire
3      si cnt ≤ |𝒳| alors
4          P' ← AC(P|_{X=a}, {X})
5      sinon
6          P' ← AC(FC(P|_{X=a}, X), {Y ∈ 𝒳 t.q. cnt − lastModif[Y] < |𝒳|})
7      si P' = ⊥ alors
8          retirer a de dom^P(X)
9          modif ← vrai
10     sinon
11         pour chaque C ∈ 𝒞 | X ∈ vars(C) faire
12             soit Y ∈ 𝒳 | vars(C) = {X, Y}
13             pour chaque b ∈ dom^P(Y) | b ∉ dom^{P'}(Y) faire
14                 retirer {X_a, Y_b} de rel^P(C)
15                 lastModif[Y] ← cnt
16                 modif ← vrai
17 retourner modif
```

exploitée en supprimant d'abord au moins toutes les valeurs qui avaient été supprimées lors du dernier singleton test de X_a. On réalise cette opération par un appel à FC. Puis, on applique AC depuis une file de propagation composée de toutes les variables qui étaient concernées par au moins une inférence pendant les derniers $|\mathscr{X}| - 1$ appels à $checkVar$-2. Le reste de la fonction est identique à $checkVar$-1, à l'exception de la mise à jour de $lastModif$ (ligne 15) quand un couple de valeurs est supprimé ($lastModif[X]$ n'est pas mis à jour puisque cela a déjà été fait à la ligne 10 de l'algorithme 31).

Proposition 8. Appliqué à un graphe de contraintes complet, l'algorithme sDC-2 établit sPC.

Démonstration. Tout d'abord, la preuve que toute inférence effectuée par sDC-2 est correcte est immédiate. Quand $P' \leftarrow \mathrm{AC}(\mathrm{FC}(P|_{X=a}, X), Q)$ avec $Q = \{Y \mid cnt - lastModif[Y] < |\mathscr{X}|\})$ est effectuée à la ligne 4 de l'algorithme 32, on a $P' = \mathrm{AC}(P|_{X=a}, \mathscr{X})$, ce qui garantit que l'algorithme est complet. Cela signifie simplement que P' est soit arc-consistant soit égal à $\bot$. En effet, le réseau P est maintenu arc-consistant à chaque fois qu'une modification est effectuée (ligne 11 de l'algorithme 31) et toute inférence effectuée par rapport à une valeur X_a n'a aucun impact sur $P|_{X=b}$, b étant n'importe quelle autre valeur du domaine de la variable X. Cette propriété est vraie car quand toute inférence est effectuée, elle est enregistrée dans $lastModif$. □

Proposition 9. La complexité temporelle dans le pire des cas de sDC-2 est en $O(\lambda end^3)$ et sa complexité spatiale dans le pire des cas est en $O(ed^2)$.

Démonstration. Rappelons que les complexités temporelle et spatiale de sDC-1 sont respectivement $O(\lambda end^3)$ et $O(ed^2)$. Pour obtenir le même résultat pour sDC-2, il suffit de constater que la complexité temporelle dans le pire des cas cumulée à la ligne 14 de l'algorithme 31 est en $O(ed)$, et que la complexité spatiale de la structure $lastModif$ est en $\Theta(n)$. □

Algorithme 33 : sDC-3($P = (\mathscr{X}, \mathscr{C})$: CN) : CN

```
1  P ← AC(P, 𝒳)
2  Q_{X_a} ← ⊤, ∀X ∈ 𝒳, ∀a ∈ dom(X)
3  Q ← {X_a | X ∈ 𝒳 ∧ a ∈ dom^P(X)}
4  tant que Q ≠ ∅ faire
5  |  prendre X_a de Q
6  |  R ← checkValue(P, X_a)
7  |  removeTuples(P, R)
8  |  si P = ⊥ alors retourner ⊥
9  retourner P
```

Algorithme 34 : checkValue($P = (\mathscr{X}, \mathscr{C})$: CN, X_a : Variable$_{valeur}$) : {multiplet}

```
1  R ← ∅
2  si Q_{X_a} = ⊤ alors
3  |  P' ← AC(P|_{X=a}, {X})
4  sinon
5  |  P' ← AC(FC(P|_{X=a}, X), Q_{X_a})
6  Q_{X_a} ← ∅
7  si P' = ⊥ alors
8  |  retirer a de dom^P(X)
9  |  pour chaque Y ∈ 𝒳 | Y ≠ X faire
10 |  |  pour chaque b ∈ dom^P(Y) | (X_a, Y_b) est AC faire
11 |  |  |  ajouter Y_b à Q ; ajouter X à Q_{Y_b}
12 sinon
13 |  pour chaque Y ∈ 𝒳 | Y ≠ X faire
14 |  |  pour chaque b ∈ dom^P(Y) | b ∉ dom^{P'}(Y) faire
15 |  |  |  ajouter (X_a, Y_b) à R
16 retourner R
```

Algorithme sDC-3

Pour exploiter entièrement l'incrémentalité des algorithmes de consistance d'arc pour établir la chemin consistance forte, nous introduisons une structure de données spécifique à AC-3 [6] pour chaque valeur (l'idée est en fait liée à l'approche utilisée par [BESSIÈRE ET DEBRUYNE 2005] pour l'algorithme SAC-OPT). Cette structure, représentée par l'ensemble Q_{X_a} consiste en une file de propagation dédiée au problème $P|_{X=a}$. Le principe de sDC-3 est le suivant : si P_1 correspond à $\mathrm{AC}(P|_{X=a}, \mathscr{X})$, et P_i avec $i > 1$ représente le résultat du i^{e} établissement d'AC sur X_a, alors P_{i+1} correspond à $\mathrm{AC}(P'_i, Q_{X_a})$ où P'_i est un CN tel que $P'_i < P_i$ et $Q_{X_a} = \{X \in \mathscr{X} \mid \mathrm{dom}^{P'_i}(X) \subset \mathrm{dom}^{P_i}(X)\}$. La complexité temporelle cumulée dans le pire des cas pour effectuer ces établissements d'AC successifs sur le test de singleton de X_a est en $O(ed^3) = O(n^2d^3)$ puisque AC-3 est incrémental et d'un appel à l'autre, au moins une valeur a été supprimée d'un domaine.

6. Pour établir nos résultats de complexité pour sDC-3, il n'est pas indispensable d'utiliser un algorithme de consistance d'arc optimal

Algorithme 35 : removeTuples($P = (\mathscr{X}, \mathscr{C})$: CN, R : {multiplet})

```
1  initsize ← |R|
2  cnt ← 0
3  tant que R ≠ ∅ faire
4      cnt ← cnt + 1
5      prendre (X_a, Y_b) de R
6      retirer {X_a, Y_b} de rel^P(C) avec C ∈ 𝒞 | vars(C) = {X, Y}
7      si cnt > initsize alors
8          ajouter X_a à Q ; ajouter Y à Q_{X_a}
9      ajouter Y_b à Q ; ajouter X à Q_{Y_b}
10     pour chaque Z ∈ 𝒳 | Z ≠ X ∧ Z ≠ Y faire
11         pour chaque c ∈ dom^P(Z) | (Z_c, X_a) est AC ∧(Z_c, Y_b) est AC faire
12             si ∄d ∈ dom^P(X) | {Z_c, X_d} est consistante ∧{X_d, Y_b} est consistante alors
13                 ajouter (Z_c, X_a) à la fin de R
14             si ∄d ∈ dom^P(Y) | {Z_c, Y_d} est consistante ∧{X_a, Y_d} est consistante alors
15                 ajouter (Z_c, Y_b) à la fin de R
```

Pour établir sPC sur un réseau P, on fait appel à sDC-3 (algorithmes 33, 34 et 35). Comme indiqué ci-dessus, une file de propagation Q_{X_a} est associée à chaque valeur X_a. Nous introduisons également une file Q pour stocker l'ensemble des valeurs pour lesquelles un test de singleton doit être effectué. Notons que $X_a \in Q$ ssi $Q_{X_a} \neq \emptyset$. Initialement, AC est établie sur P, toutes les files dédiées sont initialisées à une valeur spéciale notée $\top$ (équivalente à $\mathscr{X}$ en pratique), et Q est remplie. Ensuite, tant que Q n'est pas vide, une valeur est prise de Q, un test de singleton est effectué sur cette valeur, et une ou plusieurs instanciations peuvent être supprimées.

Quand $checkValue$ (algorithme 34) est appelée, il est possible de déterminer si il s'agit du premier appel pour la valeur donnée X_a ou non, grâce à la valeur spéciale $\top$. Si $Q_{X_a} = \top$, il suffit alors d'établir AC sur $P|_{X=a}$ de manière classique. Sinon, on établit AC grâce à un appel à FC et à la file de propagation dédiée Q_{X_a}. Si on détecte que a n'est pas singleton arc-consistante, elle est supprimée et on cherche après toutes les valeurs Y_b compatibles avec X_a. Pour chacune de ces valeurs, on ajoute X à Q_{Y_b} (et Y_b à Q), puisque la prochaine fois que l'on cherchera à établir AC lors du test de singleton de Y_b, la valeur X_a qui était jusqu'ici présente aura disparu. Si $X \in Q_{Y_b}$, cette modification sera prise en compte lors du prochain test de singleton de Y_b.

Si a est bien SAC, on recherche les instanciations qui peuvent être supprimées et on les place dans un ensemble R. Quand $removeTuples$ (algorithme 35) est appelée, chaque instanciation de longueur 2 $\{X_a, Y_b\}$ est considérée tour à tour et est supprimée du CN (ligne 6). On ajoute alors Y à Q_{X_a} (et Y_b à Q) : au prochain test de singleton de Y_b, la valeur X_a qui était présente jusqu'ici (sinon, l'instanciation aurait déjà été supprimée) aura disparu. Il n'est pas nécessaire d'ajouter X à Q_{Y_b} si le couple (X_a, Y_b) a été placé dans R lors de l'exécution de $checkValue$ (en enregistrant la taille initiale de l'ensemble R et en utilisant un compteur, on peut détecter ce cas de figure). En effet, si c'est le cas, cela signifie que Y_b a été supprimée pendant le dernier singleton test de X_a.

Finalement, nous devons chercher toute valeur Z_c compatible à la fois avec X_a et Y_b. S'il n'y a pas de valeur X_d compatible avec à la fois Z_c et Y_b, cela signifie que l'instanciation $\{Z_c, X_a\}$ est un nogood et peut être supprimée. De même, s'il n'y a pas de valeur Y_d compatible à la fois avec Z_c et X_a, cela signifie que l'instanciation $\{Z_c, Y_b\}$ peut être supprimée.

Proposition 10. Appliqué sur un graphe de contraintes complet, l'algorithme sDC-3 établit sPC.

Éléments de démonstration. On utilise la propriété suivante : quand $P' \leftarrow \mathrm{AC}(\mathrm{FC}(P|_{X=a}, X), Q_{X_a})$ est effectué à la ligne 3 de l'algorithme 34, on a nécessairement $P' = \mathrm{AC}(P|_{X=a}, \mathscr{X})$. □

Proposition 11. La complexité temporelle dans le pire des cas de sDC-3 est en $O(n^3d^4)$ et sa complexité spatiale dans le pire des cas est en $O(n^2d^2)$.

Démonstration. La complexité temporelle dans le pire des cas de sDC-3 peut être calculée à partir des complexités temporelles dans le pire des cas cumulées de $checkValue$ et de $removeTuples$.

Tout d'abord, il est facile de constater que, dans le pire des cas, le nombre d'appels à $checkValue$ pour une valeur X_a donnée est en $O(nd)$. En effet, entre deux appels, au moins une instanciation I présente dans la relation associée à une contrainte impliquant X tel que $X_a \in I$ a été supprimée.

La complexité temporelle dans le pire des cas cumulée pour un appel à FC lors du singleton test de X_a est alors en $O(nd \times nd) = O(n^2d^2)$ tandis que la complexité temporelle cumulée pour établir AC lors du singleton test de X_a est en $O(ed^3) \in O(n^2d^3)$ grâce à la propriété d'incrémentalité d'AC-3.

Les lignes 8 à 11 ne peuvent être exécutées qu'une fois pour X_a et ne représentent qu'un coût en $O(nd)$. La complexité cumulée des lignes 13 à 15 pour une valeur X_a est en $O(n^2d^2)$.

La complexité temporelle dans le pire des cas résultante pour $checkValue$ pour une valeur est donc en $O(n^2d^3)$, et donc une complexité cumulée pour $checkValue$ en $O(n^3d^4)$ puisqu'il y a $O(nd)$ valeurs.

D'autre part, le nombre cumulé de tours de la boucle principale de $removeTuples$ est en $O(n^2d^2)$, puisque le nombre d'instanciations de longueur 2 dans le réseau de contraintes est en $O(n^2d^2)$ et puisque, à chaque tour, au moins une instanciation est supprimée (voir ligne 6 de l'algorithme 35). Comme la complexité temporelle dans le pire des cas d'un tour de la boucle principale de $removeTuples$ est en $O(nd^2)$, on déduit que la complexité temporelle dans le pire des cas de $removeTuples$ est en $O(n^3d^4)$. On déduit de ces résultats que la complexité temporelle dans le pire des cas de sDC-3 est en $O(n^3d^4)$.

En termes d'espace, rappelons que la représentation d'un CN se fait en $O(n^2d^2)$. Les structures de données introduite dans sDC-3 sont les ensembles Q_{X_a}, qui sont en $O(n^2d)$, l'ensemble Q en $O(nd)$ et l'ensemble R en $O(n^2d^2)$. On obtient donc $O(n^2d^2)$. □

Remarquons que si on utilise un algorithme d'AC optimal comme AC-2001, la complexité temporelle dans le pire des cas de sDC-3 reste en $O(n^3d^4)$ mais la complexité spatiale dans le pire des cas devient $O(n^3d^2)$, puisque la structure $last$ de AC-2001, en $O(n^2d)$, doit être gérée indépendamment pour chaque valeur. On peut également essayer de la partager, mais, contrairement à [BESSIÈRE ET DEBRUYNE 2005], l'intérêt est ici assez limité. D'autre part, on peut facilement choisir d'utiliser AC-3^{rm} (ou AC-3^{bit+rm}), puisque la structure res de cet algorithme peut être partagée simplement pour toutes les valeurs.

À propos des complexités

λ étant bornée par $O(n^2d^2)$, sDC-1 et sDC-2 peuvent atteindre $O(n^5d^5)$. Cela peut sembler plutôt élevé (c'est la complexité de PC-1), mais nous pensons que les deux algorithmes atteignent très rapidement le point fixe (en pratique, le nombre d'appels aux fonctions $checkVar$-1 et $checkVar$-2 reste relativement faible), puisque toutes les inférences sur les valeurs et surtout les instanciations inconsistantes peuvent être immédiatement prises en compte. D'autre part, sDC-2 bénéficie en partie de l'incrémentalité d'AC-3, et reste donc très proche de sDC-3 qui admet une bonne complexité temporelle dans le pire des cas en $O(n^3d^4)$. D'autre part, la proposition suivante indique que le temps perdu à appliquer un des trois algorithmes proposés sur un réseau déjà sPC est plutôt raisonnable. Dans ce cas, les trois algorithmes ont le même comportement.

Proposition 12. Appliqués à un réseau de contraintes sPC, la complexité temporelle dans le pire des cas de sDC-1, sDC-2 et sDC-3 est en $O(n^3d^3)$.

Démonstration. Si le réseau est sPC, le nombre de tests de singleton sera en $O(nd)$. Un test de singleton étant en $O(ed^2) \in O(n^2d^2)$ si on utilise un algorithme d'AC optimal, on obtient $O(n^3d^3)$. □

Proposition 13. La complexité dans le meilleur des cas de sDC-1, sDC-2 et sDC-3 est en $O(n^2d^2)$.

Démonstration. Le meilleur des cas survient quand toutes les contraintes sont universelles. En effet, dans ce cas, établir la consistance d'arc équivaut à un appel à FC, puisque il suffit de vérifier que chaque valeur de chaque variable est compatible avec l'assignation actuelle. Un test de singleton est alors en $O(nd)$, et la complexité globale est en $O(n^2d^2)$. □

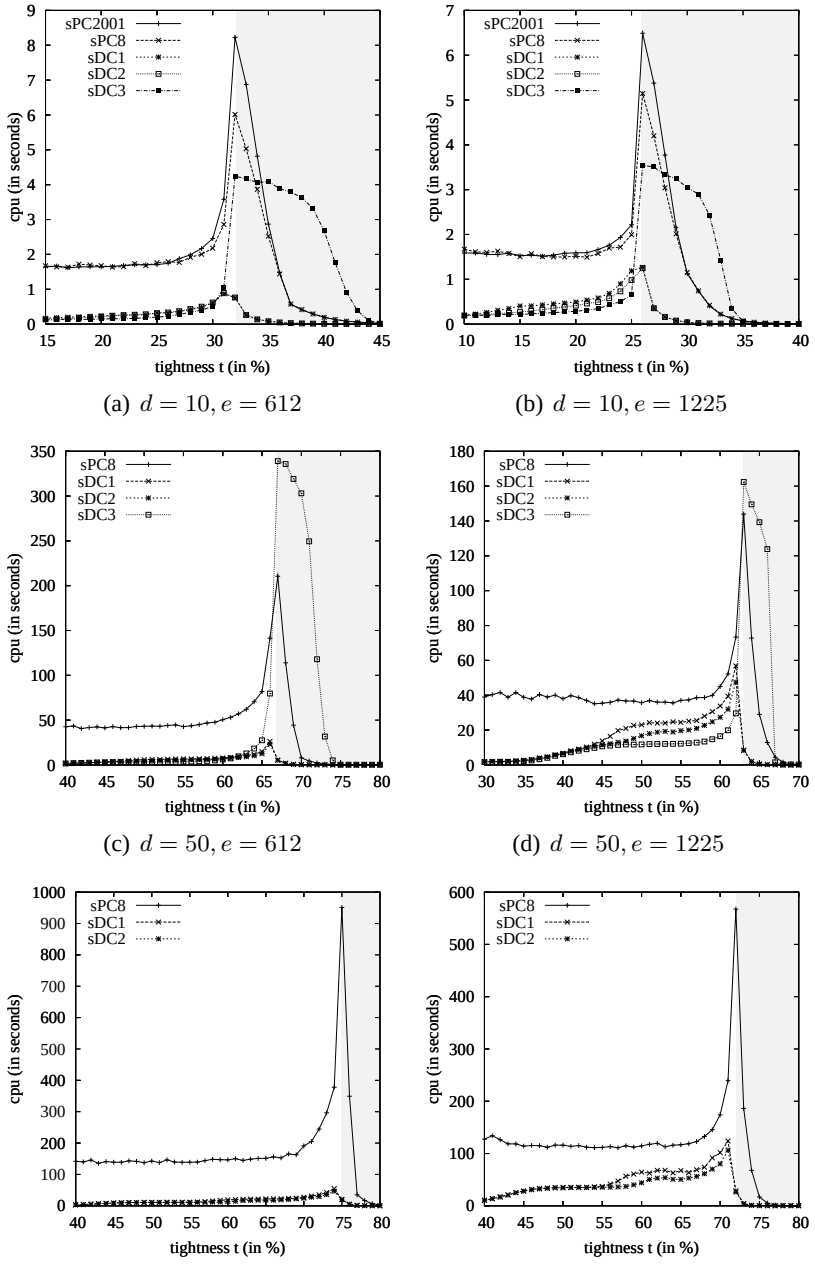
On peut considérer un autre cas intéressant : après avoir effectué une première passe d'AC (équivalente à FC), de nombreuses révisions peuvent être évitées en exploitant la Proposition 1 de [BOUSSEMART *et al.* 2004C,]. En considérant un réseau sPC et en supposant que toutes les révisions peuvent être évitées en utilisant cette *condition de révision* [MEHTA ET VAN DONGEN 2005B], la complexité temporelle dans le pire des cas devient $O(n^2d^2 + n^3d)$ puisque pour chaque test de singleton le nombre de révisions venant après FC est en $O(n^2)$, chacune d'entre elles étant en $O(1)$ puisqu'on évite les calculs au cours de ces révisions. On peut comparer ce résultat au coût de la phase d'initialisation PC-8 et PC-2001, qui est de $O(n^3d^2)$ dans le même contexte. Cela signifie que l'on peut s'attendre ici à une amélioration d'un facteur $O(\min(n,d))$.

2.3.3 Expérimentations

Pour montrer l'intérêt pratique des approches décrites dans cette section, nous avons conduit des expérimentations sur une machine virtuelle Sun Java 5 pour Linux, équipée d'un processeur Intel i686 cadencé à 2,4 Ghz et de 1 024 Mio de RAM. Nous avons comparé les temps CPU nécessaires pour établir sPC (ou prouver l'inconsistance) sur un réseau donné avec les algorithmes sDC-1, sDC-2, sDC-3, sPC-8 et sPC-2001. L'algorithme de consistance d'arc sous-jacent utilisé pour sDC était AC-3^{bit} équipé du mécanisme de condition de révision [BOUSSEMART *et al.* 2004C, MEHTA ET VAN DONGEN 2005B,]. Lors de chaque test, le graphe de contraintes est rendu complet par l'ajout de contraintes universelles avant l'application des algorithmes de filtrage.

Nous avons tout d'abord testé les diférents algorithmes sur des instances aléatoires, et constaté les résultats pour les classes de la forme $\langle 2, 50, d, 1225, t\rangle$ avec $d \in \{10, 50, 90\}$ et t compris entre $0{,}01$ et $0{,}99$. Nous avons également essayé les classes de la forme $\langle 2, 50, d, 612, t\rangle$, c'est à dire des instances aléatoires impliquant 50% de contraintes universelles explicites et 50% de contraintes de dureté t.

La figure 2.4 montre le temps CPU moyen nécessaire pour établir sPC sur ces différentes classes d'instances. La zone ombrée de chaque graphe indique les duretés pour lesquelles plus de 50% des instances générées ont été prouvées inconsistances (nous rappelons que nous n'utilisons pas d'algorithme de recherche lors de ces expérimentations). Tout d'abord, on peut remarquer que quand la taille du domaine d atteint 50 (respectivement 90), sPC-2001 (respectivement sDC-3) tombent à court de mémoire, et n'apparaissent donc pas sur les graphes. Ensuite, en concentrant notre attention sur les trois algorithmes introduits dans cette section, on peut faire les observations suivantes : pour de faibles duretés, sDC-1, sDC-2 et sDC-3 ont un comportement similaire, ce qui peut s'expliquer par le fait qu'aucune propgataion n'a lieu. Pour des densités proches mais inférieures au seuil (voir aussi figure 2.5), sDC-3 a un meilleur comportement que sDC-1 et sDC-2 puisqu'il bénéficie de l'exploitation totale de l'incrémentalité d'AC-3. Au niveau et après le seuil, sDC-3 est très pénalisé par son grain très fin, ce qui l'empêche de prouver rapidement l'inconsistance. Ce phénomène est particulièrement visible sur le graphe 2.4(d).



(a) $d = 10, e = 612$ (b) $d = 10, e = 1225$

(c) $d = 50, e = 612$ (d) $d = 50, e = 1225$

(e) $d = 90, e = 612$ (f) $d = 90, e = 1225$

FIGURE 2.4 – Résultats moyens obtenus sur 100 instances aléatoires binaires de classes $\langle 2, 50, d, e, t \rangle$.

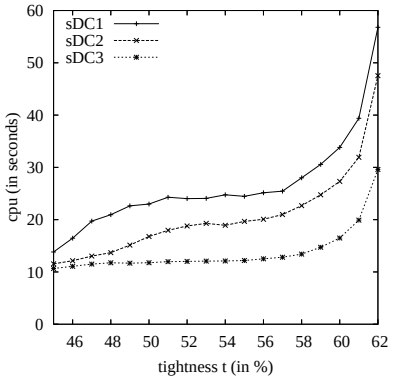


FIGURE 2.5 – Zoom sur le comportement moyen de sDC-1, sDC-2 et sDC-3 en dessous du seuil pour 100 instances aléatoires binaires de classes $\langle 2, 50, 50, 1225, t\rangle$.

D'autre part, le principal résultat de cette expérimentation est que sDC-2, légèrement plus efficace que sDC-1, est bien plus rapide que sPC-8 et sPC-2001. Pour des duretés faibles, l'écart entre sDC-2 et sPC-8 est particulièrement significatif (jusqu'à deux ordres de magnitude pour $d = 90$), ce qui est en partie dû aux révisions évitées comme décrit dans le paragraphe 2.3.2. Pour des duretés proches du seuil, l'écart reste important (environ un ordre de magnitude pour $d = 90$). On peut même observer que l'écart progresse quand la densité diminue. Ce n'est pas particulièrement surprenant puisque λ augmente avec le nombre de contraintes universelles, et les algorithmes sPC classiques travaillent directement sur les instanciations explicitement autorisées.

La table 2.11, représentant deux séries d'instances académiques, confirme les résultats obtenus sur les instances aléatoires. Sur ces instances structurées, sDC-2 est environ 20 fois plus efficace que sPC-8 sur les plus grandes, que des inférentes effectuées ou non. Les instances des n dames sont déjà sPC, ce qui n'est pas le cas des instances Langford.

2.3.4 En résumé

Dans cette section, nous avons introduit trois algorithmes permettant d'établir la consistance de chemin forte. Nous avons montré que l'algorithme sDC-2 était un bon compromis entre l'algorithme de base sDC-1 et l'algorithme plus fin sDC-3. Bien que la complexité dans le pire des cas de sDC-2 puisse sembler élevée, sa proximité avec sDC-3, qui admet une complexité proche de l'optimal, suggère son efficacité pratique. En pratique, sur des instances aléatoires, sDC-2 est légèrement plus lent que sDC-3 quand le nombre d'inférences est faible, mais bien plus rapide à la phase transition de la consistance de chemin. Comparé aux algorithmes sPC-8 et sPC-2001, ce dernier étant pourtant optimal, sDC-2 est généralement plus rapide d'un ordre de magnitude sur des instances de grande taille.

Peut-être peut-on s'interroger sur l'intérêt des algorithmes de consistance de chemin quand le graphe de contraintes n'est pas complet. En effet, quand un couple de valeurs est identifié comme étant non PC, il doit être supprimé du réseau. Quand aucune contrainte liant les deux variables impliquées n'existe dans le réseau, il faut introduire une nouvelle contrainte (et donc modifier le graphe de contraintes). Pour éviter cet inconvénient, il est possible d'établir les consistances de manière conservative, c'est à dire sans introduire de nouvelles contraintes. Cela introduit des consistances comme CPC [DEBRUYNE 1999] ou CDC, qui sera étudié dans la section suivante (2.4).

On pourra également envisager une autre alternative à l'introduction de nouvelles contraintes : enregistrer des nogoods dans une structure de données spécifique [FROST ET DECHTER 1994, SCHIEX

Instances		sPC-8	sPC-2001	sDC-1	sDC-2	sDC-3
queens-30	*cpu*	5,06	5,37	2,22	2,28	2,60
	mem	17	76	17	17	37
queens-50	*cpu*	50,90	–	4,60	4,50	5,30
	mem	30		22	22	149
queens-80	*cpu*	557,90	–	26,80	24,7	–
	mem	97		44	44	
queens-100	*cpu*	1 549,00	–	62,00	58	–
	mem	197		73	73	
langford-3-16	*cpu*	45,45	66,66	4,91	4,44	57,80
	mem	27	612	21	21	129
langford-3-17	*cpu*	63,48	–	6,06	6,07	76,79
	mem	34		22	22	157
langford-3-20	*cpu*	140,00	–	11,00	9,70	198,00
	mem	43		26	26	250
langford-3-30	*cpu*	1 247,00	–	60,00	50,00	–
	mem	138		56	56	

TABLE 2.11 – Résultats obtenus sur les instances académiques « n dames » et « Langford » : cpu en secondes et mem(oire) en Mio.

ET VERFAILLIE 1993], d'autant que cette approche a été récemment réétudiée par la communauté CSP [KATSIRELOS ET BACCHUS 2005, BOUTALEB *et al.* 2006, RICHAUD *et al.* 2006,]. En utilisant ces techniques, la consistance duale pourrait même être appliquée à des réseaux non binaires.

2.4 Consistance duale conservative

La consistance de chemin, et plus généralement les consistances de relation, sont quelque peu négligées par les développeurs de systèmes de programmation par contraintes. La raison principale est qu'exploiter de telles consistances implique de modifier les relations associées aux contraintes, et plus ennuyeux, de modifier la structure du graphe de contraintes associé au réseau. En effet, lorsque une instanciation est identifiée comme étant chemin-inconsistante, elle doit être éliminé du réseau. Lorsqu'il n'existe pas de contrainte dans le réseau portant sur les deux variables impliquées, une nouvelle contrainte doit être insérée (modifiant en conséquence le graphe de contraintes). Par exemple, le CN qui représente l'instance $scen$-11 du problème d'allocation de fréquences radio (RLFAP pour Radio-Frequency Assignment Problem) comporte 680 variables et $4\ 103$ contraintes. Dans le pire des cas, établir PC sur ce réseau entraînera la création de $C^2_{680} - 4\ 103 = 226\ 757$ nouvelles contraintes, ce qui peut réellement être contre-productif aussi bien en temps (la complexité des algorithmes de consistance d'arc dépendent généralement du nombre de contraintes) qu'en espace. De plus, les heuristiques basées sur le degré des variables déviennent inopérantes.

Cependant, il est possible d'éviter le défaut principal de PC en adoptant une approche *conservative*. cela signifie simplement que nous pouvons limiter notre attention aux contraintes existantes lorsque des instanciations inconsistantes sont recherchées. On obtient alors la consistance de chemin conservative (CPC pour Conservative Path Consistency) [DEBRUYNE 1999]. De la même manière, nous définissons la consistance duale conservative (CDC pour Conservative Dual Consistency). Nous montrons que PC est plus forte (en terme de filtrage) que CDC, qui elle-même est plus forte que CPC : CDC peut filtrer plus

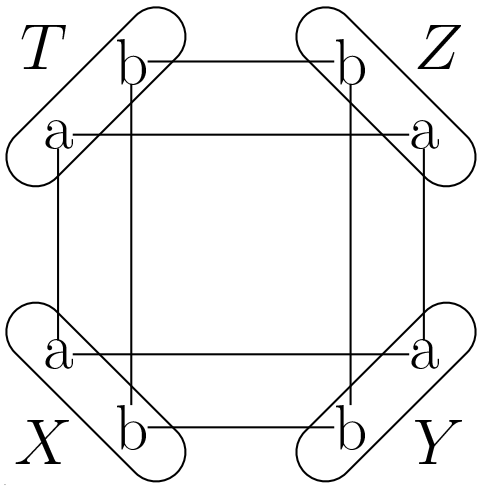


FIGURE 2.6 – Un réseau (il n'y pas de contraintes reliant X avec Z et Y avec T) qui est CDC et sCDC mais ni sPC, ni PC. Par exemple, (X_a, Z_b) n'est pas PC.

d'instanciations inconsistantes que CPC. Nous proposons d'utiliser l'algorithme simple sDC-1 présenté précédemment (section 2.3.2) pour établir la CDC (forte). Cet algorithme ne nécessite aucune structure de données spécifique (hormis celles de l'algorithme AC sous-jacent) et peut converger rapidement vers un point fixe (unique).

Nous définissons la consistance duale conservative :

Définition 23 (Consistance Duale Conservative)**.** Étant donné un CN $P = (\mathscr{X}, \mathscr{C})$,
- (X, Y) est dual-consistante conservatif (CDC) si et seulement si soit $\nexists C \in \mathscr{C} \mid \text{vars}(C) = \{X, Y\}$, soit (X, Y) est DC
- P est CDC si et seulement si $\forall (X, Y) \in \mathscr{X}^2 \mid X \neq Y$, (X, Y) est CDC.

De manière classique, un réseau est dit fortement dual-consistant conservatif (sCDC) si et seulement si il est à la fois dual-consistant conservatif et arc-consistant.

Les travaux présentés dans cette section ont été réalisés en coopération avec Stéphane Cardon et Christophe Lecoutre et ont fait l'objet à la fois d'un article ainsi que d'un poster, tous deux présentés lors de la 22e Conference on Artificial Intelligence (AAAI'2007) [LECOUTRE *et al.* 2007A,].

2.4.1 Étude qualitative

Nous avons démontré dans la section 2.3.1 que $\text{DC} = \text{PC}$ (proposition 5). Nous allons ici étudier les relations existantes entre la consistance duale conservative (CDC) et la consistance de chemin et sa variante conservative.

Proposition 14. $\text{PC} \succ \text{CDC}$.

Démonstration. Clairement, $\text{DC} \succeq \text{CDC}$, puisque CDC est une forme limitée de DC. Comme $\text{DC} = \text{PC}$ (proposition 5), nous obtenons $\text{PC} \succeq \text{CDC}$. De plus, la figure 2.6 montre un réseau qui est CDC mais pas PC. □

Proposition 15. $\text{CDC} \succ \text{CPC}$.

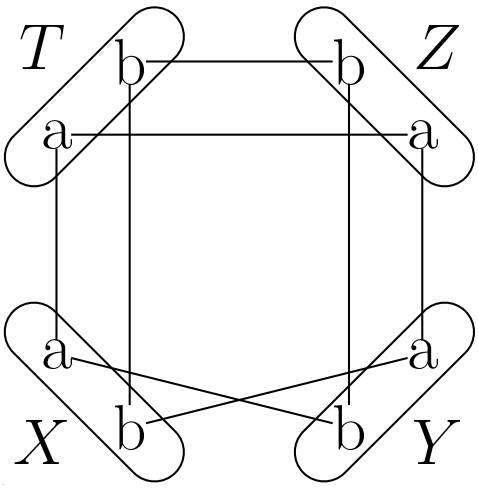


FIGURE 2.7 – Un réseau (il n'y a pas de contraintes reliant X avec Z et Y avec T) qui est CPC et sCPC mais ni sCDC, ni CDC. Par exemple, (X_a, T_a) n'est pas CDC puisque $\mathrm{AC}(P_{|X=a}) = \bot$.

Démonstration. Par un raisonnement similaire à la première partie de la démonstration de la proposition 5, nous pouvons montrer que $\mathrm{CDC} \succeq \mathrm{CPC}$. De plus, la figure 2.7 montre un réseau qui est CPC mais pas CDC. En effet, comme il n'y a pas de 3-clique, le réseau est obligatoirement CPC. □

Avant d'entrer dans les détails des relations entre sPC, sCDC et sCPC, remarquons qu'établir AC (une seule fois) sur un réseau PC est suffisant pour obtenir un réseau sPC. Ce fait bien connu reste vrai pour CDC. En effet, toute valeur arc-inconsistante est complètement isolée, comme montré dans la proposition suivante.

Proposition 16. Soit un réseau de contraintes $P = (\mathscr{X}, \mathscr{C})$ qui est PC (resp. CDC). Une valeur X_a de P n'est pas AC si et seulement si $\forall Y \in \mathscr{X}$ (resp. t.q. $\exists C \in \mathscr{C} \mid \mathrm{vars}(C) = \{X, Y\}$), $\forall b \in \mathrm{dom}(Y)$, (X_a, Y_b) n'est pas PC (resp. CDC).

Démonstration. Si une valeur X_a n'est pas AC, alors $\mathrm{AC}(P|_{X=a}) = \bot$. Toutes les instanciations incluant X_a ne seront donc pas CDC. X_a sera donc totalement isolée (X_a n'aura aucun support) après l'application de CDC sur P.

Comme $\mathrm{PC} \succ \mathrm{CDC}$, cette propriété reste vraie pour PC. □

Corollaire 2. $\mathrm{AC} \circ \mathrm{PC} = \mathrm{sPC}$ et $\mathrm{AC} \circ \mathrm{CDC} = \mathrm{sCDC}$.

Fait intéressant, la proposition précédente n'est pas vérifiée pour CPC. Considérons l'exemple mis en avant par la figure 2.8. Supposons que Z_b soit supprimée. Si nous filtrons par CPC, tous les multiplets en pointillés sont supprimés. Ensuite, si nous filtrons par AC, la valeur X_b est supprimée. Nous pouvons dès lors imaginer le même motif ($X' := X, Y' := Y, Z' := X, T := T'$) raccroché à notre figure par $X = Z'$. Et ainsi de suite, pour n'importe quel entier n, nous pouvons construire un réseau P tel que $(\mathrm{AC} \circ \mathrm{CPC})^{n+1}(P) = \mathrm{sCPC}(P)$ alors que $(\mathrm{AC} \circ \mathrm{CPC})^{n}(P) \neq \mathrm{sCPC}(P)$.

Proposition 17. $\mathrm{sPC} \succ \mathrm{sCDC}$.

Démonstration. $\mathrm{PC} \succeq \mathrm{CDC}$ par la proposition 14. De plus, par monotonie, nous déduisons $\mathrm{AC} \circ \mathrm{PC}(P) \leq \mathrm{AC} \circ \mathrm{CDC}(P)$. Ainsi, par le corollaire 2, nous obtenons $\mathrm{sPC} \succeq \mathrm{sCDC}$. Finalement, la figure 2.6 montre un réseau qui est sCDC mais pas sPC. □

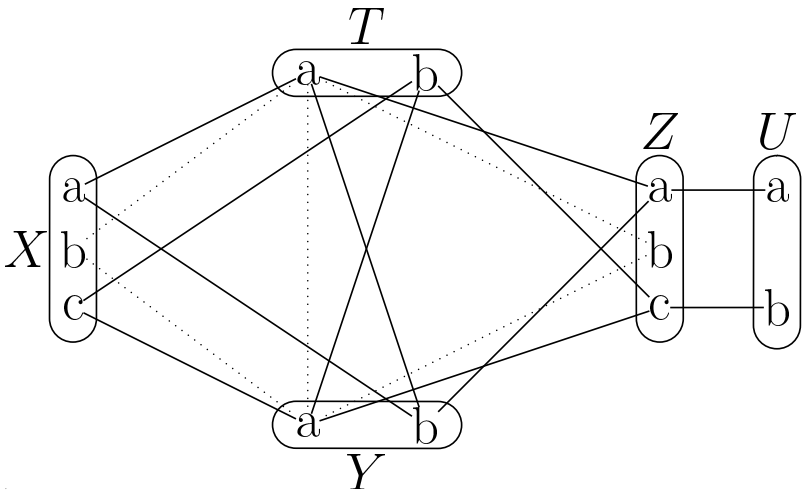


FIGURE 2.8 – Motif montrant que $\mathrm{AC} \circ \mathrm{CPC} \neq \mathrm{sCPC}$. Il n'y a pas de contrainte reliant X à Z.

Proposition 18. $\mathrm{sCDC} \succ \mathrm{sCPC}$.

Démonstration. Considérons l'hypothèse $H(n)$ suivante : $\mathrm{AC} \circ \mathrm{CDC}(P) \leq (\mathrm{AC} \circ \mathrm{CPC})^n(P)$. Pour $n = 1$, elle est immédiatement vérifiée puisque $\mathrm{CDC} \succ \mathrm{CPC}$. Maintenant, supposons $H(n)$ et montrons que $H(n+1)$ reste vraie. Nous avons, par hypothèse, $\mathrm{AC} \circ \mathrm{CDC}(P) \leq (\mathrm{AC} \circ \mathrm{CPC})^n(P)$. Donc, nous obtenons $\mathrm{CPC} \circ \mathrm{AC} \circ \mathrm{CDC}(P) \leq \mathrm{CPC} \circ (\mathrm{AC} \circ \mathrm{CPC})^n(P)$ par monotonie.

Comme $\mathrm{AC} \circ \mathrm{CDC}(P) = \mathrm{sCDC}(P) \leq \mathrm{CDC}(P) \leq \mathrm{CPC}(P)$, et comme CPC est inopérant sur un réseau sCDC, on obtient $\mathrm{CPC} \circ \mathrm{AC} \circ \mathrm{CDC}(P) = \mathrm{AC} \circ \mathrm{CDC}(P) \leq \mathrm{CPC} \circ (\mathrm{AC} \circ \mathrm{CPC})^n(P)$.

Par monotonie, nous avons donc $\mathrm{AC} \circ \mathrm{AC} \circ \mathrm{CDC}(P) \leq \mathrm{AC} \circ \mathrm{CPC} \circ (\mathrm{AC} \circ \mathrm{CPC})^n(P)$. Nous pouvons donc déduire $\mathrm{AC} \circ \mathrm{CDC} \leq (\mathrm{AC} \circ \mathrm{CPC})^{n+1}(P)$. Comme $\mathrm{sCPC}(P) = (\mathrm{AC} \circ \mathrm{CPC})^n(P)$ pour un quelconque entier n fini, nous avons montré que $\mathrm{sCDC} \succeq \mathrm{sCPC}$.

Finalement, la figure 2.7 montre un réseau qui est sCPC mais pas sCDC. □

Nous avons également cherché à comparer CDC avec PPC (voir section 1.2.2). Nous obtenons le résultat intéressant suivant :

Proposition 19. $\mathrm{CDC} \succ \mathrm{PPC}$

Démonstration. Si une instanciation $\{X_a, Y_b\}$ est CDC, alors elle est AC et si on effectue le test de singleton $X = a$, la valeur Y_b dispose encore d'un support dans chacune des variables voisines de Y. Chacun de ces supports dispose lui-même d'un support dans chacune de ses propres variables voisines. Et ainsi de suite, jusqu'à arriver à la variable X, où X_a est le support unique de toutes les valeurs de toutes les variables voisines puisque X a été réduit à un singleton. On peut ainsi prouver que tous les chemins allant de Y_b à X_a sont PC, et vice versa. L'instanciation $\{X_a, Y_b\}$ est donc PC. Si un CN P est CDC, alors tous les couples de valeurs sont PC. P est donc PPC.

Finalement, la figure 2.9 montre un CN triangulé CPC (et donc PPC) qui n'est pas CDC : l'instanciation $\{X_0, Y_1\}$ est CPC puisque Z_0, T_1 (et T_2) et V_0 sont compatibles à la fois avec X_0 et Y_1 (les triangles correspondants sont mis en gras sur la figure de gauche). Tous les autres couples de valeurs sont évidemment CPC. Cependant, si on effectue le test de singleton $X = 0$, on supprime X_1 et donc par propagation toutes les valeurs représentées par des cercles ainsi que tous les couples marqués en pointillés sur la figure de droite. En particulier, la valeur Y_1 est supprimée : $\{X_0, Y_1\}$ n'est donc pas CDC. □

La proposition suivante est triviale puisque toute valeur non SAC est supprimée par un algorithme de filtrage sCDC.

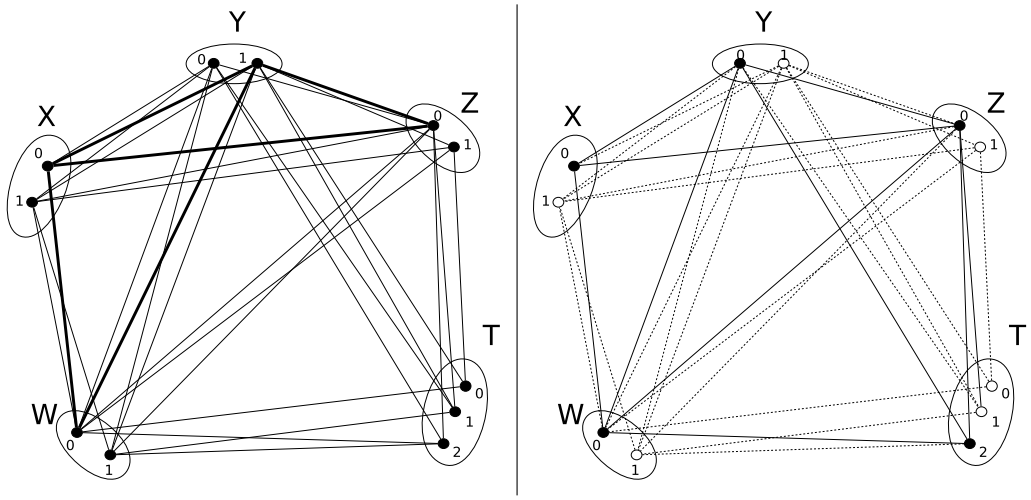


FIGURE 2.9 – Un CN triangulé CPC (et donc PPC) mais pas CDC : le couple (X_0, Y_1) n'est pas CDC.

Proposition 20. $\mathrm{sCDC} \succ \mathrm{SAC}$.

2.4.2 Établir la consistance duale conservative forte

Pour établir la consistance duale conservative forte, il nous suffit d'appliquer l'algorithme sDC-1 (algorithme 29 page 66). En effet, si on regarde attentivement les lignes 8 et 9 de la fonction $checkVar$-1 (algorithme 30), on constate que cette fonction fait le tour des contraintes existantes et ne supprime les instanciations dual-inconsistantes que dans ces contraintes. Les complexités temporelle et spatiale sont également identiques à celles de sDC-1 (respectivement $O(\lambda end^3)$ et $O(ed^2)$). Le facteur e est ici évidemment prépondérant, pusique le nombre de contraintes n'est pas forcément égal à C_n^2. Comme λ est borné par $O(ed^2)$, la complexité temporelle reste bornée par $O(e^2nd^5)$. Dans le cas conservatif, sDC-1 atteint également rapidement un point fixe, et le temps perdu à appliquer sDC-1 sur un réseau déjà sCDC est tout à fait limité.

Corollaire 3. Appliqué à un réseau sCDC, la complexité temporelle dans le pire des cas de sDC-1 est $O(end^3)$.

Il est également intéressant de comparer sDC-1 sur un réseau non complet avec les algorithmes établissant sCPC. Nous donnons quelques précisions sur les algorithmes sCPC-8 et sCPC-2001 qui peuvent être dérivés directement de PC-8 [CHMEISS ET JÉGOU 1998] et PC-2001 [BESSIÈRE *et al.* 2005,]. Tout d'abord, la complexité spatiale dans le pire des cas de ces deux algorithmes est respectivement $O(ed^2)$ et $O((e+K)d^2)$. La phase d'initialisation (nous ne discuterons pas de la phase de propagation) de ces deux algorithmes est la même : pour chaque 3-clique du graphe de contraintes, il faut vérifier que chaque instanciation de longueur 2 autorisée par une relation (associée à une contrainte) de cette clique est PC. Ceci est fait en $O(Kd^3)$. À partir de ces observations, il apparaît que, moins dense est le réseau, moins important est le nombre de 3-cliques, et plus faible est l'espace et le temps requis par ces algorithmes. D'un autre côté, si on considère des réseaux denses (e tendant vers C_n^2), on peut ajuster la complexité temporelle dans le pire des cas de la phase d'initialisation à $O(n^3d^3)$. Ceci constitue la même complexité temporelle dans le pire des cas qu'une seule passe (appeler $checkVar$-1 pour toutes les variables une seule fois) de sDC-1. Pour les réseaux peu denses, sDC-1 devrait être donc plus lent que sCPC-8 et sCPC-2001, pour les réseaux denses (ou hautement structurés), nous pensons vraiment que sDC-1, grâce à sa capacité d'effectuer des inférences rapidement, devrait être plus rapide.

	AC-3^{rm}	SAC-SDS	sCPC-8	sCPC-2001	sDC-1
Langford (4 instances)					
cpu	0,22	0,46	4,02	4,94	**0,52**
λ	105 854	105 769	75 727	75 727	75 727
blackhole-4-13 (7 instances) ($K = 92\,769$; $D = 20\%$)					
cpu	1,26	19,39	140,54	–	**46,91**
λ	8 206 320	8 206 320	8 206 320	8 206 320	**7 702 906**
$\langle 40\ 180\ 84\ 0{,}9\rangle$ (20 instances) (**K = 12** ; $D = 10\%$)					
cpu	0,71	10,57	2,28	**2,02**	17,42
λ	272 253	244 887	244 272	244 272	**210 874**
$\langle 40\ 8\ 753\ 0{,}1\rangle$ (20 instances) ($K = 8\,860$; $D = 96\%$)					
cpu	0,16	0,21	0,62	0,69	**0,20**
λ	43 320	43 320	43 318	43 318	43 318
job-shop enddr1 (10 instances) ($K = 600$; $D = 21\%$)					
cpu	1,58	4,06	7,91	10,54	**4,67**
λ	2 937 697	2 937 697	2 937 697	2 937 697	**2 930 391**
RLFAP scens (11 instances)					
cpu	0,86	–	25,96	–	**3,47**
λ	1 674 286	–	1 471 132	1 471 132	**1 469 286**

TABLE 2.12 – Résultats expérimentaux (cpu en secondes)

L'algorithme que nous proposons est aussi en rapport avec les algorithmes SAC-OPT et SAC-SDS [BESSIÈRE ET DEBRUYNE 2005] proposés pour établir la singleton consistance d'arc. SAC-OPT admet une complexité temporelle dans le pire des cas en $O(end^3)$ mais une complexité spatiale élevée dans le pire des cas en $O(end^2)$. SAC-SDS relâche l'optimalité temporelle pour sauver de l'espace mémoire : sa complexité temporelle dans le pire des cas est en $O(end^4)$ et sa complexité spatiale dans le pire des cas est en $O(n^2d^2)$. sDC-1 possède l'avantage de réduire plus efficacement l'espace de recherche, puisque $\mathrm{sCDC} \succ \mathrm{SAC}$, tout en limitant la complexité spatiale dans le pire des cas à $O(ed^2)$.

2.4.3 Expérimentations

De façon à montrer l'intérêt pratique de l'approche décrite dans ce papier, nous avons mené une expérimentation intensive sur une machine virtuelle Sun Java 5 pour Linux équipée d'un processeur Intel i686 2,4GHz avec 1 024 Mio de RAM. Nous avons comparé le temps CPU et le niveau de filtrage (la valeur λ obtenue après avoir appliqué l'algorithme) de différents algorithmes appliqués simplement (i.e. sans recherche). Ces algorithmes sont une version optimisée de AC3^{rm} [LECOUTRE ET HEMERY 2007], SAC-SDS, sCPC-8, sCPC-2001 et sDC-1. Plus précisément, nous avons utilisé $\mathrm{AC} \circ$ CPC-8 et $\mathrm{AC} \circ$ CPC-2001 comme approximation pour établir sCPC car nous avons observé qu'une seule passe était suffisant le plus souvent pour atteindre sCPC.

Nous avons tout d'abord testé les différents algorithmes sur différentes séries de problèmes [LECOUTRE 2006]. Les contraintes définies en intention (i.e. par un prédicat) pour certaines instances ont été converties en extension (ceci n'eut d'impact significatif sur le temps cpu que pour les grosses instances des séries $fapp$). Comme attendu (voir table 2.12), sCDC-1 filtre plus que les autres algorithmes : plus petite est la valeur de λ, plus réduit est l'espace de recherche. Hormis la série $\langle 2; 40; 180; 84; 0{,}9\rangle$, sDC-1 est entre 2 et 8 fois plus rapide que sCPC-8 et sCPC-2001. En outre, sDC-1 est presque aussi rapide que SAC-SDS qui, de son côté, nécessite trop de mémoire sur certaines séries (symbolisé par

$-$). Les instances aléatoires binaires de la classe $\langle 2; 40; 180; 84; 0{,}9\rangle$ ne comportent en moyenne que 12 3-cliques, ce qui explique pourquoi il est économique d'établir sCPC.

La table 2.13 fournit d'autres résultats représentatifs, instance par instance. Une nouvelle fois, il apparaît que, hormis l'instance $fapp$-01-200-4, sDC-1 surclasse largement les algorithmes qui établissent sCPC. Il y a une différence par un ordre de magnitude sur l'instance $haystack$-40 et par presque deux ordres de magnitude sur l'instance $knights$-50-5. La performance relativement mauvaise de sDC-1, en terme de temps cpu, sur l'instance $fapp$-01-200-4 peut une nouvelle fois être expliquée par la densité très faible (D est seulement à $0{,}5\%$) et le petit nombre de 3-cliques. Cependant, l'amélioration en terme de filtrage est tout à fait significative. De plus, sCDC-1 est bien moins consommatrice de mémoire que SAC-SDS et les algorithmes PC. La différence existante avec AC-3^{rm} provient du fait qu'un plus grand nombre de relations peut être partagé entre contraintes lorsque les relations ne sont pas modifiées.

Pour finir, nous avons comparé la performance de l'algorithme générique état-de-l'art MAC avec et sans sCDC établi en pré-traitement sur des instances difficiles RLFAP (Radio Link Frequency Assignment Problem). Même si des techniques (redémarrages, enregistrement de nogoods, etc.) permettent de résoudre plus efficacement ces instances, nous ne les employons pas ici de manière à observer l'impact réel (sur la recherche) de sCDC établi en pré-traitement. La table 2.14 montre que pour les instances les plus difficiles, sCDC en pré-traitement est payant : MAC seul est environ 40% plus lent que sCDC + MAC, et visite presque 2 fois plus de nœuds.

2.4.4 En résumé

Les prouveurs proposés pour la satisfaction de contraintes sont généralement construits sur la base d'un algorithme de recherche systématique ou locale. Effectuer des inférences, autant que possible, lors d'une phase de pré-traitement peut grandement améliorer leurs performances. Par exemple, établir la singleton consistance d'arc sur des réseaux fortement structurés peut s'avérer payant. Malheureusement, les algorithmes qui établissent SAC ne sont pas capables de prendre en compte une grande part de l'information qui est disponible lorsqu'ils sont exécutés. En effet, si une valeur Y_b est éliminée après avoir effectué l'assignation $X \leftarrow a$ et établi AC, nous sommes certains que $X = a$ et $Y = b$ ne sont pas compatibles.

Dans cette section, nous avons proposé de prendre en compte des inférences de ce type de manière conservative, i.e. sans ajouter aucune nouvelle contrainte. Nous avons introduit une nouvelle consistance appelée consistance duale (DC) et nous sommes focalisés sur sa variante conservative CDC. Il a été montré en particulier que CDC est une consistance de relation qui est plus forte que PC conservative (CPC), et qu'établir la CDC forte (i.e. établir à la fois CDC et AC) peut être réalisé de manière tout à fait simple et naturelle. Les résultats expérimentaux que nous avons obtenus sur un vaste échantillon de problèmes montrent clairement l'intérêt pratique de CPC sur les réseaux de forte densité.

2.5 Conclusion

Dans ce chapitre, nous avons avant tout cherché à développer des méthodes d'inférence génériques, capables d'améliorer les performances des algorithmes dédiés à la résolution d'un vaste ensemble de problèmes modélisés sous forme de CSP. Nous avons cherché à construire ces méthodes d'inférence autour des algorithmes de consistance d'arc les plus performants, et en particulier les algorithmes à « gros grain » basés sur GAC-3. GAC-3^{rm} est l'algorithme que nous considérions comme le plus rapide en pratique sur un grand nombre de problèmes pour établir la consistance d'arc généralisée au sein d'un algorithme de recherche de type MGAC ou un algorithme d'inférence de type SAC. Pour le cas binaire, en exploitant les opérations bit-à-bit, nous sommes parvenus à améliorer les performances de AC-3^{rm}.

	AC-3^{rm}	SAC-SDS	sCPC-8	sCPC-2001	sDC-1
driverlogw-09 ($K = 233\,834$; $D = 8\%$)					
cpu	1,60	48,42	33,84	36,52	**10,83**
mem	14	87	59	155	**23**
λ	369 736	147 115	306 573	306 573	**18 958**
haystack-40 ($K = 395\,200$; $D = 2\%$)					
cpu	9,64	–	580,48	–	**55,91**
mem	19	–	209	–	**107**
λ	48 670 518	–	48 670 518	–	48 670 518
knights-50-5 ($K = 10$; $D = 100\%$)					
cpu	12,38	34,43	1 759,00	–	**21,49**
mem	5	163	29	–	**19**
λ	31 331 580	0	0	–	0
pigeons-50 ($K = 19\,600$; $D = 100\%$)					
cpu	1,38	2,85	33,82	44,52	**2,7**
mem	2	12	9	636	**5**
λ	2 881 200	2 881 200	2 881 200	2 881 200	2 881 200
qcp-25-264-0 ($K = 43{,}670$; $D = 5\%$)					
cpu	2,28	6,08	8,15	10,49	**2,08**
mem	8	210	29	215	**21**
λ	77,234	77,234	76,937	76,937	76,937
qwh-25-235-0 ($K = 35\,700$; $D = 4{,}5\%$)					
cpu	1,87	5,62	7,09	9,05	**2,56**
mem	7	183	26	173	**19**
λ	56 721	56 721	56 380	56 380	56 380
fapp01-200-4 (**K = 247** ; $D = 0{,}5\%$)					
cpu	10,73	–	**16,05**	18,63	104,05
mem	15	–	22	254	**17**
λ	3 612 163	–	3 317 135	3 317 135	**2 117 575**
scen-11 ($K = 13\,775$; $D = 1{,}7\%$)					
cpu	2,87	–	85,82	78,49	**9,78**
mem	5	–	22	426	**16**
λ	5 434 107	–	4 829 442	4 829 442	**4 828 650**

TABLE 2.13 – Résultats expérimentaux (mem en Mio)

Instance		MAC	sDC-1 + MAC
scen11-f8	*cpu*	**8**	14
	nodes	$14K$	**5K**
scen11-f5	*cpu*	259	**225**
	nodes	$1\,327K$	**680K**
scen11-f3	*cpu*	2 338	**1 725**
	nodes	$12\,000K$	**5 863K**
scen11-f2	*cpu*	7 521	**5 872**
	nodes	$37\,000K$	**21 000K**
scen11-f1	*cpu*	17 409	**13 136**
	nodes	$93\,000K$	**55 000K**

TABLE 2.14 – Impact de sCDC en pré-traitement de MAC

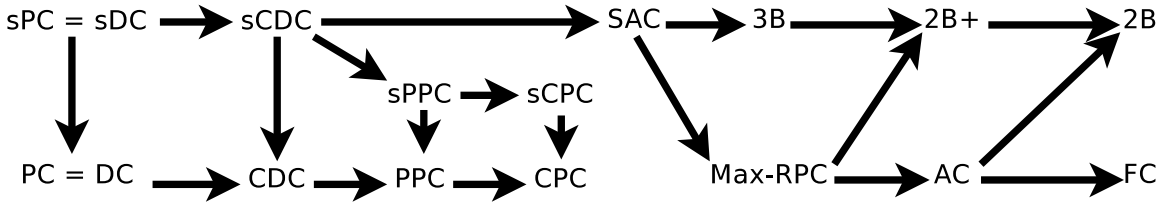


FIGURE 2.10 – Récapitulatif des consistances étudiées

Nous avons alors développé des algorithmes adaptés de AC, Max-RPC et SAC limités aux bornes des domaines des CSP discrets, puis décrit des formes d'inférence plus fortes mais toujours construites autour de la consistance d'arc : la consistance duale et la consistance duale conservative.

Nous avons à chaque fois étudié d'un point de vue théorique les comportements attendus de chaque algorithme, et toujours cherché à établir une comparaison exhaustive des nouvelles techniques avec les méthodes considérés comme les plus efficaces par l'état de l'art de la littérature. La figure 2.10 récapitule les plus importantes des consistances abordées dans ce chapitre. Une flèche $\rightarrow$ signifie « est strictement plus forte que ». L'absence de flèche signifie que les consistances sont incomparables (deux consistances ϕ et ψ sont incomparables s'il existe un CN ϕ-consistant mais pas ψ-consistant et un autre CN ψ-consistant mais pas ϕ-consistant).

Si la 2B consistance et ses développements semblent surtout efficaces sur certaines catégories de problèmes (impliquant des domaines de grande taille ou mettant en œuvre des contraintes d'inégalité, la consistance duale et la consistance duale conservative se sont révélés en pratique plus efficaces que les algorithmes de chemin consistance existants dans la plupart des cas, et pourraient finalement se révéler une excellente alternative à la singleton consistance d'arc classique.

Chapitre 3

Heuristiques de recherche

Pour trouver la solution à un problème NP-complet, les méthodes polynomiales d'inférence ne suffisent généralement pas (et, à moins que $P = NP$, il est peu problable que l'on puisse résoudre un CSP uniquement par inférence). On doit donc combiner les méthodes d'inférence à des algorithmes de recherche exponentiels (ou incomplets) afin de trouver une solution à un CSP ou de prouver qu'aucune solution n'existe.

L'algorithme le plus couramment utilisé pour résoudre les CSP reste l'algorithme complet M(G)AC, présenté dans la section 1.3.1 de ce document. La quasi-totalité des prouveurs présentés aux deux premières éditions de la compétition internationale de prouveurs de CSP (à l'exception de certaines versions de CSP4J, présenté dans le chapitre 4 de ce document, et d'un prouveur basé sur EFC) étaient des prouveurs basés sur M(G)AC. Pour limiter l'explosion combinatoire inhérente à cet algorithme, outre les méthodes d'inférence, des heuristiques de branchement ainsi que des méthodes d'analyse des conflits ont été développées.

On sait depuis longtemps que l'ordre dans lequel les hypothèses (ou décisions) sont effectuées ont un fort impact sur la taille de l'arbre de recherche. À chaque nœud de l'arbre de recherche, il faut choisir quelle *valeur* affecter à une *variable*. Jusqu'ici, ces décisions ont été effectuées en choisissant dans un premier temps la variable (sélection verticale), puis la valeur à affecter dans un second temps (sélection horizontale). La première étape de sélection a fait l'objet de nombreux travaux et de nombreuses heuristiques de choix de variable ont été proposées (cf section 1.3.2). L'heuristique $dom/wdeg$ est pour l'instant considérée comme l'heuristique générique la plus robuste. Cependant, le choix des valeurs (la deuxième étape de la décision) a été longtemps considéré comme marginal dans l'impact sur la taille de l'arbre de recherche. Dans la première partie de ce chapitre, nous proposons de nouvelles heuristiques basées sur le choix des valeurs. Ces heuristiques seront déduites des heuristiques de branchement utilisées pour résoudre le problème SAT, via l'utilisation des codages existants du problème CSP vers SAT.

Outre les heuristiques, ont peut également améliorer la performance des prouveurs en améliorant la flexibilité de la recherche. Le principal défaut de l'algorithme MGAC, c'est que si un mauvais choix est fait au début de la recherche, cela peut nécessiter de parcourir un espace très important de l'espace de recherche avant de pouvoir revenir sur cette erreur. Même les heuristiques les plus évoluées seront toujours susceptibles de faire ce type d'erreurs. Des mécanismes de redémarrage ont déjà été développés pour limiter ce défaut de MGAC, mais on constate aussi que les algorithmes de recherche locale ne sont pas affectés par ce défaut. Par contre, le fait que les algorithmes de recherche locale soient généralement incomplets et ne pouvant que difficilement bénéficier des mécanismes d'inférence ne joue généralement pas en leur faveur. L'idée d'exploiter la dualité des deux stratégies de recherche a germé depuis longtemps et le développement d'algorithmes hybrides reste un défi lancé à la communauté depuis plusieurs années [SELMAN *et al.* 1997,]. Dans la deuxième section de ce chapitre, nous présentons un moyen d'améliorer

les performances de MGAC couplé à $dom/wdeg$ en utilisant un algorithme de recherche locale basé sur la méthode Breakout. En alternant l'exécution des deux algorithmes, nous proposons d'échanger des informations dans un sens et dans l'autre afin de rendre la recherche la plus efficace possible.

3.1 Déduire de nouvelles heuristiques à partir des codages vers SAT

Il existe plusieurs facteurs expliquant le peu d'intérêt qui a été porté aux heuristiques de choix de valeurs. Tout d'abord, choisir une valeur pertinente à chaque point de branchement de l'algorithme MGAC nécessite souvent bien plus de calculs que choisir une variable, en particulier si l'on cherche à effectuer cette sélection de manière dynamique. D'autre part, sur des instances insatisfiables ou quand on cherche toutes les solutions, il faut obligatoirement supprimer toutes les valeurs d'au moins une variable, et l'ordre dans lequel les valeurs sont éliminées importe donc peu. Comme l'ont montré [SMITH ET STURDY 2005], cet argument ne tient en fait que quand la recherche est basée sur un algorithme à *branchement d-aire* mais pas sur un algorithme à *branchement binaire* (cf section 1.3.1).

On considère traditionnellement deux principes pendant la recherche : à chaque étape, on sélectionne la variable la plus contrainte, puis on sélectionne la valeur la moins contrainte (par exemple en utilisant l'heuristique $min\text{-}conflicts$ [FROST ET DECHTER 1995]). Ces principes correspondent à deux stratégies nommées $fail\text{-}first$ (échec précoce) et $promise$, et on cherchera toujours à faire correspondre une heuristique à l'une ou l'autre des stratégies [BECK *et al.* 2004, WALLACE 2005,].

[NEVEU ET PRCOVIC 2002] ont proposé une heuristique de choix de valeurs dynamique basée sur une méthode d'apprentissage par l'échec (on explore quelques sous-arbres en cherchant à apprendre des informations statistiques des échecs rencontrés). Bien que ces travaux aient considéré un branchement d-aire en se focalisant sur la recherche de solutions, ils semblent indiquer qu'il est possible de définir des heuristiques de choix de valeurs capables de réduire significativement l'effort de recherche. [SMITH ET STURDY 2005] indiquent que c'est particulièrement vrai pour certains types de contraintes, avec un branchement binaire. Nous avons cherché à développer de nouvelles heuristiques (en supposant bien sûr un algorithme de recherche à branchement binaire). En particulier, nous avons fait la correspondance entre le branchement binaire pour MAC et le branchement utilisé dans les prouveurs basés sur la procédure DPLL pour résoudre le problème SAT, et avons cherché un lien entre les heuristiques SAT et les heuristiques pour CSP. En effet, en considérant un codage quelconque d'un CSP en problème SAT, sélectionner un couple composé d'une variable et d'une valeur correspond à sélectionner un littéral dans SAT.

Dans cette section, nous montrons une correspondance directe entre $min\text{-}conflicts$ (respectivement $max\text{-}conflicts$) et le nombre maximal d'occurrences de littéraux dans la formule SAT obtenue en utilisant le codage par supports (respectivement le codage direct) d'une instance de CSP. Nous proposons également de nouvelles heuristiques de choix de valeurs dérivées de l'heuristique classique pour SAT *Jeroslow-Wang* (JW) [JEROSLOW ET WANG 1990]. Les heuristiques obtenues exploitent la bidirectionnalité des contraintes pour donner une meilleure approximation de la réduction de domaine qui découle de l'assignation d'une valeur à une variable, mais aussi quand une valeur est supprimée du domaine d'une variable donnée. L'exemple suivant illustre cette particularité :

Exemple 1. Soit C la contrainte binaire décrite par la figure 3.1. Notons que toute valeur du domaine de X apparaît dans deux multiplets autorisés et est en conflit avec deux valeurs du domaine de Y. Une heuristique de choix de valeurs classique comme $min\text{-}conflicts$ ne peut distinguer les différentes valeurs de X puisque toutes les valeurs de X ont le même nombre de supports dans Y.

Il n'est pas toujours suffisant de considérer uniquement le nombre de conflits (ou de supports) pour choisir la valeur la plus (ou la moins) prometteuse. En effet, en considérant une branchement binaire, si on affecte la valeur a à X (première branche de l'arbre de recherche), deux valeurs sont supprimées de

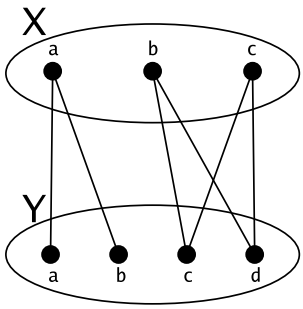


FIGURE 3.1 – Une contrainte C entre X et Y.

$\mathrm{dom}(Y)$. Si on supprime maintenant la valeur a de $\mathrm{dom}(X)$ (seconde branche de l'arbre de recherche), deux valeurs sont également supprimées de $\mathrm{dom}(Y)$. D'autre part, si on affecte la valeur b ou c à X, deux valeurs sont supprimées de $\mathrm{dom}(Y)$, et quand b ou c sont supprimés de $\mathrm{dom}(X)$, aucune valeur ne peut être supprimée de $\mathrm{dom}(Y)$. La valeur a est donc en fait plus contrainte que b ou c. Cette illustration montre qu'il peut être important de considérer l'impact sur les deux branches quand une heuristique évalue les valeurs à sélectionner.

L'estimation du nombre de valeurs supprimées quand une valeur du domaine d'une variable donnée est retirée (correspondant à la réfutation de l'hypothèse effectuée sur un nœud courant de l'arbre de recherche) n'a encore jamais été considérée lors de la conception d'heuristiques de choix de valeurs génériques, à l'exception de travaux récents et indépendants [SZYMANEK ET O'SULIVAN 2006]. D'autre part, notre approche peut être utilisée pour obtenir de manière plus générale une heuristique de choix de valeurs convenable pour n'importe quel type de contrainte.

Les travaux présentés dans ce chapitre ont été réalisés en coopération avec Christophe Lecoutre et Lakhdar Saïs et ont fait l'objet d'un article publié dans le Volume 1 de JSAT (Journal on Satisfiability, Boolean Modeling and Computation), édition spéciale sur l'intégration SAT/CSP [LECOUTRE *et al.* 2007D,].

3.1.1 Heuristiques pour MGAC

Nous nous référons ici à l'algorithme MGAC à branchement binaire décrit par l'algorithme 10 page 32 et nous nous focalisons sur le choix de l'hypothèse de branchement effectuée à la ligne 3 de l'algorithme. Au lieu de sélectionner directement un couple X_a, la plupart des prouveurs de CSP sélectionnent d'abord une variable, puis une valeur du domaine de cette variable. En utilisant cette technique, on a $O(n + d)$ éléments à considérer au total, alors que le choix global d'un couple (Variable, Valeur) nécessiterait de considérer $O(nd)$ éléments. En général, les prouveurs de CSP utilisent une heuristique de choix de variables orientées $fail\text{-}first$ comme dom, $brelaz$ ou $dom/wdeg$. Le choix de la valeur est considéré comme étant moins important, et on se contente généralement d'un ordre lexicographique ($lexico$). Notons que quand les valeurs des domaines sont mélangées, comme c'est généralement le cas lors des compétitions de prouveurs (mais pas dans l'utilisation réelle des prouveurs), les sélections lexicographique et aléatoire ($random$) des valeurs sont équivalentes. Les heuristiques de choix de valeurs plus évoluées exploitent l'ensemble connu de multiplets autorisés $\mathrm{rel}(C)$ associé à chaque contrainte C. L'heuristique de ce type la plus courante est nommée $min\text{-}conflicts$ (ou $max\text{-}supports$) et sélectionne la valeur présentant le moins de conflits (multiplets interdits), ou le plus de supports (multiplets autorisés), ce qui est équivalent [HULUBEI ET O'SULLIVAN 2005, GEELEN 1992, FROST ET DECHTER 1995]. $min\text{-}conflicts$ (respectivement $max\text{-}conflicts$) suit la stratégie $promise$ (respectivement $fail\text{-}first$).

Un avantage apparent des heuristiques $random$ ou $lexico$ reste le choix de la valeur en $O(1)$, là où les autres heuristiques sont généralement en $O(d\eta)$, η indiquant la complexité pour évaluer une valeur par l'heuristique ($\eta = d^{k-1}$ pour $min\text{-}conflicts$, par exemple). Cependant, il est possible de limiter cet inconvénient lié aux heuristiques évoluées en se limitant à une variante statique des heuristiques. L'ordre des valeurs est calculé pour chaque domaine lors d'une phase de prétraitement, avant le début de la recherche. En réordonnant les valeurs dans les domaines, on peut atteindre la complexité en $O(1)$ pour choisir la valeur [MEHTA ET VAN DONGEN 2005A]. À compter de ce point, nous considèrerons uniquement les variantes statiques des heuristiques de choix de valeurs.

3.1.2 Appliquer les heuristiques SAT aux CSP

La section 1.1.3 rappelle les deux principales conversions existantes de CSP vers SAT qui seront considérées dans cette section : le codage direct et le codage des supports, ainsi que la formulation générale de l'heuristique Jeroslow Wang « à deux faces » pour le littéral x de la CNF Σ : $H_w(\Sigma, x) = \sum_{x \in c} w(|c|) + \sum_{\neg x \in c} w(|c|)$, avec $c \in \Sigma$. C'est cette formulation qui sera utilisée dans la suite de cette section. On notera $\Sigma^D(P)$ la formule CNF obtenue par le codage direct du CN P et $\Sigma^S(P)$ la formule CNF obtenue par le codage par supports de P. En utilisant l'un et l'autre de ces codages, nous présentons maintenant les heuristiques de choix de valeurs obtenues à partir de l'instanciation des heuristiques SAT $H_w^{\otimes}$.

Appliquer $H_w^{\otimes}$ par le Codage Direct

Montrons tout d'abord que l'heuristique de branchement SAT sur le codage direct d'un CSP correspond aux heuristiques de choix de valeurs pour CSP $max\text{-}conflicts$ ou $min\text{-}conflicts$.

Définition 24. Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, $X \in \mathscr{X}$ et $a \in \mathrm{dom}(X)$. Le score, noté $D\text{-}H_w(P, X_a)$, de X_a dans P est défini comme suit :

$$D\text{-}H_w(P, X_a) = W[\mathrm{dom}(X)] + w(2) \times \sum_{C \in \mathscr{C} \mid X \in vars(C)} |\mathcal{C}fI(C, X_a)|$$

tel que

$$W[\mathrm{dom}(X)] = w(|\mathrm{dom}(X)|) + w(2) \times (|\mathrm{dom}(X)| - 1)$$

(On rappelle que $\mathcal{C}fI(C, X_a)$ représente l'ensemble des conflits de X_a dans la contrainte C)

Proposition 21. Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, $X \in \mathscr{X}$ et $a \in \mathrm{dom}(X)$
On a $D\text{-}H_w(P, X_a) = H_w[\Sigma^D(P), x_a]$

Démonstration. Chaque littéral positif x_a apparaît exactement une fois dans les clauses *au moins une*. La taille de la clause contenant x_a correspond à la taille du domaine de la variable X. Le littéral négatif $\neg x_a$ apparaît dans $(|\mathrm{dom}(X)| - 1)$ *au plus une* clauses binaires. Le terme $W[\mathrm{dom}(X)]$ correspond à ces clauses. D'autre part, $\neg x_a$ apparaît dans chaque clause binaire *conflit* correspondant aux multiplets incompatibles où apparaît X_a dans les contraintes impliquant X. □

Si l'heuristique de choix de valeurs $D\text{-}H_w^{\otimes}$ est appliquée uniquement sur les valeurs d'une variable donnée X (sélection horizontale), alors toutes les valeurs apparaissent exactement le même nombre de fois dans les clauses *au moins une* et *au plus une*. $W[\mathrm{dom}(X)]$ devient alors inutile, et en supprimant également le terme constant $w(2)$ on obtient :

$$D\text{-}H_w(P, X_a) = \sum_{C \in \mathscr{C} \mid X \in vars(C)} |\mathcal{C}fI(C, X_a)|$$

Seul le nombre de conflits apparaît encore dans la formule. Dans ce cas, $D\text{-}H_w^{\otimes}$ avec $\otimes = max$ (respectivement $\otimes = min$) définit le même ordre que $max\text{-}conflicts$ (respectivement $min\text{-}conflicts$).

Appliquer $H_w^{\otimes}$ par le Codage des Supports

Pour le codage des supports, la longueur des clauses dépend du nombre de supports d'une valeur dans une contrainte donnée. En considérant $H_w^{\otimes}$ sur $\Sigma^S(P)$, on dérive de nouvelles heuristiques de choix de valeurs intéressantes :

Définition 25. Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, $X \in \mathscr{X}$ et $a \in \mathrm{dom}(X)$. Le score, noté $S\text{-}H_w(P, X_a)$ de X_a dans P est défini comme suit :

$$S\text{-}H_w(P, X_a) = W[\mathrm{dom}(X)] + \sum_{C \in \mathscr{C} \mid X \in vars(C)} [W_{\downarrow}(C, X_a) + W_{\uparrow}(C, X_a)]$$

avec

$$W[\mathrm{dom}(X)] = w(|\mathrm{dom}(X)|) + w(2) \times (|\mathrm{dom}(X)| - 1)$$
$$W_{\downarrow}(C, X_a) = w(1 + |\mathcal{S}pI(C, X_a)|)$$
$$W_{\uparrow}(C, X_a) = \sum_{Y_b \in \mathcal{S}pV(C, X_a)} w(1 + |\mathcal{S}pI(C, Y_b)|)$$

(On rappelle que $\mathcal{S}pI(C, X_a)$ représente l'ensemble des supports de X_a dans la contrainte C)

Proposition 22. Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, $X \in \mathscr{X}$ et $a \in \mathrm{dom}(X)$. On a :
$S\text{-}H_w(P, X_a) = H_w(\Sigma^S(P), x_a)$

Démonstration. Comme pour le codage direct, le premier facteur $W[\mathrm{dom}(X)]$ correspond aux clauses *au moins une* et *au plus une*. D'autre part, pour chaque contrainte C impliquant la variable X, chaque littéral x_a correspondant à la valeur X_a apparaît dans les clauses *support* :

- une seule fois négativement. La taille de la clause est 1 (le littéral négatif $\neg x_a$) plus le nombre de supports de a dans la contrainte C. Ceci correspond au terme $W_{\downarrow}(C, X_a)$.
- positivement dans toutes les clauses support correspondant aux valeurs $Y_b \in \mathcal{S}pV(C, Y_b)$. Ces clauses sont de la forme $\neg y_b \vee x_{v_1} \cdots \vee x_{v_k}$, avec $\mathrm{vars}(C) = \{X, Y\}$, et Y_b étant un support de $v_1, \ldots, v_k$. La taille de ces clauses est 1 (l'occurrence du littéral négatif $\neg y_b$) plus le nombre de supports de Y_b dans C. Ceci correspond au terme $W_{\uparrow}(C, X_a)$.

□

Si l'on se limite au choix des valeurs dans une seule variable donnée, $W[\mathrm{dom}(X)]$ est constant pour toutes les valeurs et peut être ignoré.

En fonction de la fonction de pondération w et de l'opérateur $\otimes$, on peut maintenant concevoir à partir de $S\text{-}H_w^{\otimes}$ différentes nouvelles heuristiques de choix de valeurs qui exploitent la bidirectionnalité des contraintes (prend en compte les *deux* branches de l'algorithme de recherche à branchement binaire).

Instanciation 1 ($S\text{-}H_{occ}^{\otimes} = S\text{-}H_{w(\alpha)=1}^{\otimes}$). L'heuristique de choix de valeurs suivante est obtenue en utilisant $w(\alpha) = 1$ comme fonction de pondération :

$$S\text{-}H_{occ}(P, X_a) = |\mathrm{dom}(X)| + |\Gamma(X)| + \sum_{C \in \mathscr{C} \mid X \in vars(C)} |\mathcal{S}pI(C, X_a)|$$

Nous obtenons ici une heuristique de choix de valeurs correspondant à l'heuristique SAT basée sur le nombre d'occurrences des littéraux dans $\Sigma^S(P)$. Quand le choix est limité aux valeurs d'une variable donnée, l'ordre dans lequel les valeurs seront sélectionnées dépend uniquement du nombre de supports ($|\mathrm{dom}(X)|$ et $|\Gamma(X)|$ sont constants), qui est inversement proportionnel au nombre de conflits. $S\text{-}H_{occ}^{min}$ (respectivement $S\text{-}H_{occ}^{max}$) définit donc le même ordre que $max\text{-}conflicts$ (respectivement $min\text{-}conflicts$).

Instanciation 2 ($S\text{-}H^{\otimes}_{jw} = S\text{-}H^{\otimes}_{w(\alpha)=2^{-\alpha}}$). L'heuristique de choix de valeurs suivante est obtenue en utilisant $w(\alpha) = 2^{-\alpha}$ comme fonction de pondération :

$$S\text{-}H_{jw}(P, X_a) = W_{jw}[\mathrm{dom}(X)] + \tfrac{1}{2} \times \sum_{C \in \mathscr{C} \mid X \in vars(C)} \left(2^{-|\mathcal{S}pI(C,X_a)|} + W_{\uparrow jw}(C, X_a)\right)$$

avec

$$W_{jw}[\mathrm{dom}(X)] = 2^{-|\mathrm{dom}(X)|} + \tfrac{1}{4} \times (|\mathrm{dom}(X)| - 1)$$
$$W_{\uparrow jw}(C, X_a) = \sum_{Y_b \in \mathcal{S}pV(C,X_a)} 2^{-|\mathcal{S}pI(C,Y_b)|}$$

Instanciation 3 ($S\text{-}H^{\otimes}_{inv} = S\text{-}H^{\otimes}_{w(\alpha)=\alpha}$). On obtient l'heuristique de choix de valeurs suivante en utilisant $w(\alpha) = \alpha$ comme fonction de pondération :

$$S\text{-}H_{inv}(P, X_a) = 3\,|\mathrm{dom}(X)| - 2 + |\Gamma(X)| + \sum_{C \in \mathscr{C} \mid X \in \mathrm{vars}(C)} |\mathcal{S}pI(C, X_a)| + W_{inv}(C, X_a)$$

avec

$$W_{inv}(C, X_a) = |\mathcal{S}pI(C,X_a)| + \sum_{Y_b \in \mathcal{S}pV(C,X_a)} |\mathcal{S}pI(C, Y_b)|$$

De nouvelles heuristiques de choix de valeurs pour les CSP

Les seconde et troisième instanciations définissent de nouvelles heuristiques de choix de valeurs pour la résolution de CSP. En limitant l'évaluation aux valeurs d'une variable donnée, on obtient les simples heuristiques de choix de valeurs suivantes H_{jw} (respectivement H_{inv}) correspondant respectivement aux instanciations 2 et 3 dont les termes non discriminants ont été écartés.

Définition 26. Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, $X \in \mathscr{X}$ et $a \in \mathrm{dom}(X)$.

$$H_{jw}(P, X_a) = \sum_{C \in \mathscr{C} \mid X \in \mathrm{vars}(C)} \left(2^{-|\mathcal{S}pI(C,X_a)|} + \sum_{Y_b \in \mathcal{S}_p(C,X_a)} 2^{-|\mathcal{S}pI(C,Y_b)|}\right)$$

Par la suite, nous appellerons $max\text{-}jw$ la nouvelle heuristique de choix de valeurs H^{max}_{jw}. Dans l'exemple 1, nous obtenons $H_{jw}(P, X_a) = 1{,}25$ et $H_{jw}(P, X_b) = H_{jw}(P, X_c) = 0{,}75$

En utilisant l'opérateur de sélection $\otimes = max$, la valeur a est alors choisie. Ce choix correspond clairement à la meilleure évaluation des deux branches correspondant à $X = a$ et $X \neq a$. En effet, toutes les valeurs de $\mathrm{dom}(X)$, quand elles sont affectées à X, mènent à la suppression de deux valeurs du domaine de Y, alors qu'en supprimant a du domaine de X, deux valeurs sont également supprimées du domaine de Y là où aucune valeur ne serait supprimée en enlevant b ou c du domaine de X.

Cependant, dans l'exemple 1, H_{jw} ne permet pas de distinguer les différentes valeurs de Y ($\forall v \in \{a,b,c,d\} H_{jw}(P, Y_v) = 0{,}75$), bien qu'affecter la valeur a à Y supprime deux valeurs de $\mathrm{dom}(X)$ là où sélectionner $Y = c$ ne supprime qu'une valeur.

Définition 27. Soit $P = (\mathscr{X}, \mathscr{C})$ un CN, $X \in \mathscr{X}$ et $a \in dom(X)$.

$$H_{inv}(P, X_a) = \sum_{C \in \mathscr{C} \mid X \in \mathrm{vars}(C)} \left(2 \times |\mathcal{S}pI(C, X_a)| + \sum_{Y_b \in \mathcal{S}pV(C,X_a)} |\mathcal{S}pI(C, Y_b)|\right)$$

Codage	$H_w^{\otimes}$		**Heuristique de choix de valeurs**		**Stratégie**
			connue ?	nom	
(D)irect	$\otimes = max$	$w(\alpha) = 1$	O	$max\text{-}conflicts$	$fail\text{-}first$
	$\otimes = min$	$w(\alpha) = 1$	O	$min\text{-}conflicts$	$promise$
(S)upport	$\otimes = max$	$w(\alpha) = 1$	O	$min\text{-}conflicts$	$promise$
	$\otimes = min$	$w(\alpha) = 1$	O	$max\text{-}conflicts$	$fail\text{-}first$
	$\otimes = max$	$w(\alpha) = \alpha$	N	$max\text{-}inverse$	$promise$
	$\otimes = min$	$w(\alpha) = \alpha$	N	$min\text{-}inverse$	$fail\text{-}first$
	$\otimes = max$	$w(\alpha) = 2^{-\alpha}$	N	$max\text{-}jw$	$fail\text{-}first$

TABLE 3.1 – Résumé : heuristiques obtenues de D-$H_w^{\otimes}$ et S-$H_w^{\otimes}$

Par la suite, nous appellerons $max\text{-}inverse$ (respectivement $min\text{-}inverse$) la nouvelle heuristique de choix de valeurs H_{inv}^{max} (respectivement H_{inv}^{min}).

En appliquant H_{inv} sur les variables X et Y de l'exemple 1, nous obtenons :
$H_{inv}(P, X_a) = 8$, $H_{inv}(P, X_b) = H_{inv}(P, X_c) = 10$,
$H_{inv}(P, Y_a) = H_{inv}(P, Y_b) = 4$ et $H_{inv}(P, Y_c) = H_{inv}(P, Y_d) = 8$

On peut constater que $min\text{-}inverse$ mène au même ordre de valeurs sur la variable X que $max\text{-}jw$, alors que sur la variable Y, $min\text{-}inverse$ sélectionne une des valeurs a ou b et $max\text{-}jw$ donne le même score à toutes les valeurs du domaine de Y.

Toutes les heuristiques obtenues de S-$H_w^{\otimes}$ et D-$H_w^{\otimes}$ sont résumées dans la table 3.1, en supposant que la sélection des valeurs est limitée aux valeurs du domaine d'une même variable. En effet, sélectionner le meilleur couple X_v parmi tous les couples possibles est trop coûteux en temps CPU. Dans ce dernier cas, il n'existe cependant aucune correspondance entre les instanciations présentées ci dessus et les heuristiques de choix de valeurs connues pour les CSP : toutes les instanciations mènent à de nouvelles heuristiques de choix de couple (Variable, Valeur).

On peut par exemple constater grâce à la table 3.1 que l'heuristique de choix de valeurs dérivée de $H_w^{\otimes}$ (avec $\otimes = min$ et $w(\alpha) = 1$) en utilisant le codage *(D)irect* mène à l'heuristique *connue* (O) $min\text{-}conflicts$, de stratégie *promise*, ou encore que $H_w^{\otimes}$ (avec $\otimes = min$ et $w(\alpha) = \alpha$) en utilisant le codage des *(S)upports* mène à la nouvelle (N) heuristique appelée $min\text{-}inverse$, de stratégie $fail\text{-}first$.

Calculer les heuristiques

Dans le cas général (i.e. réseaux de contraintes impliquant des contraintes n-aires), le nombre maximal de multiplets qu'une contrainte peut admettre est en $O(d^k)$. Pour compter le nombre de supports pour chaque couple (Variable, Valeur), il faut énumérer pour chaque contrainte l'ensemble des multiplets, ce qui représente une complexité en $O(ed^k)$. Cette complexité correspond au coût pour établir H_{occ} (min- et $max\text{-}conflicts$) de manière statique, avant la recherche. Dans chaque domaine, les valeurs sont triées en fonction de leurs scores, ce qui permet pendant la recherche d'effectuer le choix de la valeur en $O(1)$.

Calculer H_{occ} à chaque nœud de l'arbre de recherche peut être envisagé, mais cela prend beaucoup de temps. Si une heuristique de choix de variables a sélectionné une variable X, la complexité pour sélectionner une valeur dans le domaine de cette variable est en $O[\Gamma(X)d^k] \in O(n^{k-1}d^k)$. Les tests expérimentaux que nous avons effectués dans cette voie se sont révélés catastrophiques, même sur des réseaux binaires.

Pour calculer H_{jw} et H_{inv}, il est possible de procéder en deux étapes. Tout d'abord, on calcule le nombre de supports pour chaque valeur en $O(ed^k)$ comme ci-dessus. Ensuite, le score de chaque valeur pour H_{jw} et H_{inv} peut être calculé facilement : en supposant que $SpV(C, X_a)$ peut être obtenu en $O(1)$,

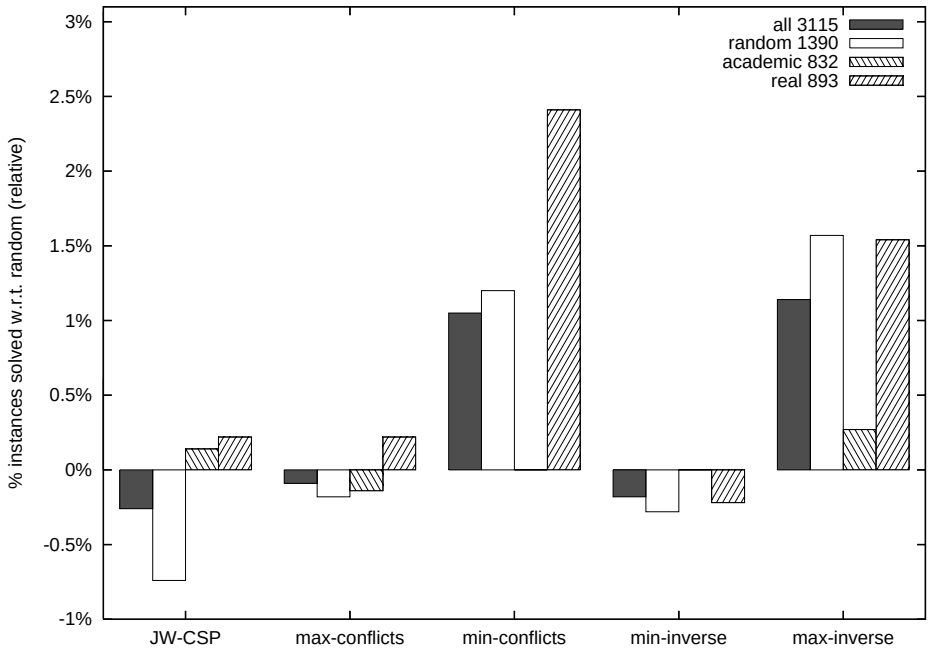


FIGURE 3.2 – Écart relatif du nombre d'instances résolues par les différentes heuristiques par rapport à l'heuristique $random$. ($timeout = 1200s$)

la complexité dans le pire des cas pour cette deuxième passe est la même que pour la première. On peut donc calculer H_{jw} et H_{inv} en $O(ed^k)$.

3.1.3 Expérimentationss

Pour montrer l'intérêt pratique de notre approche, nous avons implanté les différentes heuristiques décrites dans cette section sur notre plate forme $CSP4J$ [VION 2006] et conduit une expérimentation sur l'ensemble des 3 115 instances utilisées lors de la seconde compétition internationale de prouveurs de CSP [VAN DONGEN *et al.* 2006A, LECOUTRE 2006,] (à l'exception des instances pseudo-booléennes, celles-ci impliquant des contraintes implicites d'arité très élevée ne permettant pas d'énumérer les supports). CSP4J a fonctionné sur une machine virtuelle Sun Java 5 équipée d'un processeur Intel i686 cadencé à 2,4 GHz et de 1 024 Mio de RAM.

L'algorithme de recherche employé est MGAC équipé de l'algorithme d'inférence AC-3^{rm} et de l'heuristique de choix de variables $dom/wdeg$. Toutes les heuristiques de choix de valeurs ont été implentées de manière statique : un ordre est établi avant la recherche et reste fixe pendant tout le processus de recherche [MEHTA ET VAN DONGEN 2005A].

Le résultat de la comparaison des heuristiques sur l'ensemble des instances de la compétition est donné sur la figure 3.2. La figure montre l'écart relatif du nombre d'instances résolues en moins de 1200 secondes par rapport à l'heuristique de choix de valeurs $random$. Toutes les heuristiques présentant un écart relatif positif permettent de résoudre plus de problèmes que $random$. Les résultats généraux montrent que $max\text{-}inverse$ se révèle être la meilleure heuristique sur l'ensemble des instances. L'autre heuristique orientée $promise$, $min\text{-}conflicts$, donne également de bons résultats. Cependant, on peut constater que la plupart des instances académiques sont en fait facilement résolues par notre prouveur quelle que soit l'heuristique de choix de valeurs utilisée. Sur les instances aléatoires et industrielles (*real*), les heuristiques $promise$ présentent un bon comportement. En résumé, l'heuristique

Instance	$random$	$max\text{-}jw$	$max\text{-}cfl$	$min\text{-}cfl$	$min\text{-}inv$	$max\text{-}inv$
os-taillard-5-95-0	**50,58**	82,67	74,29	111,75	54,7	83,55
os-taillard-5-95-1	8,31	10,16	**5,82**	7,44	7,16	6,48
os-taillard-5-95-2	850,20	462,80	49,22	558,89	189,97	**46,25**
os-taillard-5-95-3	35,26	23,01	25,95	30,88	**22,34**	26,59
os-taillard-5-95-4	1 195,40	112,87	151,11	201,37	**97,33**	$timeout$
os-taillard-5-95-5	376,77	269,18	192,79	**71,93**	137,06	308,33
...						
median-5-95	89,39	78,51	72,03	92,94	**63,07**	107,29

os-taillard-5-100-0	$timeout$	248,64	6,68	45,12	**4,17**	$timeout$
os-taillard-5-100-1	45,07	73,91	24,29	**5,64**	12,87	158,54
os-taillard-5-100-2	$timeout$	**239,67**	$timeout$	$timeout$	$timeout$	$timeout$
os-taillard-5-100-3	$timeout$	$timeout$	**36,69**	692,08	$timeout$	$timeout$
os-taillard-5-100-4	**77,16**	$timeout$	$timeout$	$timeout$	$timeout$	$timeout$
os-taillard-5-100-5	$timeout$	$timeout$	829,96	703,45	$timeout$	**251,02**
...						
median-5-100	$> 1\,200$	$> 1\,200$	$> 1\,200$	**402,33**	$> 1\,200$	433,33

TABLE 3.2 – Résultats sur des isntances d'Open Shop de taille 5x5. Le timeout est fixé à 1 200 secondes de temps CPU. Les instances 5-95- sont insatisfiables, les instances 5-100- sont satisfiables

$max\text{-}inverse$ semble finalement la plus robuste.

Pour obtenir une idée plus précise des comportements des stratégies $promise$ et $fail\text{-}first$, nous donnons sur la figure 3.3 une comparaison détaillée entre les deux heuristiques proposées $max\text{-}inverse$ (de stratégie $promise$) et $min\text{-}inverse$ (de stratégie $fail\text{-}first$), sur toutes les instances satisfiables d'une part, et insatisfiables d'autre part. Les points proches de la diagonale correspondent à des instances résolues de manière similaire par les deux heuristiques. Les points sous la diagonal sont les instances pour lesquelles l'heuristique représentée en ordonnées (ici, $max\text{-}inverse$), résoud plus rapidement les problèmes, et vice versa. Un $timeout$ a été fixé à 600 secondes. Les instnances qui ont fait l'objet d'un $timeout$ pour l'une ou l'autre heuristique sont représentés sur l'axe $x = 600s$ ou $y = 600s$ correspondant.

Sur l'ensemble des instances, $min\text{-}inverse$ est clairement moins performantes que $max\text{-}inverse$ (le nuage de points est plus dense sous la diagonale). Cependant, si on distingue les instances satisfiables et insatifiables, on se rend compte que l'heuristique $min\text{-}inverse$ (de type $fail\text{-}first$) est en fait meilleure sur les instances insatisfiables. La figure 3.4 confirme ce comportement : on cherche ici à trouver toutes les solutions à un ensemble de problèmes aléatoires. Pour trouver toutes les solutions, il faut obligatoirement parcourir tout l'espace de recherche. Dans ce cas, une heuristique $fail\text{-}first$ comme $min\text{-}inverse$ parvient à réduire substantiellement la taille de l'arbre à développer.

Finalement, nous nous sommes intéressés tout particulièrement aux problèmes d'Open Shop, et avons testé nos heuristiques sur les instances d'Open Shop de Taillard [TAILLARD 1993] (cf section 1.1.2. Les instances d'Open Shop les plus difficiles sont les instances insatisfiables proches de la solution optimale : pour prouver qu'une solution en temps T est optimale, il faut prouver qu'il n'existe pas de solution en temps $T - 1$. La table 3.2 montre d'une part le temps CPU employé pour prouver qu'il n'existait pas de solution en temps $95\% \times T_{OPT}$, et d'autre part le temps CPU employé pour trouver une solution en temps T_{OPT}. Dix instances ont été testées dans chaque cas, et le résultat médian ainsi que les résultats

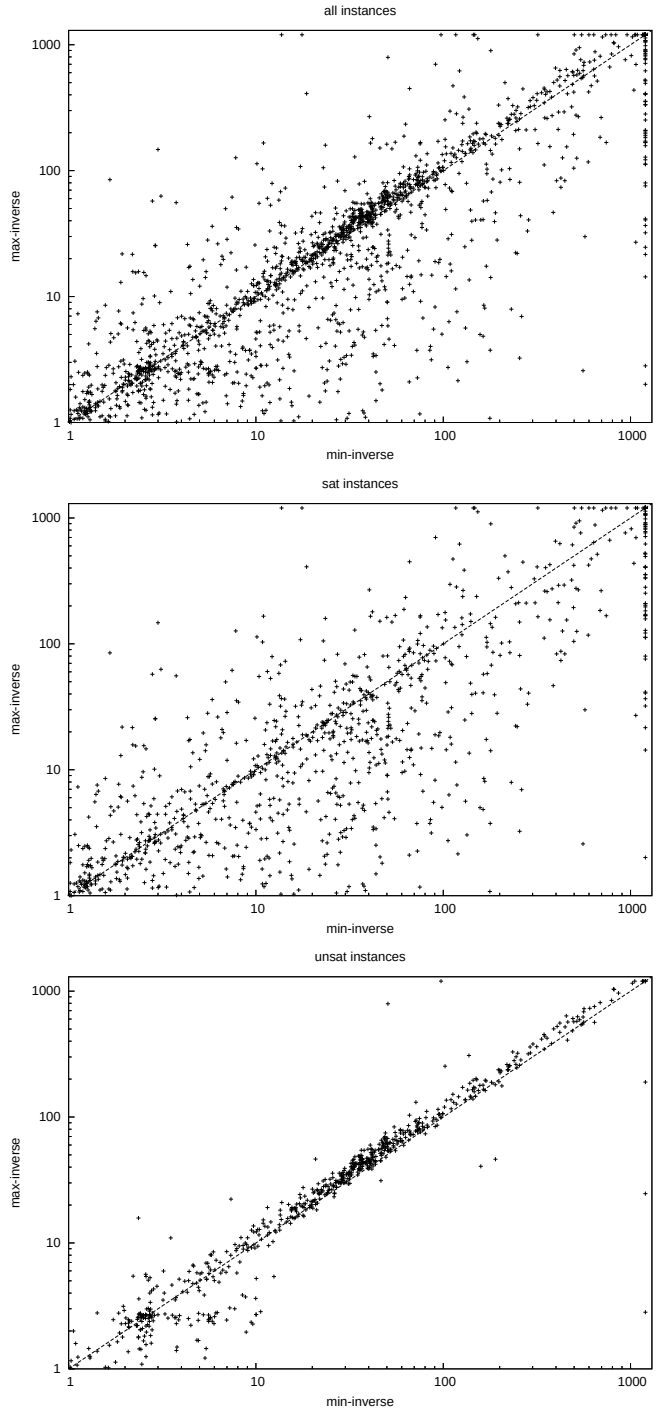


FIGURE 3.3 – Problème de décision : performances comparées de $min\text{-}inverse$ et $max\text{-}inverse$ (temps CPU par rapport au temps CPU, en secondes), sur respectivement toutes les instances, les instances satisfiables et les instances insatisfiables.

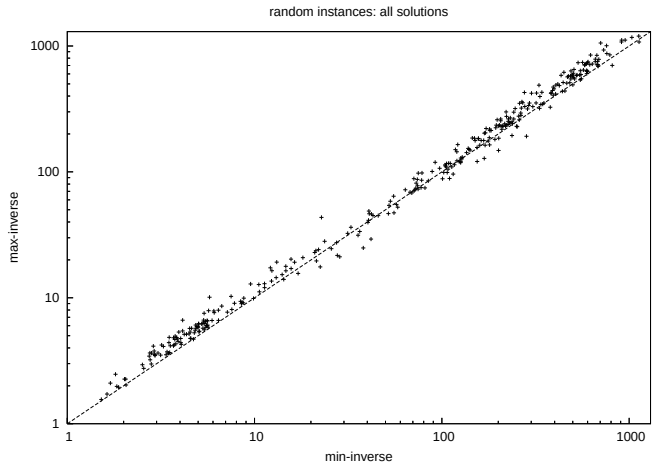


FIGURE 3.4 – Recherche de toutes les solutions : performances comparées de $min\text{-}inverse$ et $max\text{-}inverse$, temps CPU par rapport au temps CPU sur des instances aléatoires

pour les six premières instances sont montrés dans le tableau. Ici encore, les heuristiques $fail\text{-}first$ sont plus efficaces sur les instances unsatisfiables et les heuristiques $promise$ plus effiaces sur les instances satisfiables.

3.1.4 En résumé

Dans cette section, nous nous sommes attachés à convertir des heuristiques de branchement pour SAT en heuristiques de choix de valeurs pour les CSP. Grâces aux codages directs et par supports, nous avons montré comment dériver de nouvelles heuristiques de choix de valeurs pour les CSP. Ces heuristiques nous ont permis de mesurer pour la première fois l'impact des décisions positives mais aussi négatives effectués par les algorithmes de recherche à branchement binaire. En utilisant la formulation générale de Jeroslow-Wang « à deux faces », nous avons introduit une relation claire entre les relations existantes entre les heuristiques SAT et CSP.

Nous avons également montré expérimentalement que les heuristiques $fail\text{-}first$ (respectivement $promise$) sont plus efficaces pour résoudre des problèmes insatisfiables (respectivement satisfiables). Des variantes simples des heuristiques basées sur Jeroslow-Wang ($min\text{-}inverse$ et $max\text{-}inverse$) nous ont permis de résoudre plus d'instances satisfiables et insatisfiables parmi l'ensemble des instances issues de la compétition de prouveurs de CSP que les heuristiques classiques.

Nous avons constaté que pour les instances insatisfiables, ou pour chercher l'ensemble des solutions d'un problème, il est payant d'utiliser une stratégie de type $fail\text{-}first$. Nous pensons que ce phénomène s'explique par la capacité de l'heuristique de choix de variables $dom/wdeg$ de réfuter rapidement des sous-arbres insatisfiables. En effet, une heuristique de choix de valeurs $fail\text{-}first$ devrait permettre d'orienter la recherche sur les sous-arbres insatisfiables, et, une fois ceux-ci éliminés, l'espace de recherche est rapidement réduit.

Finalement, bien que nos résultats expérimentaux semblent confirmer que l'impact entre les différentes heuristiques de choix de valeurs reste faible, nous pensons qu'il peut être utile de bien comprendre l'impact des différentes stratégies de branchement. Selon nous, plus l'heuristique de choix de variables et les méthodes d'inférence seront efficaces, et plus il sera interessant de suivre le principe $fail\text{-}first$ au niveau des valeurs.

3.2 La méthode Breakout et les CSP

Les méthodes de recherche développées pour la résolution de CSP appartiennent généralement à une des deux grandes familles : algorithmes systématiques et algorithmes de recherche locale. À l'heure actuelle, on utilise généralement l'algorithme MGAC pour résoudre une instance de CSP. Le principal inconvénient de ces algorithmes basés sur le retour-arrière, c'est que si une mauvaise hypothèse est faite au départ, il faudra explorer un sous-arbre de taille potentiellement très importante sans succès. En établissant la consistance d'arc généralisée à chaque nœud de l'arbre, on traite en partie ce problème en détectant à l'avance les hypothèses qui ne mèneront à aucune solution [SABIN ET FREUDER 1994]. Une autre possibilité est d'utiliser des heuristiques de choix de variable permettant de se focaliser sur les parties les plus difficiles du CN, ce qui permet également de réduire le nombre d'hypothèses à effectuer avant d'effectuer des retours-arrière. $dom/wdeg$ est à ce jour une des heuristiques de choix de variables généralistes les plus efficaces [BOUSSEMART *et al.* 2004A,]. Bien qu'ils ne tirent généralement pas profit des méthodes d'inférence comme GAC, les algorithmes de recherche locale comme *Hill-Climbing Min-Conflicts* [MINTON *et al.* 1992,] ou la *Recherche Tabu* [GALINIER ET HAO 1997] sont souvent beaucoup plus efficaces que les algorithmes systématiques pour trouver une solution sur les problèmes de grande taille. Cependant, les algorithmes de recherche locale sont incomplets, c'est à dire qu'ils ne fournissent aucune garantie de trouver une solution ou de prouver l'incohérence d'un problème.

L'idée de combiner les avantages de la recherche systématique et de la recherche locale en concevant des algorithmes hybrides n'est pas neuve, et de nombreux travaux ont étudié la coopération entre la recherche locale et la recherche systématique, au point de devenir un véritable défi pour les problèmes de satisfaction de contraintes [SELMAN *et al.* 1997,]. Dans cette section, nous présentons une approche hybride consistant à alterner l'exécution de deux algorithmes, un de chaque type. De plus, nous proposons de faire interagir ces deux algorithmes entre eux et d'établir une coopération basée sur l'apprentissage et la communication d'informations d'une exécution à l'autre.

Les travaux présentés dans cette section restent préliminaires. Ils ont fait l'objet d'un article présenté lors des Troisièmes Journées Francophones de la Programmation par Contraintes (JFPC'2007) [VION 2007].

3.2.1 Recherches alternées

La manière la plus simple d'hybrider deux algorithmes reste de les lancer en parallèle, et de s'arrêter dès qu'un des deux renvoie une solution. Comme il est possible d'utiliser un système de redémarrage avec la plupart des algorithmes (en utilisant un critère d'expiration), nous proposons ici de les lancer en *séquence*. Dans les deux principes d'hybridation, les algorithmes restent indépendants. Ainsi, les avantages des deux stratégies se combinent : s'il existe une meilleure stratégie pour un problème donné (par exemple, la recherche locale pour les problèmes denses de grande taille, ou la recherche systématique pour les problèmes incohérents), un temps limité sera perdu avec les mauvaises stratégies de recherche.

De plus, grâce à notre principe séquentiel, il est possible d'extraire de nombreuses informations intéressantes d'une exécution à l'autre, même si la recherche échoue, d'une manière similaire aux techniques décrites dans [MAZURE *et al.* 1998,] ou [TERRIOUX 2001], dont nous discuterons ci-dessous. Nos travaux se focalisent sur l'alternance de l'algorithme systématique MGAC utilisant l'heuristique de choix de variables $dom/wdeg$ avec WMC. Nous montrons comment des informations peuvent être transmises d'un algorithme à l'autre, et comment la combinaison de ces algorithmes mène à une amélioration globale du processus de recherche sur les CSP structurés.

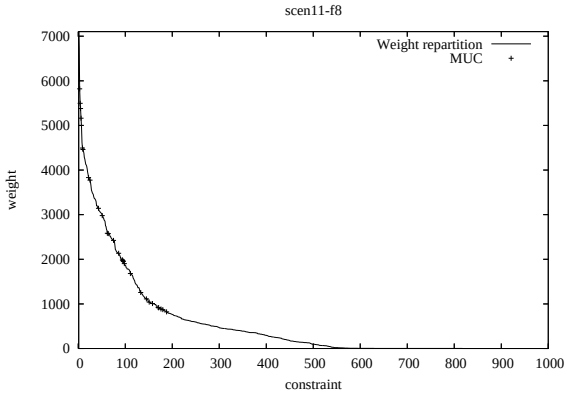


FIGURE 3.5 – Répartition des poids après une exécution de WMC sur scen11-f8

À propos de la pondération de contraintes

Le principal défaut des techniques de recherche systématique comme MGAC est bien connu : de mauvaises hypothèses au début de la recherche peuvent conduire à explorer de très grands sous-arbres qui peuvent être évités si l'heuristique est capable de se focaliser sur de petits sous-problèmes très difficiles ou même inconsistants. On dit alors que le prouveur est soumis au « *trashing* » : il redécouvre les mêmes incohérences un grand nombre de fois. D'autre part, on peut constater que certaines instances ne sont en réalité pas très difficiles à résoudre (par exemple, l'arbre minimal pour prouver leur incohérence est très réduit), mais peuvent avoir un comportement « *heavy tailed* » si elles sont mélangées aléatoirement et résolues plusieurs fois : lors de certaines exécutions, les temps de résolution sont alors très importants.

La pondération de contraintes utilisée par $dom/wdeg$ a été conçue pour identifier les sous-problèmes difficiles et permet de limiter grandement les phénomènes de *trashing* et *heavy-tail* sur les problèmes structurés [BOUSSEMART *et al.* 2004A,].

Mazure et al. [MAZURE *et al.* 1998,] indiquent que les statistiques sur les contraintes violées pendant la recherche locale peuvent être utilisées efficacement comme oracle pour une recherche systématique. On peut donc penser que le poids des contraintes obtenu après l'exécution de WMC se révèlera très utile pour initialiser les poids de $dom/wdeg$. La figure 3.5 montre l'efficacité pratique de cette hypothèse. Le graphe représente la répartition des poids des contraintes après une simple exécution de WMC sur un gros problème incohérent d'allocation de fréquences radio (RLFAP). Après 50 000 itérations, les 28 contraintes connues pour appartenir à au moins un noyau minimalement inconsistant (mises en évidence sur la figure) obtiennent un poids moyen de 2 590 alors que le poids moyen sur l'ensemble des 4 103 contraintes est de 136.

Utiliser une recherche systématique partielle pour extraire des nogoods

Le principal avantage de la recherche systématique réside dans sa capacité à prouver l'incohérence des problèmes. En utilisant une recherche binaire comme décrit en section 1.3.1, l'heuristique effectue une hypothèse de type $X = a$ à chaque nœud, ce qui crée deux sous problèmes : $\mathrm{GAC}(P|_{X=a})$ et $\mathrm{GAC}(P|_{X\neq a})$. Les deux sous-problèmes sont explorés en séquence jusqu'à ce qu'une solution soit trouvée ou que l'incohérence des deux sous-problèmes soit prouvée.

Afin d'améliorer les performances de la recherche et éviter le problème des mauvaises hypothèses, l'utilisation d'une stratégie de redémarrage combinée avec des heuristiques adaptatives s'est révélée très efficace pour résoudre les CSP avec MGAC [GOMES *et al.* 2000,]. Un nombre maximal de retour-arrières est fixé, et la recherche est interrompue et reprise du début avec de nouvelles hypothèses. Ce-

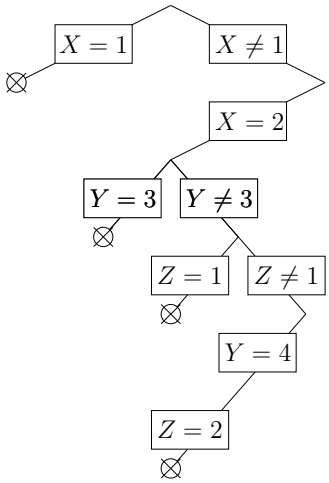


FIGURE 3.6 – Un arbre de recherche interrompue

pendant, cela signifie que toute la recherche effectuée avant le redémarrage est perdue. Lecoutre et al. proposent d'utiliser des nogoods afin de garder une trace des sous-problèmes inconsistants et éviter de les explorer plusieurs fois [LECOUTRE *et al.* 2007C,].

Quand la recherche expire, des nogoods sont identifiés à partir de données extraites d'une partie limitée (de taille linéaire) de la trace de l'arbre de recherche, et ajoutés au réseau de contraintes sous la forme de nouvelles contraintes. Tous les nogoods sont des conséquences logiques du problème et peuvent être considérés comme des contraintes additionnelles et redondantes qui enregistrent le travail effectué par les algorithmes. Pour des raisons pratiques, la taille des nogoods est limitée de sorte que le graphe de contraintes ne devienne pas trop dense. Plusieurs nogoods ayant la même portée sont fusionnés naturellement pour former une contrainte unique.

Le nombre de nogoods stockés à l'expiration d'une recherche est borné par le nombre d'hypothèses *encore en cours* au moment où la recherche est interrompue. Le nombre d'hypothèses positives ($X = a$) est borné par le nombre de variables n. La taille des nogoods stockés dépend directement du nombre d'hypothèses positives. Pour ne pas créer de contraintes d'arité trop importante, difficiles à gérer, on limite le nombre d'hypothèses positives prises en compte, et donc la taille des nogoods, à ρ (en général, on choisit $\rho = k$, l'arité maximale des contraintes). Le nombre d'hypothèses négatives est naturellement borné par nd. Le nombre de nogoods appris à l'interruption de chaque exécution est donc dans le pire des cas de $O(\rho nd)$. *On a la garantie que pour un arbre de recherche partiel donné, il ne peut y avoir de nogoods subsumés par (c'est à dire consistant une conséquence logique de) un autre nogood. Le nombre de nogoods est donc minimal.*

D'autre part, le *scope*, c'est à dire les variables impliquées par les nogoods appris, est naturellement limité. En particulier, la variable choisie comme premier point de choix, à la racine de la recherche, apparaîtra dans tous les nogoods de taille 2 ou plus. De même, les deux premiers points de choix apparaîtront dans tous les nogoods de taille 3 ou plus. De cette manière, on peut montrer que dans le pire des cas, l'ensemble des nogoods appris en fin de recherche représenteront $O(n\rho)$ *scopes* différents. Cela garantit que si on stocke les nogoods sous forme de contraintes, le nombre de contraintes additionnelles restera limité, en particulier si on limite ρ à une valeur relativement faible (en général, 2 ou 3).

Exemple 2. La figure 3.6 donne un exemple d'arbre correspondant à une recherche interrompue. L'heuristique a dans un premier temps effectué l'hypothèse $X = 1$, qui a été réfutée. L'hypothèse $X \neq 1$ est alors envisagée, ce qui revient à supprimer définitivement la valeur 1 de $\mathrm{dom}(X)$ (nogood de taille 1).

On effectue alors l'hypothèse $X = 2$, puis $Y = 3$, qui est également réfutée. On sait maintenant que

Algorithme 36 : learnNoGoods($P = (\mathscr{X}, \mathscr{C})$: CN)

```
1  scope ← ∅
2  tuple ← ∅
3  pour level ← 0 à |history| − 1 faire
4      X ← history[level]
5      scope ← scope ∪ X
6      si ∃C ∈ 𝒞 | vars(C) = scope alors
7          soit C la contrainte telle que vars(C) = scope
8      sinon
9          soit C la contrainte universelle telle que vars(C) = scope
10         𝒞 ← 𝒞 ∪ C
11     tuple ← tuple ∪ ⊥
12     pour chaque v ∈ dom^init(X) | removed[X_v] = level faire
13         tuple[level] ← v
14         supprimer tuple de C
15     tuple[level] ← dom(X)[0]
```

$(X, Y) = (2, 3)$ n'apparaîtra pas dans les solutions du CSP. Il s'agit d'un nogood de taille 2 (on peut l'écrire sous la forme $X \neq 2 \vee Y \neq 3$).

On envisage alors l'hypothèse $Z = 1$, qui est à nouveau réfutée. On peut donc apprendre le nogood $X \neq 2 \vee Z \neq 1$.

Finalement, on teste l'hypothèse $Y = 4$, puis $Z = 2$, cette dernière étant réfutée. On apprend donc le nogood $X \neq 2 \vee Y \neq 4 \vee Z \neq 2$.

On peut aller légèrement plus loin que l'approche proposée par [LECOUTRE *et al.* 2007C,] en enregistrant également toutes les valeurs supprimées par la consistance d'arc au moment où les hypothèses et leurs réfutations sont effectuées. Par exemple, si après avoir effectué l'hypothèse $X = 2$, on supprime les valeurs 1 et 2 du domaine de Y par consistance d'arc, on obtient également les nogoods $X \neq 2 \vee Y \neq 1$ et $X \neq 2 \vee Y \neq 2$.

Nous proposons d'utiliser l'algorithme 36 pour apprendre les nogoods, y compris ceux qui ont été déduits par consistance d'arc. Cet algorithme suppose que l'on dispose d'un ensemble ordonné $history$ comportant l'ensemble des variables faisant actuellement l'objet d'une hypothèse positive. D'autre part, on dispose également d'une structure $removed[X_v]$, qui stocke pour chaque couple (variable, valeur) le niveau où la valeur X_v a été supprimé (ou une valeur par exemple négative si la valeur n'a pas été supprimée).

Pour l'exemple 2, $history = \{X, Y, Z\}$, ou encore $history[0] = X$, $history[1] = Y$, etc. Dans le même exemple, on a aussi $removed[X_1] = 0$, $removed[Y_3] = 1$, $removed[Z_1] = 1$, $removed[Z_2] = 2$ (plus toutes les valeurs qui ont pu être supprimées aux divers niveaux par la consistance d'arc ou autre méthode d'inférence). On peut limiter $history$ à ρ éléments pour limiter la taille et le nombre de nogoods. On reconstitue les nogoods par ordre de taille, chaque « taille » correspondant à un tour de la boucle principale. Le premier tour correspond aux nogoods de taille 1, et va créer des contraintes unaires, ce qui correspond à supprimer des valeurs du domaine initial des variables. À chaque tour de boucle, le $scope$ évolue donc (ligne 5) : $\{X\}$ au premier tour, $\{X, Y\}$ au second, etc. Pour chaque $scope$, on recherche la contrainte correspondante, en introduisant une contrainte universelle le cas échéant (lignes 6-10). Finalement, pour chaque valeur supprimée au niveau correspondant, on supprime un multiplet de la contrainte (lignes 12-14). Tout comme la structure $scope$, la structure $tuple$ est progressivement créée

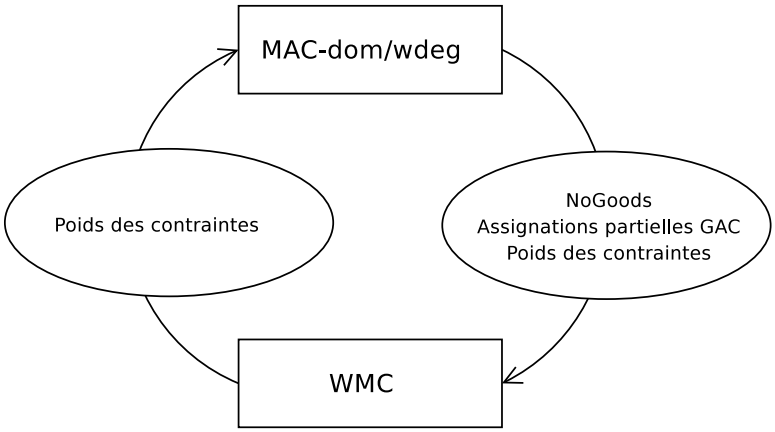


FIGURE 3.7 – Le système d'hybridation

à chaque tour de boucle (lignes 11 et 15). À noter qu'à la ligne 15, $\mathrm{dom}(X)$ est forcément un singleton puisque X a fait l'objet d'une hypothèse positive.

Les prochaines exécutions de WMC et MGAC pourront toutes deux utiliser ces contraintes additionnelles pour guider la recherche ou filtrer l'espace de recherche plus efficacement, d'une manière similaire aux propositions de [TERRIOUX 2001]. Cependant, il faut relativiser l'impact de ces nogoods, puisque l'apprentissage d'un nogood de taille ρ implique l'exploration d'une fraction potentiellement importante ($\frac{1}{d^\rho}$) de l'espace de recherche (en $O(d^n)$, pour mémoire, soit $O(d^{n-\rho})$ éléments de l'espace de recherche). Pour maximiser le nombre de nogoods appris par une exécution de MGAC, on utilise une heuristique de choix de valeurs orientée $fail\text{-}first$ comme $max\text{-}inverse$ (cf section 3.1).

Algorithme hybride

Une information supplémentaire peut être conservée depuis MGAC vers la recherche locale : quand MGAC expire, l'affectation partielle courante, ainsi que tout le réseau de contraintes résultant est arc-consistant. Il est possible de générer l'affectation complète initiale pour WMC à partir de ce réseau.

Toutes les variables affectées conservent leurs valeurs. Les autres variables sont affectées de manière gloutonne : une valeur est sélectionnée dans les valeurs arc-consistantes restantes, de sorte que aussi peu de contraintes que possible soient violées. WMC cherche alors à réparer cette affectation. Une récapitulation de l'algorithme final est présentée figure 3.7.

Bien que WMC et MGAC ne nécessitent pas d'autres paramètres que $maxIterations$ et $maxBT$, respectivement, trouver les bonnes valeurs fut extrêmement délicat, le paramètre idéal dépendant encore une fois fortement du problème à traiter. Une quantité limitée de ressources doit être allouée à chaque algorithme. Nous avons donc cherché à faire en sorte que chaque algorithme utilise autant de ressources l'un que l'autre.

La durée d'une exécution de WMC peut être contrôlée en fixant un nombre maximal d'itérations. La durée d'une itération dépend fortement de la taille du problème. Notre implantation Java effectue asymptotiquement environ $\frac{25\ 000\ 000}{nd}$ itérations par seconde sur des CSP aléatoires binaires avec un processeur i686 cadencé à 3 GHz. Notez que pour des problèmes industriels comme les RLFAP (Radio Link Frequency Assignment Problem) ou FAPP (Frequency Assignment Problem with Polarization constraints), nd atteint facilement 20 000 et dépasse les 2 000 000 sur les plus grosses instances (qui sont cependant facilement résolues par MGAC de par la nature hautement hétérogène de ces problèmes). Notre implantation Java de MGAC effectue asymptotiquement environ $\frac{200\ 000\ 000}{ed^2}$ hypothèses par seconde sur des problèmes binaires aléatoires.

Algorithme 37 : Hybrid(P = $(\mathscr{X},\mathscr{C})$: CN, $maxIter$: Entier) : Booleen

```
1  maxTries ← 1
2  maxBT ← maxIter × 8n/ed
3  répéter
4      startTime ← now()
5      répéter ⌊maxTries⌋ fois
6          essayer
7              retourner WMC(P, maxIter)
8          capturer Expiration
9      WMCDuration ← now() − startTime
10     startTime ← now()
11     essayer
12         retourner MGAC(P, maxBT)
13     capturer Expiration
14     MGACDuration ← now() − startTime
15     learnNoGoods(P)
16     maxTries ← α × maxTries
17     maxBT ← α × maxBT × WMCDuration/MGACDuration
```

L'algorithme hybride final est décrit par l'algorithme 37. Nous utilisons un mécanisme de levée et de capture d'exceptions lorsque le critère d'interruption d'un algorithme est atteint, aux lignes 7 et 12. Dans ce cas, l'algorithme n'est pas interrompu, mais reprend à la ligne suivant la capture de l'exception. À chaque itération de l'algorithme hybride principal, $maxTries$ exécutions de WMC sont effectuées avec $maxIterations$ itérations au maximum (lignes 5-8), puis MGAC est exécuté une fois avec une limite de $maxBT$ retour-arrières (lignes 11-13). À chaque fois, $maxTries$ et $maxBT$ sont augmentés par un facteur α, de sorte que MGAC puisse être complet. Dans le pire des cas, ceci surviendra quand $maxBT \geq d^n$. $maxBT$ est ajusté dynamiquement à la ligne 17 de sorte que les ressources allouées à la recherche locale et à la recherche systématique soient identiques.

3.2.2 Discussion

L'idée d'hybrider une recherche locale avec une recherche systématique n'est pas nouvelle, et plusieurs travaux étudient la coopération entre les deux types de recherche (voir section 1.3.4). L'approche hybride que nous présentons dans le présent article tombe dans la première catégorie d'hybridation que nous avons recensée, avec quelques élements des autres sections. Ainsi, notre idée principale reste d'alterner entre une recherche locale et une recherche systématique, ce qui appartient clairement à la première catégorie. En apprenant et en conservant des informations sur le problème d'une exécution à l'autre, un algorithme peut guider l'autre, et vice versa. Cet apprentissage appartient plutôt aux deux autres catégories. Un des principaux avantages de la première catégorie est que l'on se base sur des algorithmes longuement étudiés, et que toute amélioration apportée à l'une des techniques va profiter à l'ensemble du système.

L'approche proposée dans [MAZURE *et al.* 1998,] est particulièrement intéressante, puisque, comme la nôtre, elle tombe dans la première catégorie. Mazure et al. choisissent d'effectuer plusieurs exécutions d'une recherche locale en utilisant l'algorithme TSAT (un algorithme de recherche Tabu pour le problème SAT), avec un nombre limité de *flips* et *tries*. Si TSAT échoue, des statistiques obtenues sur les contraintes falsifiées pendant la recherche sont utilisées pour résoudre le problème via un algorithme complet, ainsi

que pour extraire un noyau inconsistant. Cependant, contrairement à notre algorithme, TSAT n'utilise pas directement le poids des contraintes, et l'heuristique utilisée par la recherche systématique n'est pas adaptative. Tout en confirmant les résultats de Mazure et al. appliqués aux CSP, nous pensons donc que notre approche est à la fois plus naturelle et plus robuste.

[EISENBERG ET FALTINGS 2003] avaient déjà émis l'idée que les poids obtenus après une exécution d'un algorithme utilisant la méthode Breakout pouvaient mettre en évidence des noyaux insatisfiables. Eisenberg montre expérimentalement que cela fonctionne sur les instances de type coloration de graphes. Nous montrons que cela fonctionne également sur des instances réelles difficiles, et autant pour des instances insatisfiables que sur les instances satisfiables. Notre algorithme va plus loin en proposant d'apprendre des informations dans les deux sens

Pour montrer l'efficacité de la pondération de contraintes pour identifier les parties les plus difficiles des problèmes, nous avons essayé de lancer chaque algorithme sur des instances incohérentes du problème *Queens-Knights*. Il s'agit d'une fusion du problème cohérent des Reines avec le problèmes incohérent des Cavaliers, comme décrit dans [BOUSSEMART *et al.* 2004A,]. Le problème constitué par 50 reines et 5 cavaliers est modélisé par 55 variables ($d = 2\ 500$) et 1 235 contraintes. Il contient un noyau minimalement inconsistent (MUC) de 5 variables et 5 contraintes. Après une première exécution de WMC avec 50 000 itérations, les 5 contraintes impliquées dans le MUC obtiennent un poids de 5 127 ou 5 128. Toutes les autres contraintes ont un poids inférieur ou égal à 2. Si l'on effectue une simple exécution de MGAC avec 4 000 retour-arrières (ce qui consomme la même quantité de ressources que 50 000 itérations de WMC sur ce problème), les 5 contraintes du noyau ont respectivement un poids de 1 891, 1 765, 120, 1 et 1. Trois autres contraintes obtiennent un poids de 2 et les autres contraintes gardent leur poids initial de 1.

Cette expérimentation montre que WMC peut être bien plus efficace que MGAC pour identifier des noyaux incohérents, et prouve l'efficacité potentielle de notre approche. Résoudre le problème des *Queens-Knights* avec notre prouveur hybride est fait en une exécution de WMC de 100 secondes suivi d'une exécution de MGAC de 78s (2 468 retour-arrières). L'algorithme MGAC seul effectue 6 exécutions pour un total de 328s et 12 652 retour-arrières.

3.2.3 Expérimentations

Les expérimentations sont effectuées par notre prouveur CSP4J, conçu en langage Java 5, sur un cluster de machines virtueles Sun Java 5 pour pour Linux, chacune équipée d'un processeur x86-64 cadencé à 2,4 GHz et 1 Gio de RAM. Nous avons comparé les performances des algorithmes suivants (les paramètres sont choisis pour être empiricalement optimaux pour des problèmes aléatoires binaires au seuil à 50 variables et 23 valeurs) :

- MGAC avec redémarrages seul (en augmentant à chaque redémarrage $maxBT$ de 50%) et apprentissage de nogoods comme décrit en section 3.2.1. Notre algorithme MGAC est basé sur l'algorithme d'établissement de l'arc consistance généralisée GAC3^{rm} [LECOUTRE ET HEMERY 2007]
- MCRW avec p fixé à 0,4 et $150\ 000$ itérations par tour
- Tabu avec une liste Tabu fixée à 30 élements et $150\ 000$ itérations par tour
- WMC avec $2\ 000$ itérations par tour
- Notre approche hybride avec $2{,}000$ itérations par tour pour WMC, et avec $maxBT$ et $maxTries$ augmentés de 50% à chaque tour ($\alpha = 1{,}5$).

Les problèmes aléatoires, de nature très homogène, sont une bonne référence pour estimer le comportement asymptotique des algorithmes. Nous avons comparé MGAC, WMC ainsi que notre algorithme hybride sur une série de problèmes aléatoires difficiles présentant des duretés variées. Nous fixons $n = 50$ et $d = 23$ et faisons varier t et e de telle sorte que tous les problèmes soient situés au seuil ($e = -\frac{n \ln d}{\ln(1-t)}$). Quand la dureté des contraintes diminue, le nombre de variables (et donc la densité du graphe) augmente,

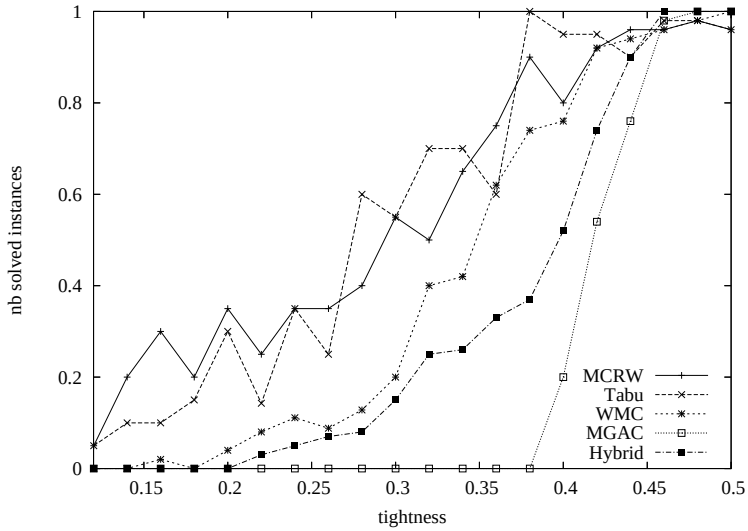


FIGURE 3.8 – Résolution d'instances aléatoires

et vice versa. Pour que le théorème permettant le calcul du seuil soit applicable, la dureté des contraintes doit être inférieure à $0{,}5$. D'autre part, il faut que $e \leq C_n^2$ pour pouvoir concevoir le problème, ce qui correspond à $t = 0{,}12$ pour les problèmes à 50 variables et 23 valeurs d'après la formule ci-dessus. On a également forcé les problèmes à être cohérents (cf section 1.1.2). *Toutes les instances sont au seuil et cohérentes.* On essaye alors de résoudre les différentes instances obtenues dans un délai de 600 secondes, et on compte combien d'instances ont pu être résolues.

Les résultats obtenus sont montrés sur la figure 3.8. Ils montrent que, bien que MGAC soit extrêmement puissant sur les problèmes peu denses, aux contraintes dures, les algorithmes de recherche locale sont plus efficace à l'opposé, sur les problèmes denses aux contraintes lâches. Ce sont aussi des problèmes considérés comme plus difficiles. Sur des problèmes aussi homogènes que ces instances aléatoires, la pondération de contraintes est pratiquement sans effet. Les performances décevantes de l'algorithme hybride sont donc compréhensibles. Cependant, on peut constater que l'algorithme hybride est plus robuste, dans le sens où le nombre global d'instances résolues est plus important que pour MGAC seul, tout en restant susceptible de résoudre des instances incohérentes. D'autre part, la mesure de la densité ou de la dureté n'est pas fiable pour des problèmes structurés hétérogènes, puisqu'ils peuvent être composés de plusieurs sous-problèmes difficiles ayant chacun une valeur différente de dureté ou de densité. On ne peut donc utiliser ces mesures, dans l'état actuel des choses, pour sélectionner le meilleur algorithme pour un problème donné. L'utilisation d'algorithmes hybrides robustes est donc particulièrement intéressante dans un environnement « boîte noire », où l'on ne peut compter sur aucune information sur la nature des problèmes.

Nous avons également exécuté les différents algorithmes sur toutes les instances de la Seconde Compétition Internationale de Prouveurs CSP [VAN DONGEN *et al.* 2006A,]. Les algorithmes de recherche locale n'étant pas conçus pour résoudre les problèmes impliquant des contraintes d'arité élevée (ce qui nécessiterait un traitement non booléen des contraintes [GALINIER ET HAO 2004]), les problèmes contenant des contraintes d'arité supérieure à 4 ont été ignorés (il s'agit essentiellement des problèmes de pseudo-booléens ou impliquant des contraintes globales). Nous avons conservé les instances pour lesquelles au moins un des algorithmes a pu trouver une solution, et avons éliminé les instances incohérentes.

		MGAC		MCRW		Tabu		WMC		Hybrid	
série	nb	rés	temps	rés	temps	rés	temps	rés	temps	rés	temps
all interval	13	12	**0,44**	**13**	0,89	12	0,57	**13**	0,68	**13**	2,45
qcp-qwh-bqwh	253	248	0,78	**253**	0,60	**253**	**0,41**	246	0,76	**253**	2,11
aim	96	**94**	0,52	64	8,95	63	17,20	86	**0,50**	82	0,85
fapp	147	141	11,20	**142**	4,20	62	>600,00	132	**2,44**	141	125,77
ruler	9	**8**	17,42	5	59,34	5	**8,87**	6	12,00	**8**	9,98
shop	127	103	1,91	76	46,08	56	>600,00	**110**	**1,18**	104	6,33
par	18	**16**	0,89	10	**0,56**	10	**0,56**	10	**0,56**	14	7,24
queens	20	17	5,92	18	0,98	**19**	0,92	18	**0,84**	**19**	32,54
ramsey	11	9	4,68	**11**	2,60	9	3,54	**11**	**1,71**	**11**	22,28
rlfap	25	**25**	2,68	**25**	1,97	19	1,71	21	**1,14**	24	0,96
rand-d>n	94	**94**	3,89	31	>600,00	16	>600,00	33	>600,00	42	>600,00
rand-d<n	672	561	22,32	550	37,98	**592**	**19,70**	589	22,62	582	22,41

TABLE 3.3 – Resultats sur des instances satisfiables : nombre d'instances résolues et temps médian en secondes

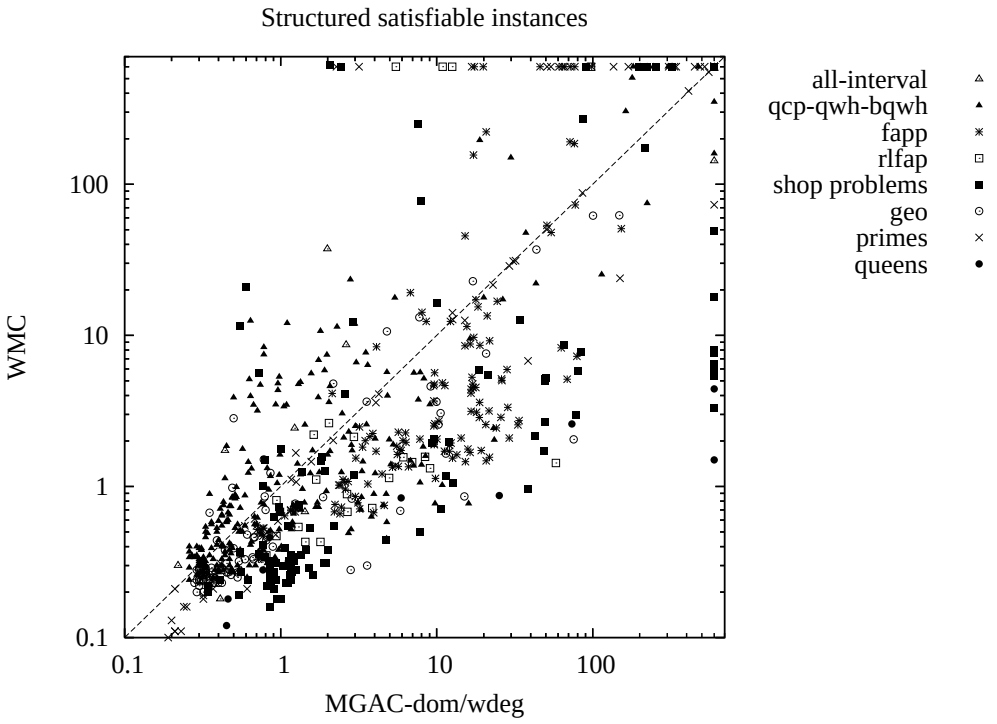


FIGURE 3.9 – Comparaison de WMC et MGAC sur des instances structurées (instances binaires uniquement)

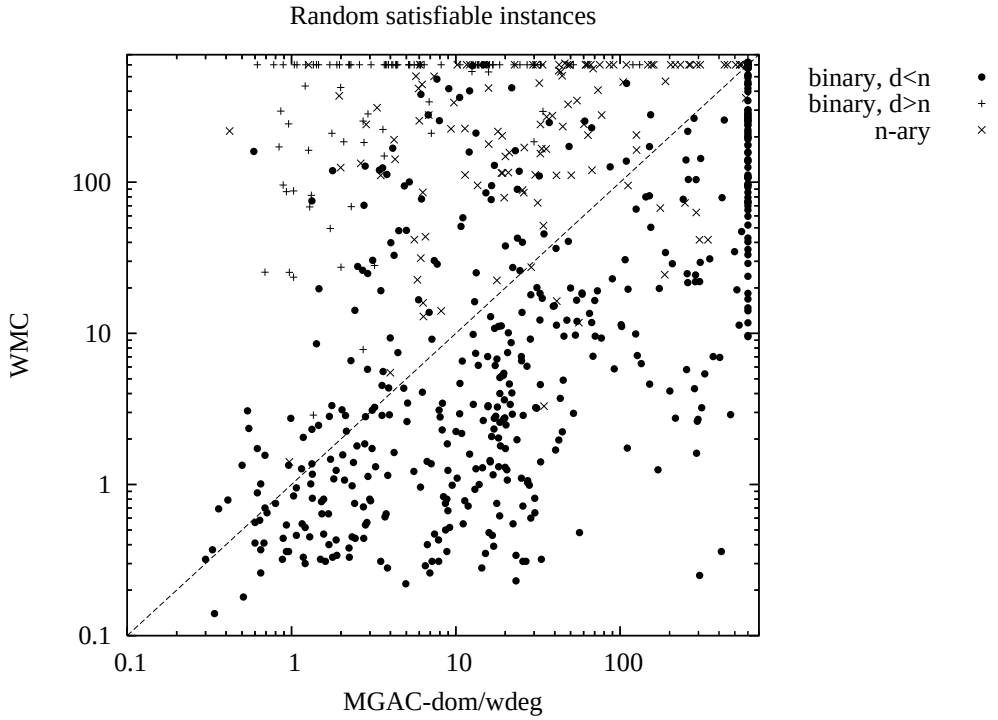


FIGURE 3.10 – Comparaison de WMC et MGAC sur des instances aléatoires

Les résultats obtenus sont reportés dans la table 3.3. Pour chaque algorithme, la colonne *rés* indique combien de problèmes ont pu être résolus en moins de 600 secondes. Les algorithmes Tabu et MCRW sont intéressants dès lors que leurs paramètres correspondent au type de problème (notamment ici les problèmes aléatoires ou les problèmes qwh/qcp). L'algorithme hybride parvient à combiner la puissance de MGAC et de WMC en réussissant à résoudre globalement le plus grand nombre de problèmes structurés.

La figure 3.9 représente le temps utilisé pour résoudre une instance avec un algorithme par rapport à un autre. Les points proches de la diagonale correspondent à des instances résolues de manière similaire par les deux algorithmes. Les points sous la diagonale sont des instances pour lesquelles l'algorithme représentées en ordonnées (ici WMC) est plus efficace. On projette sur les axes $x = 600$ ou $y = 600$ les instances qui n'ont pas été résolues par l'un ou l'autre des algorithmes. Cette figure met en évidence l'intérêt d'utiliser WMC par rapport à MGAC pour certaines classes spécifiques d'instances aléatoires, en particulier les problèmes d'atelier (*shop*), de n dames (*queens*) ou d'allocation de fréquences (*rlfap*).

La figure 3.10 confirme simplement les résultats de la table 3.3 concernant les problèmes aléatoires : WMC est plus efficace sur les problèmes binaires tels que $d < n$.

La table 3.4 montre le comportement de MGAC et de notre prouveur hybride sur une série de problèmes industriels (RLFAP, Open-Shop) et académiques incohérents. Dans **tous** les cas, le nombre global d'affectations pour MGAC est beaucoup plus faible en coopération avec WMC que seul, ce qui prouve l'efficacité des techniques d'apprentissage proposées. Dans la plupart des cas, le temps CPU global est également meilleur. Parfois, en particulier sur les instances faciles ou homogènes, trop de temps est perdu lors de la recherche locale (sur *scen11-f8*, *os-95-5* ou *os-95-8* par exemple) par rapport au temps gagné sur MGAC.

instance	MGAC-$dom/wdeg$ seul			hybrid					
				WMC		MGAC-$dom/wdeg$			$total\ CPU$
	$runs$	$assgns$	CPU	$runs$	CPU	$runs$	$assgns$	CPU	
scen11-f8	1	13 596	**107,9**	1	126,4	1	**470**	19,1	145,5
scen11-f7	2	38 692	268,4	1	124,8	1	**2 924**	31,9	**156,7**
scen11-f6	2	40 323	293,5	1	129	1	**6 894**	62,1	**191,1**
scen11-f5	3	68 307	713,3	2	258,3	2	**47 692**	378,6	**634,3**
os-5-95-2	7	79 599	105,8	1	20,8	1	**3 148**	10,7	**31,5**
os-5-95-5	4	30 262	**52,1**	2	39,5	2	**11 372**	29,2	68,7
os-5-95-8	4	24 137	**55,5**	2	47,0	2	**12 543**	30,2	77,2
qK-50-5-add	6	12 652	328,0	2	188,0	2	**2 495**	79,3	**267,3**
qK-50-5-mul	7	13 482	**452,3**	2	398,0	2	**2 495**	73,0	525,3
qK-80-5-add	> 10	$> 38\,000$	$> 6\,000{,}0$	3	854,9	3	**6 395**	603,1	**1 558,0**
qK-80-5-mul	> 9	$> 25\,000$	$> 6\,000{,}0$	2	1 666,2	2	**6 395**	971,6	**2 637,8**

TABLE 3.4 – Résultats sur plusieurs problèmes difficiles

3.2.4 En résumé

Dans cette section, nous avons introduit une approche hybride entre la recherche systématique et la recherche locale, respectivement représentées par les algorithmes MGAC et WMC. Nous proposons un système d'apprentissage pour améliorer les performances des deux algorithmes. Des expérimentations avec nos implantations de MGAC, MCRW, recherche Tabu et WMC, basées sur les derniers résultats publiés, et sur de nombreux benchmarks disponibles ont montré la robustesse de notre approche, avec de bons résultats en particulier sur les instances structurées, bien que leur taille et leur nature soient très variées.

Tout en confirmant les résultats obtenus par [MAZURE *et al.* 1998,] sur le problème SAT et [EISENBERG ET FALTINGS 2003] sur les problèmes de coloration de graphes, nous améliorons les relations entre les deux algorithmes en liant de manière très naturelle les heuristiques utilisées par chaque algorithme grâce notamment à l'utilisation de l'heuristique $dom/wdeg$. Ceci nous a mené à une stratégie de recherche robuste où l'impact de la recherche locale et systématique peut être finement paramétrée.

Ces travaux restent préliminaires : d'autres informations pourraient également être extraites d'une exécution d'un algorithme, et un paramétrage adaptatif des algorithmes pourrait améliorer encore la robustesse de la recherche. La possibilité de lancer plusieurs algorithmes en parallèle doit également être prise en considération.

Un avantage non négligable des algorithmes hybrides basés sur des algorithmes très connus et étudiés, comme MGAC et la recherche locale existe : en effet, à chaque progrès significatif obtenu sur l'un ou l'autre algorithme, c'est l'ensemble de l'algorithme hybride qui bénéficie des meilleurs résultats de chacune des méthodes de recherche.

3.3 Conclusion

Après nous être attachés, dans le précédent chapitre, à améliorer les techniques d'inférences génériques utilisées pour la résolution générale des CSP, nous avons ici cherché à améliorer les méthodes de recherche permettant de résoudre en pratique un grand nombre d'instances de problèmes NP-complets.

Les principales améliorations pouvant être apportées aux algorithmes de recherche sont de deux ordres :

– Tout d'abord, la flexibilité des algorithmes de recherche nous semble un point primordial. Les heuristiques les plus évoluées restent des approximations susceptibles de faire des erreurs, et il nous semble important de tout faire pour que ces erreurs aient un impact limité sur les performances de

la recherche. Le développement d'algorithmes de branchement binaires, de stratégies de redémarrages ou l'hybridation avec les algorithmes de recherche locale sont autant de voies ouvertes vers le développement d'algorithmes capables de résoudre un CSP de la manière la plus intelligente possible.

– Les heuristiques, qui permettent aux algorithmes systématiques d'ordonner intelligemment la recherche de manière à rendre chaque hypothèse la plus cohérente possible, permettent de réduire grandement le nombre de possibilités à explorer. On va chercher à les améliorer le plus possible, par exemple d'un point de vue syntaxique en cherchant à *informer* le plus possible les heuristiques. En prenant en compte le plus possible d'informations disponibles sur le problème, on réussit à mieux déterminer les hypothèses les plus intéressantes. C'est dans cette optique que nous avons développé les heuristiques de choix de valeurs décrites dans la section 3.1.

Aujourd'hui, les techniques d'apprentissage nous semblent susceptibles de faire progresser l'efficacité des prouveurs sur les deux plans. D'une part, en enregistrant des nogoods, il est possible de mettre de côté toute une partie de l'arbre de recherche sans perdre le travail effectué. D'autre part, l'accumulation de statistiques au cours de la recherche, telle la pondération de contraintes opérées par WMC et $dom/wdeg$, ou des approches similaires à celles introduites par [MAZURE *et al.* 1998,], permettent aux heuristiques d'apprendre de leurs erreurs et ainsi de devenir progressivement de plus en plus efficaces au cours de la recherche.

Chapitre 4

CSP4J : une bibliothèque de résolution de CSP « boîte noire » pour Java

Ce chapitre présente l'aboutissement des travaux décrits dans ce document. Toutes les techniques et algorithmes décrits précédemment ont été mis en place dans un prouveur de CSP conçu comme une API « boîte noire » (*black box*) de résolution de problèmes CSP et Max-CSP pour le langage Java. Les systèmes de programmation par contraintes actuellement disponibles sont généralement conçus comme des « boîtes de verre » (*glass box*), peuvent être paramétrés très finement et sont pratiquement des langages de programmation à part entière. Cependant, les prouveurs travaillant sur le problème SAT suivent généralement le précepte de boîte noire, influencés par la prépondérance des compétitions dans la communauté SAT. L'idée est de donner le moins de travail possible à l'utilisateur, et d'effectuer automatiquement (« intelligemment ») le plus possible de paramétrages. L'expertise demandée à l'utilisateur, en particulier dans le domaine de la programmation, doit être aussi limitée que possible. Dans le meilleur des cas, il n'aura qu'à définir son problème, le soumettre au prouveur et obtenir une solution sans avoir préalablement choisi aucune méthode ni paramètre [PUGET 2004]. Le développement récent de compétitions de prouveurs de CSP [VAN DONGEN *et al.* 2006A,] et l'apparition de nouveaux prouveurs inspirés des prouveurs SAT [GENT *et al.* 2006A,] sont autant de symboles de l'intérêt de la communauté scientifique pour cette nouvelle optique.

CSP4J est en développement depuis 2005 et a rapidement acquis une certaine maturité, lui ayant permis de participer aux premières compétitions avec des résultats corrects. Étant donné l'optique « boîte noire » dans lequel CSP4J a été développé, nous essayons de demander le moins d'informations possible à l'utilisateur. D'autre part, nous ne nous sommes pas focalisés sur l'implantation et le développement d'un grand nombre de contraintes globales, bien que le modèle objet retenu soit suffisamment flexible pour en ajouter à l'infini. À l'heure actuelle, CSP4J ne propose que la célèbre contrainte globale « all-different », équipée d'un algorithme de propagation spécifique simple.

CSP4J implante tous les « moteurs de recherche » décrits dans le chapitre 3 de ce document.

- **MGAC**, est un prouveur complet basé sur l'algorithme bien connu MGAC [SABIN ET FREUDER 1994] utilisant l'heuristique de choix de variables $dom/wdeg$ [BOUSSEMART *et al.* 2004A,] et l'heuristique de choix de valeurs $max\text{-}inverse$ statique. MGAC est le moteur le plus polyvalent et est proposé par défaut.
- **MCRW**, est un prouveur incomplet basé sur l'algorithme dit *Min-Conflicts Hill-Climbing with Random Walks* [MINTON *et al.* 1992,]. Ce moteur peut être utilisé pour résoudre les problèmes de type Max-CSP de manière incomplète dans un environnement « anytime » : le moteur peut être interrompu à tout moment, et la meilleure solution trouvée obtenue. Cette technique peut être très intéressante en particulier pour obtenir des solutions « faisables » en un temps de calcul très faible.

– **Tabu**, est une alternative à MCRW et dispose de caractéristiques similaires. Il effectue une recherche Tabu [GALINIER ET HAO 1997] souvent considérée comme plus robuste que le *Hill-Climbing*.
– **WMC**, est un autre prouveur incomplet basé sur la méthode *Breakout* [MORRIS 1993]. Les caractéristiques sont similaires à MCRW et Tabu, mais ne fonctionne pas vraiment sur les problèmes Max-CSP, et reste plutôt destiné à l'obtention rapide d'une solution faisable.
– **Combo**, est un prouveur complet basé sur une hybridation entre MGAC et WMC (cf section 3.2).

Lors de la résolution d'un CSP, la consistance duale conservative forte est appliquée au problème lors d'une phase de prétraitement.

Notre API est disponible sous la licence LGPL [STALLMAN 1999]. Pour montrer l'intérêt de la bibliothèque, nous avons développé plusieurs applications de démonstration, toutes disponibles sous la licence GPL [STALLMAN 1991]. L'une de ces applications consiste en un compilateur d'instances au format XCSP 2.0. C'est cette application qui a participé aux compétitions de prouveurs de CSP [VAN DONGEN *et al.* 2006A,]. Cette application implante une contrainte « prédicat » permettant de compiler les contraintes en intention ou en extension définies par le format XCSP 2.0.

Parmi les autres applications, nous proposons :
– un générateur et prouveur de problèmes aléatoires, utile pour évaluer (*benchmark*) les algorithmes et les machines,
– un extracteur de noyaux minimaux instatisfiables (MUC pour Minimal Unsatisfiable Core), dont l'objectif est d'extraire un ensemble de variables et de contraintes minimal insatisfiable à partir d'un CSP insatisfiable plus grand,
– un prouveur de problèmes Open-Shop, capable de rechercher des solutions faisables et éventuellement la solution optimale de problèmes d'Open Shop,
– une application capable de résoudre un Sudoku de taille quelconque.

4.1 Résoudre un CSP dans une boîte noire

Les compétitions de prouveurs montrent que l'utilisation de prouveurs non paramétrés pour résoudre un très grand nombre de problèmes différents est possible. Dans le cadre des compétitions de prouveurs CSP, les problèmes sont définis par le format très général XCSP 2.0.

Afin d'être capable de résoudre n'importe quel type de problème, CSP4J a été conçu avec deux règles d'or en tête : généricité et flexibilité. Le choix du langage orienté objet Java 5 a été fait dans l'optique d'une meilleur flexibilité. La conception orientée objets de CSP4J permet de modéliser les problèmes sous forme d'objets.

Quelques classes et interfaces forment le cœur de CSP4J, comme décrit par le diagramme UML représenté figure 4.1 : les classes $Problem$, $Variable$ et $Constraint$ définissent une instance de CSP. L'interface $Solver$ est « implémentée » par chacun des moteurs de recherche disponibles.

La classe $Variable$ **:** Elle peut être utilisée directement via son constructeur par défaut : le paramètre $domain$ contient simplement le domaine de la variable (c'est à dire l'ensemble des valeurs que la variable peut prendre) sous la forme d'un tableau d'entiers.

La classe $Constraint$ **:** Cette classe abstraite doit être spécialisée pour définir une contrainte spécifique à un problème. En particulier, il faut surcharger la méthode abstraite $check()$ de manière à ce qu'elle retourne **vrai** ou **faux** en fonction de la valeur du multiplet courant. On peut accéder aux valeurs du multiplet grâce à la méthode $getValue(int\ variablePosition)$, $variablePosition$ correspondant à la position de la variable dans la contrainte, comme définie par le $scope$ dans le constructeur. Le listing

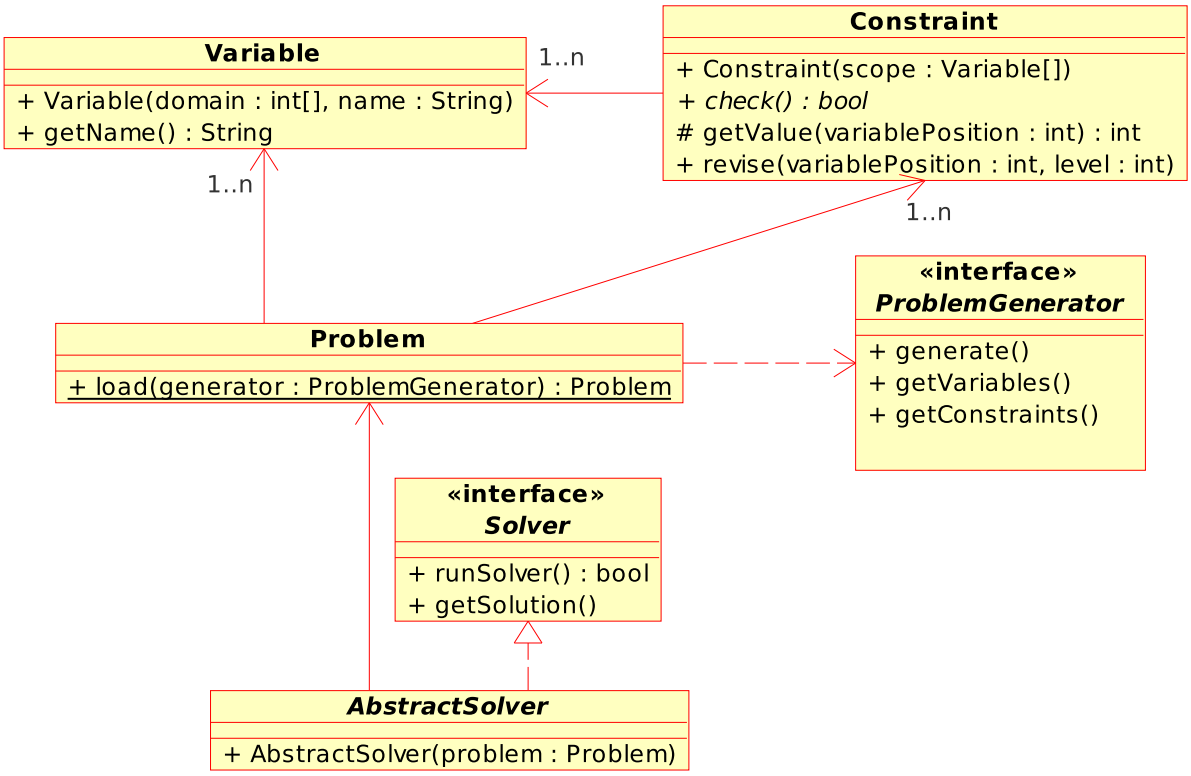


FIGURE 4.1 – Principaux éléments de CSP4J du point de vue de l'utilisateur

```
public final class DTPConstraint extends Constraint {

  final private int duration0;
  final private int duration1;

  public DTPConstraint(final Variable[] scope,
    final int duration0, final int duration1) {
    super(scope);
    this.duration0 = duration0;
    this.duration1 = duration1;
  }

  @Override
  public boolean check() {
    final int value0 = getValue(0);
    final int value1 = getValue(1);

    return value0 + duration0 < value1 || value1 + duration1 < value0;
  }

}
```

Listing 4.1 – La contrainte temporelle disjonctive (DT)

```
final Predicate predicate = new Predicate();
predicate.setExpression("(X0 + X1 < X2) || (X2 + X3 < X0)");
predicate.setParameters("int X0 int X1 int X2 int X3");

final PredicateConstraint dtpConstraint =
  new PredicateConstraint(scope, predicate);
dtpConstraint.setParameters(scope[0].getName() + " " + duration0
  + " " + scope[1].getName() + " " + duration1);
try {
  dtpConstraint.compileParameters();
} catch (FailedGenerationException e) {
  System.err.println("Failed to compile constraint");
  System.exit(1);
}
```

Listing 4.2 – Définir une contrainte DT par des prédicats

4.1 donne un exemple simple de contrainte : la contrainte temporelle disjonctive, qui est au centre des problèmes d'atelier. On peut alternativement utiliser $PredicateConstraint$ pour définir une contrainte, comme sur le listing 4.2. Cependant, la classe $PredicateConstraint$ fait partie de l'application Competitor de CSP4J, qui est disponible uniquement sous licence GPL.

Il est également possible, si nécessaire, de surcharger la méthode $revise(int\ variablePosition, int\ level)$ afin de développer des propagateurs spécifiques à une variable. Sinon, la révision sera effectuée par un algorithme générique de consistance d'arc.

La classe $Problem$ **:** Elle permet de représenter un CSP. On va utiliser l'interface $ProblemGenerator$ pour définir une classe permettant de générer le problème à résoudre. Pour définir un problème, il faut créer une classe « implémentant » l'interface $ProblemGenerator$. Une instance du problème sera chargée par un appel à la méthode statique $Problem.load(ProblemGenerator)$. L'interface $ProblemGenerator$ ne définit que trois méthodes :

- $generate()$: cette méthode est appelée lors du chargement du problème (par la méthode $Problem.load()$). Elle peut être utilisée pour créer les contraintes et les variables.
- $Collection\ \langle Variable\rangle\ getVariables()$: cette méthode doit renvoyer l'ensemble des variables du problème.
- $Collection\ \langle Constraint\rangle\ getConstraints()$: cette méthode doit renvoyer l'ensemble des contraintes du problème.

L'interface $Solver$ **et la classe** $AbstractSolver$ **:** Ces deux objets sont au cœur des moteurs pour CSP4J. Tous les moteurs de CSP4J sont des classes qui spécialisent $AbstractSolver$. La méthode $runSolver()$ lance la résolution et renvoie **vrai** si le problème est satisfiable ou **faux** sinon. La méthode $getSolution()$ permet alors de renvoyer la dernière solution trouvée (la meilleure solution trouvée à un instant donné pour un problème Max-CSP). Pour utiliser CSP4J comme un prouveur Max-CSP incomplet, il faut exécuter $runSolver()$ depuis un thread afin de contrôler son exécution.

Pour illustrer comment CSP4J peut être utilisé dans une application Java, le listing 4.3 définit le problèmes des Pigeons décrit dans la section 1.1.2, en utilisant une clique de contraintes de différence ($\neq$), à l'aide de la contrainte $PredicateConstraint$. Une fois le problème défini et chargé, le processus de résolution peut être exécuté en quelques lignes de code (listing 4.4).

```
public class Pigeons implements ProblemGenerator {
  final private int size;
  final private List<Variable> variables;
  final private Collection<Constraint> constraints;
  final private Predicate predicate;

  public Pigeons(int size) {
    this.size = size;
    variables = new ArrayList<Variable>(size);
    constraints = new ArrayList<Constraint>();
    predicate = new Predicate();
    predicate.setExpression("X0 != X1");
    predicate.setParameters("int X0 int X1");
  }

  public void generate() throws FailedGenerationException {
    final int[] domain = new int[size - 1];
    for (int i = size - 1; --i >= 0;) {
      domain[i] = i;
    }
    for (int i = size; --i >= 0;) {
      variables.add(new Variable(domain, "V" + i));
    }
    for (int i = size; --i >= 0;) {
      for (int j = size; --j >= i + 1;) {
        constraints.add(diff(variables.get(i), variables.get(j)));
      }
    }
  }

  private Constraint diff(final Variable var1,
    final Variable var2) throws FailedGenerationException {
    PredicateConstraint constraint = new PredicateConstraint(
        new Variable[] { var1, var2 }, predicate);
    constraint.setParameters(var1.getName() + " " + var2.getName());
    constraint.compileParameters();
    return constraint;
  }

  public Collection<Variable> getVariables() {
    return variables;
  }

  public Collection<Constraint> getConstraints() {
    return constraints;
  }
}
```

Listing 4.3 – Le problème des Pigeons

```
public static void main() throws FailedGenerationException, IOException {
  final Problem problem = Problem.load(10);
  final Solver solver = new MGAC(problem);
  final boolean result = solver.runSolver();
  System.out.println(result);
  if (result) {
    System.out.println(solver.getSolution());
  }
}
```

Listing 4.4 – Résoudre le problème des 10 pigeons

Les classes $XMLGenerator$ et $PredicateConstraint$: Afin de participer aux compétitions internationales de prouveurs CSP [VAN DONGEN *et al.* 2006A,], et pour avoir accès à la grande bibliothèque de problèmes mise en place à l'occasion de ces compétitions [LECOUTRE 2006], la possibilité de charger des problèmes au format XCSP 2.0 élaboré à l'occasion des compétitions [VAN DONGEN *et al.* 2006B,] a été réalisée. La classe $XMLGenerator$, qui implémente $ProblemGenerator$, permet de charger un problème, en créant toutes les variables et les contraintes définies par un fichier xml au format XCSP 2.0. La seule ligne de commande nécessaire pour obtenir un problème à partir d'un fichier .xml est alors par exemple :

```
Problem prb = Problem.load(new XMLGenerator("instances/probleme.xml")) ;
```

$XMLGenerator$ va créer des contraintes, soit en intention ($PredicateConstraint$) soit en extension ($ExtensionConstraint$, fourni avec CSP4J). Si il s'agit d'une contrainte binaire en extension, elle sera enregistrée en utilisant les structures de données présentée dans la section 2.1. Les prédicats définis pour les contraintes en intention sont compilés grâce à la librairie JEL [METLOV 2006][7]. Pour des raisons d'efficacité, et pour permettre l'utilisation de certaines techniques décrites précédemment, comme la consistance duale conservative ou encore les opérations bit-à-bit, la méthode $Problem.load()$ tente alors de convertir un maximum de contraintes en extension, jusqu'à l'éventuel épuisement de la mémoire allouée à la machine virtuelle.

4.2 Une application : résoudre un problème d'Open Shop

Le problème d'Open Shop, qui généralise les problèmes d'ordonnancement tout en restant très simple et très difficile à résoudre de manière optimale, est particulièrement intéressant pour évaluer les performances d'un système de programmation par contraintes. Le problème est décrit dans la section 1.1.2. Nous avons choisi de résoudre des problèmes d'Open Shop définis suivant le format adopté par Guéret et Prins pour leur article [GUÉRET ET PRINS 1999]. Christelle Guéret met ainsi à disposition 80 problèmes types de taille 3×3 à 10×10. Chaque fichier contient une ligne de commentaires identifiée par le caractère « ! », deux lignes indiquant respectivement le nombre de machines m et le nombre de travaux j, et enfin une matrice $m \times j$, chacune des j lignes contenant les m durées des tâches, séparées par un espace (voir un exemple listing 4.5).

La première étape sera de lire ce fichier et de modéliser le problème en conséquence sous forme d'un problème de décision : est-il possible de trouver un ordonnancement des $m \times j$ tâches tel que toutes les tâches soient réalisées au bout d'un temps T et que les contraintes de ressources soient satisfaites ? Nous

7. JEL est disponible sous licence GNU GPL et est redistribué avec les applications test de CSP4J. La licence GPL nous interdit de distribuer $PredicateConstraint$ sous licence LGPL comme la librairie CSP4J elle-même.

```
! FILE GP05-01.TXT. 1ST LINE=M, 2ND=N, OTHERS=MATRIX P MXN
5
5
289 26 35 649 1
1 316 366 254 63
392 1 570 34 3
21 1 1 45 932
297 656 28 18 1
```

Listing 4.5 – Exemple de fichier décrivant un problème d'Open Shop de taille 5×5

développerons ensuite un algorithme capable de déterminer l'ordonnancement optimal en temps T_{OPT} par approximations successives.

4.2.1 Modélisation

Le problème d'Open Shop est construit entièrement à partir de contraintes disjonctives de type $X_{i,k} - X_{i,j} > d_{i,j} \vee X_{i,j} - X_{i,k} > d_{i,k}$ (cf section 1.1.2). La contrainte $dtpConstraint$ décrite par le listing 4.1 implante une telle contrainte. Le constructeur de la contrainte prend comme paramètres un tableau constitué des deux variables $\{X_{i,j}, X_{i,k}\}$ représentant les temps de départ des deux tâches $t_{i,j}$ et $t_{i,k}$, ainsi que leur durée via deux paramètres constants $d_{i,j}$ et $d_{i,k}$. La classe $dtpConstraint$ ne définit pas de fonction $revise$ spécifique et le problème sera donc résolu par un algorithme MAC-3^{bit+rm} classique.

L'interprétation du fichier ne présente pas de difficulté particulière. Nous définissons une classe $OpenShopGenerator$ qui « implémente » l'interface $ProblemGenerator$. La construction du générateur ne nécessite qu'un paramètre : le chemin d'accès au fichier définissant le problème. Le constructeur lit le fichier et crée un tableau d'entiers à deux dimensions $m \times j$ contenant la durée $d_{i,j}$ de chaque tâche.

La classe $OpenShopGenerator$ contient les attributs suivants :

- Le tableau d'entiers à deux dimensions $d[][]$ précité
- Un tableau de variables à deux dimensions représentant le temps de démarrage de chaque tâche
- Les tailles m et j du problème
- Le temps limite T. Une méthode $setUB$ permet de fixer cette limite manuellement. Par défaut, elle est fixée à la limite haute calculée par $getUB$
- Une collection de contraintes qui contiendra les contraintes du problème

Deux méthodes getLB et getUB permettent respectivement de calculer une estimation basse T_{LB} et haute T_{UB} de T_{OPT} (voir ci-dessous).

La méthode $generate$ est « implémentée ». Elle crée toutes les variables. Le domaine de chaque variable $X_{i,j}$ est un ensemble continu d'entiers compris entre 0 et $T - d_{i,j}$ inclus. On crée ensuite simplement les cliques de contraintes « verticales » puis « horizontales » (cf figure 1.9 page 11) par des boucles « pour » imbriquées.

La méthode $getVariables$ renvoie le tableau de variables linéarisé en une simple collection, et la méthode $getConstraints$ renvoie simplement la collection de contraintes générée par la méthode $generate$.

La résolution du problème de décision pour un temps T peut alors être exécutée de manière similaire au problème de pigeons du listing 4.4. Il arrive que la solution trouvée le soit dans un temps inférieur à T (en particulier si T est largement supérieur à T_{OPT}). La méthode $evaluate$ renvoie le temps global d'une solution renvoyée par un prouveur de CSP4J.

Algorithme 38 : findOpt(file : Chemin d'accès) : Entier

```
1  openShop ← new OpenShopGenerator(file)
2  lb ← openShop.getLB()
3  ub ← openShop.getUB()
4  tant que ub > lb faire
5      test ← (ub + lb)/2
6      openShop.setUB(test)
7      solver ← new MGAC(Problem.load(openShop))
8      result ← solver.runSolver()
9      if result then
10         ub ← openShop.evaluate(solver.getSolution())
11     else
12         lb ← test + 1
13 retourner ub
```

4.2.2 Déterminer un ordonnancement optimal

Beaucoup de travaux de recherche sur les problèmes d'ordonnancement, en particulier en recherche opérationnelle, se focalisent sur le calcul d'une limite basse T_{LB} et une limite haute T_{UB} de la durée de l'ordonnancement optimal T_{OPT}. Si $T_{LB} = T_{UB}$, alors $T_{OPT} = T_{LB} = T_{UB}$. Étant donné qu'il s'agit d'une simple application de démonstration, nous nous dommes contentés d'algorithmes simplistes et peu précis pour calculer des bornes des problèmes d'Open Shop. $getLB$ renvoie le maximum de la somme de chaque ligne et de chaque colonne de la matrice des durées (les problèmes de Guéret et Prins comme ceux du listing 4.5 sont générés de telle sorte que la somme de chaque ligne et de chaque colonne vaut 1 000). Nous obtenons une limite haute très grossière, renvoyée par $getUB$, en additionnant le maximum de la somme de chaque ligne avec le maximum de la somme de chaque colonne (soit 2 000 pour les problèmes de Guéret et Prins). Notez que plus T est proche de T_{OPT}, à gauche comme à droite de la limite, plus le problème de décision est difficile (il y a un phénomène de seuil).

On cherche alors T_{OPT} par approximations successives dichotomiques comme présenté par l'algorithme 38. On peut noter qu'il est possible d'interrompre l'algorithme à tout moment et d'obtenir les meilleures bornes T_{LB} et T_{UB} prouvées au moment de l'interruption.

Les tables 4.1 montrent des exemples de progression de l'algorithme ainsi que les temps d'exécution sur une machine virtuelle Sun Java 6 fonctionnant sur un processeur AMD 64 bits cadencé à 2 GHz et équipé de 1 Gio de RAM. Chaque ligne correspond à un tour de boucle de l'algorithme 38. Les colonnes lb et ub indiquent les bornes T_{LB} et T_{UB} connues lors de l'itération courante, et la colonne T la limite qui sera testée par cette itération. La colonne « chg » indique le « temps de préparation » de l'instance, en particulier la conversion des contraintes en extension afin d'exploiter les opérations bit-à-bit et le calcul du nombre de supports de chaque valeur pour les heuristiques et les tests de détection de révisions inutiles. Étant donné la très grande taille des problèmes ($d = T$, $e = j \times C_m^2 + m \times C_j^2$) et la complexité des opérations de chargement en $O(ed^2)$, ce temps, bien que polynomial, n'est en général pas négligeable. Pour la même raison, la Consistance Dual Conservative n'a pas été utilisée pour ces tests. La colonne « rch » indique le temps de la recherche proprement dite par MAC. La colonne « sol » indique le temps ($\leq T$) correspondant à la solution trouvée par l'algorithme. Les temps de recherche pour les plus gros problèmes peuvent être extrêmement importants (plusieurs heures pour certains 10×10). Pour ces problèmes, on peut adopter une stratégie alternative en remplaçant la ligne 5 de l'algorithme 38 par $test \leftarrow ub - 1$. Le nombre d'appels à runSolver() sera bien plus important, mais les problèmes à résoudre

GP-06-01

	lb	ub	T	chr	rch	sol
1.	1 000	2 000	1 500	25,4	0,5	1 328
2.	1 000	1 328	1 164	14,1	4,1	UNSAT
3.	1 165	1 328	1 246	16,1	6,4	UNSAT
4.	1 247	1 328	1 287	17,6	1,4	1 282
5.	1 247	1 282	1 264	16,5	2,7	1 264
6.	1 247	1 264	1 255	16,0	6,9	UNSAT
7.	1 256	1 264	1 260	16,4	7,0	UNSAT
8.	1 261	1 264	1 262	16,4	6,8	UNSAT
9.	1 263	1 264	1 263	17,0	7,1	UNSAT
			total :	155,5	43,3	1 264

GP-07-01

	lb	ub	T	chr	rch	sol
1.	1 000	2 000	1 500	36,8	0,5	1 411
2.	1 000	1 411	1 205	22,8	0,3	1 195
3.	1 000	1 195	1 097	18,5	2,6	UNSAT
4.	1 098	1 195	1 146	20,6	4,2	UNSAT
5.	1 147	1 195	1 171	21,4	0,3	1 165
6.	1 147	1 165	1 156	20,8	5,2	UNSAT
7.	1 157	1 165	1 161	20,9	0,3	1 160
8.	1 157	1 160	1 158	20,8	5,6	UNSAT
9.	1 159	1 160	1 159	20,7	0,3	1 159
			total :	203,3	19,2	1 159

GP-08-01

	lb	ub	T	chg	rch	sol
1.	1 000	2 000	1 500	61,6	0,7	1 419
2.	1 000	1 419	1 209	38,9	0,5	1 180
3.	1 000	1 180	1 090	30,9	5,8	UNSAT
4.	1 091	1 180	1 135	33,5	8,0	1 134
5.	1 091	1 134	1 112	32,5	6,7	UNSAT
6.	1 113	1 134	1 123	33,2	7,3	UNSAT
7.	1 124	1 134	1 129	33,4	14,5	UNSAT
8.	1 130	1 134	1 132	33,8	8,4	1 132
9.	1 130	1 132	1 131	33,5	13,3	1 131
10.	1 130	1 131	1 130	33,7	12,9	1 130
			total :	365,0	78,1	1 130

GP-09-01

	lb	ub	T	chg	rch	sol
1.	1 000	2 000	1 500	84,1	0,9	1 447
2.	1 000	1 447	1 223	53,2	0,6	1 220
3.	1 000	1 220	1 110	44,2	26,9	UNSAT
4.	1 111	1 220	1 165	47,5	6,3	1 165
5.	1 111	1 165	1 138	46,4	33,4	1 137
6.	1 111	1 137	1 124	44,3	30,2	UNSAT
7.	1 125	1 137	1 131	46,0	14,5	1 130
8.	1 125	1 130	1 127	45,1	34,1	UNSAT
9.	1 128	1 130	1 129	45,3	13,7	1 129
10.	1 128	1 129	1 128	45,4	29,4	UNSAT
			total :	501,5	190,1	1 129

TABLE 4.1 – Exemples de progression de l'algorithme sur divers problèmes issus de la base de C. Guéret. chg et rch en secondes.

plus faciles. Si l'objectif n'est pas d'obtenir une solution optimale mais une solution « acceptable », on obtiendra également rapidement des bornes supérieures plus précises. L'utilisation de la 3B Consistance pourrait également améliorer grandement les performances sur ces problèmes, mais elle n'est pas encore proposée par CSP4J.

4.3 Extraction de noyaux insatisfiables

Dans les applications réelles, les utilisateurs sont rarement intéressés par la réponse « insatisfiable ». Plus que le problème de décision proprement dit, c'est l'obtention d'une solution satisfaisant toutes les contraintes (ou le plus de contraintes possible !) qui présente généralement un intérêt. Si le problème n'a pas de solution, c'est qu'il est *surcontraint*, c'est à dire que les contraintes imposées au problème sont trop fortes ou trop nombreuses. On va alors chercher à guider l'utilisateur vers une solution satisfaisante en lui indiquant quelles contraintes devraient être supprimées en priorité afin de rétablir la satisfiabilité du problème.

Un noyau minimalement insatisfiable $N = (\mathscr{X}_n, \mathscr{C}_n)$ d'un problème $P = (\mathscr{X}_p, \mathscr{C}_n)$ est un sous-problème de P ($N \subseteq P$, c'est à dire $\mathscr{X}_n \subseteq \mathscr{X}_p$ et $\mathscr{C}_n \subseteq \mathscr{C}_p$) tel que si on supprime une contrainte quelconque de N, le sous-problème résultant est satisfiable. On peut ainsi fournir à l'utilisateur une liste de contraintes parmi lesquelles au moins une doit être supprimée afin de rétablir la satisfiabilité du problème.

L'application d'extraction de noyaux minimalement insatisfiables (MUC pour Minimal Unsatisfiable Core) proposée par CSP4J se base sur l'algorithme $wcore$ [HEMERY *et al.* 2006,]. Pour développer cette application, nous avons besoin d'une fonctionnalité supplémentaire de CSP4J : la méthode $Constraint.isActive()$ renvoie **vrai** si la contrainte a amené à la suppression d'au moins une valeur du domaine d'une variable lors de la dernière résolution par un algorithme complet. L'extracteur de noyaux travaille sur des instances au format XCSP 2.0 et exploite la classe $XMLGenerator$ permettant de charger des problèmes sous ce format.

Algorithme 39 : active($P = (\mathscr{X}, \mathscr{C})$: CN) : CN

```
1 P'(X', C') ← P
2 pour chaque C ∈ C' faire
3     si ¬C.isActive() alors
4         C' ← C'\C
5 pour chaque X ∈ X faire
6     si ∄C ∈ C' | X ∈ vars(C) alors
7         X ← X\X
8 retourner (P')
```

Algorithme 40 : major($P = (\mathscr{X}, \mathscr{C})$: CN) : CN

```
1 P'(X', C') ← P
2 répéter
3     nbC ← |C'|
4     MGAC(P')
5     P' ← active(P')
6 jusqu'à nbC = |C'|
7 retourner P'
```

La méthode $wcore$ se base sur l'heuristique $dom/wdeg$ pour identifier les noyaux des problèmes, ayant pour hypothèse que les contraintes appartenant à un noyau auront tendance à avoir un poids plus important que les contraintes plus « faciles » à satisfaire, (et que les contraintes aux poids les plus faibles auront tendance à ne pas intervenir dans la preuve d'inconsistance d'un problème). Après une exécution, on identifie toutes les contraintes qui ne sont pas intervenues dans la preuve de l'inconsistance ($Constraint.isActive()$ retourne **faux**), et on les supprime du problème. Après plusieurs exécutions de cet algorithme, on obtient un majorant P' du noyau minimal insatisfiable N ($N \subseteq P' \subseteq P$). On élimine alors toutes les contraintes possibles par une méthode dichotomique. CSP4J affine légèrement l'approche de $wcore$ en travaillant dans un premier temps au niveau des variables (on supprime les variables ne formant pas un MUC).

On distingue donc trois étapes dans l'extraction d'un noyau insatisfiable :

1. Résolutions successives du problème par MGAC utilisant l'heuristique $dom/wdeg$, en supprimant après chaque résolution les contraintes inactives (algorithme 39), jusqu'à l'obtention d'un point fixe (algorithme 40, qui nécessite dans le pire des cas $O(e)$ appels à MGAC, mais le point fixe est obtenu rapidement en pratique, d'autant que les appels successifs à MGAC bénéficient de la pondération des contraintes des exécutions précédentes).
2. Obtention d'un MUC au niveau des variables par $O(n \log(n))$ appels à MGAC
3. Affinage du MUC au niveau des contraintes par $O(e \log(e))$ appels à MGAC

Il est important de noter que les poids des contraintes obtenus par $dom/wdeg$ lors de la recherche sont systématiquement maintenus d'un appel à l'autre à MGAC. De cette manière, l'heuristique de choix de valeurs réalise de bien meilleurs choix lors des appels successifs à MGAC. D'autre part, les poids donnent également une évaluation heuristique sur les contraintes et les variables, permettant d'identifier les plus susceptibles d'appartenir à un noyau insatisfiable.

Algorithme 41 : minimizeVar($P = (\mathscr{X}, \mathscr{C})$: CN) : CN

```
1  X_t ← ∅ ; X_p ← X
2  first ← 0 ; last ← |X_p| − 1
3  tant que X_p ≠ ∅ faire
4      tant que last − first > 1 faire
5          middle ← (last + first)/2
6          X'_p ← {X_p[0], ..., X_p[middle]}
7          si MGAC((X_t ∪ X'_p, {C ∈ C | vars(C) ⊆ X_t ∪ X'_p})) alors
8              first ← middle
9          sinon
10             last ← middle
11             X_p ← X'_p
12     X_t ← X_t ∪ X_p[last]
13     X_p ← X_p \ X_p[last]
14 retourner (X_t, {C ∈ C | vars(C) ⊆ X_t})
```

4.3.1 Un MUC au niveau des variables

Pour obtenir un MUC au niveau des variables, on élimine toutes les variables qui participent à l'insatisfiabilité du noyau. Si en supprimant une variable, le noyau est toujours insatisfiable, la variable était inutile. Si par contre le noyau devient satisfiable, la variable fait part intégrante du MUC. On appelle une telle variable une « variable de transition ». L'objectif est d'identifier toutes les variables de transition et de supprimer les autres variables. L'algorithme 41 ($minimizeVar$) réalise cette fonction.

On considère l'ensemble $\mathscr{X}_t$ contenant toutes les variables de transition. À l'initialisation de l'algorithme, cet ensemble est vide. Lorsque l'algorithme se termine, $\mathscr{X}_t$ contient toutes les variables du MUC. L'ensemble $\mathscr{X}_p$ contient les variables dont on ne connait pas encore la nature (variables de transition ou variables inutiles). Lorsque l'on veut minimiser le noyau d'un CN $P = (\mathscr{X}, \mathscr{C})$, on initialise $\mathscr{X}_p = \mathscr{X}$. L'algorithme se termine lorsque $\mathscr{X}_p = \emptyset$. L'ensemble $\mathscr{X}_p$ est ordonné, les variables sont triées en fonction de leur score d'après l'heuristique $dom/wdeg$, $\mathscr{X}_p[0]$ contenant la variable la plus contrainte ($dom/wdeg$ minimal), et donc la plus susceptible d'appartenir à un noyau.

L'approche dichotomique peut permettre d'identifier un grand nombre de variables inutiles simultanément, mais ne peut identifier de manière certaine une variable de transition qu'en $O(\log(n))$ appels à MGAC là où une approche classique ne nécessite qu'un seul appel à MGAC pour déterminer si une variable est inutile ou non. Dans le pire des cas, si toutes les variables appartiennent au noyau, l'algorithme 41 nécessite $O(n \log(n))$ appels à MGAC. Si une seule variable appartient au noyau, l'algorithme ne nécessite que $O(\log(n))$ appels à MGAC. Une approche classique testant chaque variable individuellement nécessite $\Theta(n)$ appels à MGAC. On constate donc que l'une ou l'autre des approches fonctionnera plus ou moins bien en fonction de la taille du noyau, qui n'est évidemment pas prévisible.

L'ensemble $\mathscr{X}_p$ étant trié en fonction de l'heuristique $dom/wdeg$, on suppose que les variables ayant le plus de chances d'appartenir au noyau sont situées en tête de l'ensemble, et les variables ayant le plus de chances d'être inutiles sont situées à la fin de l'ensemble. Lorsque l'on divise en deux cet ensemble (ligne 6 de l'algorithme 41), on est donc d'autant plus susceptibles d'éliminer un grand nombre de variables inutiles.

[HEMERY *et al.* 2006,] montrent empiriquement que l'approche dichotomique (au niveau des contraintes) semble particulièrement efficace sur des problèmes réels de grande taille comme les RLFAP ou

Algorithme 42 : minimizeCons($P = (\mathscr{X}, \mathscr{C})$: CN) : CN

```
1  C_t ← ∅ ; C_p ← C
2  first ← 0 ; last ← |C_p| − 1
3  tant que C_p ≠ ∅ faire
4      tant que last − first > 1 faire
5          middle ← (last + first)/2
6          C'_p ← {C_p[0], ..., C_p[middle]}
7          si ∃X ∈ X | ∄C ∈ C'_p | X ∈ vars(C)   ∨   MGAC((X, C_t ∪ C'_p)) alors
8              first ← middle
9          sinon
10             last ← middle
11             C_p ← C'_p
12     C_t ← C_t ∪ C_p[last]
13     C_p ← C_p \ C_p[last]
14 retourner (X, C_t)
```

FAPP (l'approche constructive évoquée dans l'article ne semble pas intéressante tant du point de vue théorique que pratique).

4.3.2 Raffiner le noyau au niveau des contraintes

Une fois obtenu un noyau minimalement insatisfiable au niveau des variables, on peut chercher si nécessaire (en fonction du besoin de l'utilisateur) à le raffiner au niveau des contraintes. Pour établir un noyau insatisfiable au niveau des contraintes, on définit la notion de « contrainte de transition ». L'algorithme $minimizeCons$ permettant d'identifier les contraintes de transition et de supprimer les contraintes inutiles est très similaire à l'algorithme $minimizeVar$ (algorithme 42).

Cependant, si on sait que le réseau sur lequel on travaille est minimalement insatisfiable au niveau des variables, on sait qu'au moins une contrainte parmi toutes les contraintes impliquant chaque variable est une contrainte de transition. Si avant un appel à MGAC on vient d'essayer de supprimer la dernière contrainte impliquant une variable donnée, le réseau est obligatoirement satisfiable et l'appel à MGAC est inutile. C'est l'objet de la première partie du test de la ligne 7. Si celui ci est **vrai**, on n'effectue bien sûr pas la deuxième partie du test.

Comme précédemment, $\mathscr{C}_p$ est trié en fonction des poids associés aux contraintes, ce qui permet d'éliminer plus rapidement les contraintes inutiles lorsque l'on coupe l'ensemble en deux à la ligne 6 de l'algorithme 42.

4.3.3 Expérimentations

On peut choisir de ne pas utiliser l'algorithme $minimizeVar$. Dans ce cas, on obtient l'algorithme $wcore$ original. La table 4.2 donne une idée de l'intérêt de minimiser d'abord le réseau au niveau des variables. Les expérimentations ont été réalisées sur une machine virtuelle Sun Java 5 pour Linux équipée d'un processeur Intel i686 fonctionnant à 3 GHz. L'écart relatif (e. r.) correspond au résultat du rapport $(cpu_{mC+mV} - cpu_{mC})/cpu_{mC}$. On indique un résultat médian pour chaque série de problèmes. L'écart relatif médian indiqué est la médiane de tous les écarts relatifs, et non pas l'écart relatif entre les médianes des temps cpu.

	minimizeCons				minimizeVar + minimizeCons				
instance	vars	cons	runs	cpu	vars	cons	runs	cpu	e. r.
Aléatoires (21 problèmes)									
$\langle 2; 30; 15; 306; 0{,}23\rangle - 2$	23	86	499	64,97	26	109	741	59,11	-9%
$\langle 2; 30; 15; 306; 0{,}23\rangle - 12$	28	110	676	94,73	28	108	664	76,16	-20%
$\langle 2; 30; 15; 306; 0{,}23\rangle - 15$	30	126	770	76,26	29	116	795	87,94	$+15\%$
$\langle 2; 30; 15; 306; 0{,}23\rangle - 20$	29	120	745	63,3	28	136	939	67,48	$+7\%$
...									
médiane	28	118	728	76,68	28	122	827	76,16	-9%
RLFAP (24 problèmes)									
scen8	14	23	124	3,33	5	9	33	0,24	-93%
scen11-f5	20	64	353	1635,5	14	79	471	3344,95	$+105\%$
scen11-f6	18	45	229	248,79	17	35	200	216,52	-13%
scen11-f7	18	43	229	93,06	16	39	245	100,86	$+8\%$
graph2-f25	14	43	223	14,5	13	39	205	10,95	-24%
graph9-f10	18	54	274	27,83	16	42	226	18,41	-34%
...									
médiane	8	16	71	4,12	8	15	67	2,45	-22%
Open Shop Taillard (20 problèmes)									
os-4-95-0	4	6	18	2,67	4	6	20	2,98	$+12\%$
os-4-95-1	4	6	14	26,05	4	6	13	23,78	-9%
os-4-95-2	4	6	13	7,15	4	6	13	6,41	-10%
...									
os-5-95-0	5	10	29	549,99	5	10	30	520,46	-5%
os-5-95-1	5	10	30	45,94	5	10	27	29,79	-35%
os-5-95-2	5	10	29	3393,61	5	10	28	2929,74	-14%
...									
médiane	5	9	24	60,72	5	10	27	48,49	-8%
FAPP (88 problèmes)									
fapp01-0200-3	7	7	35	16,07	7	7	25	4,46	-72%
fapp02-0250-1	15	17	80	13,12	3	3	18	0,95	-93%
fapp03-0300-6	3	3	20	11,76	3	3	16	0,67	-94%
fapp05-0350-8	7	7	41	87,96	11	11	50	42,02	-52%
fapp05-0350-9	11	11	42	58,33	11	11	50	42,29	-27%
fapp06-0500-3	5	5	35	23,06	3	3	19	1,13	-95%
fapp08-0700-4	15	15	90	69,66	15	15	83	4,01	-94%
fapp20-0420-8	10	9	53	12,74	5	5	19	0,89	-93%
...									
médiane	7	6	30	3,28	5	5	21	0,43	-88%
EHI (32 problèmes)									
ehi-90-315-1	19	33	153	6,14	20	35	193	6,05	-1%
ehi-90-315-13	21	34	159	6,1	21	34	193	6,01	-1%
ehi-90-315-15	21	36	170	19,79	21	39	237	19,94	$+1\%$
ehi-90-315-21	21	35	163	8,22	21	36	220	8,36	$+2\%$
ehi-90-315-24	20	33	148	5,99	20	33	180	6,01	0%
...									
médiane	21	33	150	2,61	20	33	193	2,66	$+2\%$

TABLE 4.2 – Extraction de noyaux avec et sans l'utilisation de minimizeVar

Les performances des deux approches peuvent varier grandement d'un problème à l'autre. Il arrive fréquemment que les noyaux extraits ne soient pas les mêmes avec l'une et l'autre approche. En effet, il arrive fréquemment qu'un problème contienne un très grand nombre de noyaux minimalement insatisfiables, et le noyau extrait dépendra fortement des heuristiques utilisées. Les expérimentations ne permettent pas de dégager clairement si une ou l'autre stratégie permettent d'extraire des noyaux plus ou moins grands.

Cependant, en utilisant $minimizeVar$, on améliore le temps d'extraction sur la plupart des problèmes testés. Le gain est particulièrement sensible sur les problèmes FAPP, avec un progrès de près d'un ordre de magnitude. Sur d'autres problèmes comme les EHI, la différence est indiscernable. Il est intéressant de constater que l'on obtient un gain même sur les problèmes aléatoires, où la pondération de contraintes n'a pas vraiment de signification et où les noyaux sont très proches du problème initial.

4.4 Conclusion

Malgré le manque de maturité de CSP4J, son comportement lors des deux premières Compétitions Interantionales de Prouveurs de CSP [VAN DONGEN *et al.* 2006A,] a été plutôt satisfaisant. Les moteurs de recherche MGAC et Combo ont participé lors de la deuxième compétition, face à de nombreux prouveurs concurrents très matures, y compris des prouveurs SAT ayant participé à de nombreuses compétitions SAT. CSP4J était toujours entre la 4e et la 8e place sur 12 à 16, en fonction des catégories dans lesquelles il a participé. CSP4J dispose de suffisamment de fonctionnalités pour participer à l'ensemble des catégories avec des résultats constants. Le moteur MGAC de CSP4J a résolu 2 188 problèmes, sur les 3 425 sélectionnés par le comité d'organisation, en moins de trente minutes (le moteur Combo en a résolu 2 173). Nous continuons à progresser au fur et à mesure des nouveaux développements, des diverses optimisations et des avancées scientifiques.

CSP4J est conçu comme une API pour Java 5, développé pour permettre à toute application Java d'exploiter un certaine nombre de techniques évoluées d'intelligence artificielle, et en particulier la résolution de CSP. CSP4J fonctionne comme une « boîte noire », c'est à dire qu'elle essaye de demander le moins de paramétrages possible à l'utilisateur. C'est l'esprit même des compétitions de prouveurs : un même prouveur doit être capable, sans paramétrage, de résoudre un très grand nombre de problèmes de natures très variées.

Cette section a présenté CSP4J du point de vue de l'utilisateur, en montrant deux applications potentielles de CSP4J : la résolution d'un problème combinatoire classique d'Open Shop, et l'extraction d'un noyau minimalement insatisfiable d'un problème quelconque.

Conclusion

Nous avons proposé dans cette thèse plusieurs techniques visant à résoudre en pratique le problème NP-complet de satisfaction de contraintes de manière totalement générique, c'est à dire indépendamment de la nature du problème à résoudre. Cette optique de résolution, habituellement plus familière dans la littérature associée à la résolution pratique du problème SAT que dans le cadre de la programmation par contraintes, est aussi appelée « résolution par boîtes noires ».

Nous avons distingué deux grands axes de techniques de résolution de CSP : l'inférence et la recherche. L'inférence ne vise pas à la résolution proprement dite du problème, mais exploite le plus possible les propriétés du réseau de contraintes pour détecter rapidement des valeurs ou des instanciations inconsistantes. Cela permet de réduire très substantiellement la taille de l'espace exploré par les algorithmes de recherche. L'algorithme de recherche à ce jour le plus performant, MGAC, mêle une forme puissante et efficace d'inférence, la consistance d'arc généralisée, et une recherche systématique guidée par des heuristiques génériques adaptatives comme $dom/wdeg$.

Nous avons contribué à l'amélioration des techniques d'inférence en plusieurs points, mais toujours en nous concentrant sur la propriété centrale qu'est la consistance d'arc. Dans un premier temps, nous avons proposé une optimisation simple de l'algorithme établissant la consistance d'arc le plus rapide connu, (G)AC-3^{rm}, dans le cas des contraintes binaires. En exploitant des structures basées sur des mots CPU et les opérations bit-à-bit de conjonction et disjonction, nous sommes parvenus à résoudre bien plus rapidement un grand nombre de problèmes. Nous avons dans un deuxième temps étudié le comportement de plusieurs algorithmes d'inférence connus (AC, Max-RPC, SAC) en limitant leur action aux bornes des domaines discrets. Nous avons obtenu des résultats théoriques intéressants et de bons comportements en pratique sur certaines classes de problèmes, et en particulier sur les problèmes d'ordonnancement. Finalement, nous avons proposé une alternative intéressante à la consistance de chemin (établie en prétraitement, c'est à dire avant d'effectuer une recherche), en proposant une définition équivalente que nous avons nommée la consistance duale. Cette définition nous a amené à concevoir des algorithmes, inspirés des algorithmes établissant la singleton consistance d'arc, capables d'établir la consistance de chemin de manière très efficace en pratique. Nous avons ensuite cherché à limiter les défauts bien connus de la consistance de chemin, et donc de la consistance duale : ces propriétés nécessitent généralement l'ajout d'un grand nombre de contraintes au problème pour être établies. Nous sommes parvenus à montrer que la variante conservative, c'est à dire limitée aux contraintes existant dans le problème, de la consistance duale, est plus forte que la variante conservative de la consistance de chemin, c'est à dire qu'elle est capable d'identifier plus d'instanciations inconsistantes sur un même réseau de contraintes. De plus, l'algorithme proposé pour établir la consistance duale conservative est en pratique très souvent beaucoup plus rapide que les algorithmes connus d'établissement de la consistance de chemin conservative.

Par ailleurs, nous avons également cherché à améliorer les algorithmes de recherche, et en particulier MGAC, tout d'abord en équipant celui-ci d'heuristiques de choix de valeurs, souvent négligées par la communauté scientifique. Nous nous sommes pour cela basés sur l'heuristique de Jeroslow-Wang « à deux faces », bien connue dans le cadre de la résolution du problème SAT. En utilisant les deux techniques de conversion de problèmes CSP vers SAT les plus utilisées, nous sommes parvenus à montrer comment

cette heuristique se comporterait sur le problème SAT issu de la conversion d'un CSP quelconque. Cela nous a permis tout d'abord de retrouver les comportements des heuristiques de choix de valeurs déjà utilisées pour la résolution de CSP, mais aussi de définir de nouvelles heuristiques, capables d'exploiter la bidirectionnalité des contraintes ainsi que les propriétés des branchements binaires effectués par un algorithme MGAC. Enfin, nous avons cherché à utiliser une hybridation entre un algorithme de recherche locale basé sur la pondération des contraintes et un algorithme MGAC, afin, d'une part, de mieux guider l'heuristique de choix de variables $dom/wdeg$, et, d'autre part, pour exploiter la réfutation de sous-arbres effectuée par MGAC via l'apprentissage de nogoods.

De manière transversale, l'ensemble des techniques développées dans le cadre de cette thèse a amené à la résolution d'une API pour le langage Java, capable de résoudre un CSP au sein d'une application Java quelconque. Cette API a été développée dans l'optique « boîte noire » : le moins de paramètres et d'expertise possibles sont demandés à l'utilisateur. Cette API nous a permis de développer plusieurs applications, notamment un programme cherchant des solutions optimales à des problèmes d'Open Shop, et un extracteur de noyaux insatisfiables, grâce à un algorithme $wcore$ étendu. Un prouveur basé sur CSP4J a concouru lors des compétitions internationales de prouveurs CSP avec des résultats encourageants.

La plupart de ces contributions ouvrent encore de nombreuses perspectives. La propriété de consistance duale peut servir de modèle à de très nombreuses formes de consistances de relation, plus fortes ou au contraire plus faciles à établir que les consistances déjà existantes, basées sur l'utilisation de la consistance d'arc. Les algorithmes permettant d'établir la consistance duale ainsi que sa variante conservative peuvent très probablement encore être améliorés.

L'étude des relations existant entre SAT et CSP, si elle a pu nous permettre de mettre en évidence de nouvelles heuristiques de choix de valeurs, nous amène aujourd'hui à considérer les techniques les plus modernes utilisées sur les prouveurs SAT, notamment la méthode CDCL et l'heuristique VSIDS. Il pourra être très intéressant de déterminer comment appliquer ces méthodes à la résolution de CSP.

La méthode d'hybridation proposée dans la section 3.2 de cette thèse reste simple, et on peut imaginer à la fois une forme d'hybridation plus forte, mais aussi des manières plus efficaces, plus précises d'exploiter la pondération de contraintes. D'autres statistiques peuvent être également apprises d'une recherche locale ou systématique, et pourraient être exploitées pour aller plus intelligemment vers la recherche d'une solution ou d'une preuve d'insatisfiabilité.

Bibliographie

[BACCHUS ET VAN BEEK 1998] F. Bacchus et P. van Beek. « On the conversion between non-binary and binary constraint satisfaction problems ». Dans *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 98)*, pages 311–318. AAAI Press, 1998.

[BECK *et al.* 2004] J.C. Beck, P. Prosser, et R.J. Wallace. « Variable ordering heuristics show promise ». Dans *Proceedings of CP'04*, pages 711–715, 2004.

[BELLICHA *et al.* 1994] A. Bellicha, C. Capelle, M. Habib, T. Kokény, et M.C. Vilarem. « CSP techniques using partial orders on domain values ». Dans *Proceedings of ECAI'94 workshop on constraint satisfaction issues raised by practical applications*, 1994.

[BENHAMOU ET SAÏS 1994] B. Benhamou et L. Saïs. « Tractability through symmetries in propositional calculus ». *Journal of Automated Reasoning*, 12(1) :89–102, 1994.

[BENHAMOU *et al.* 1994] F. Benhamou, D. MacAllester, et P. van Hentenryck. « CLP(Intervals) revisited ». Dans *Proceedings of ILPS'94*, pages 124–138, 1994.

[BERLANDIER 1995] P. Berlandier. « Improving domain filtering using restricted path consistency ». Dans *Proceedings of IEEE-CAIA'95*, 1995.

[BESSIÈRE 1994] C. Bessière. « Arc consistency and arc consistency again ». *Artificial Intelligence*, 65 :179–190, 1994.

[BESSIÈRE ET DEBRUYNE 2005] C. Bessière et R. Debruyne. « Optimal and suboptimal singleton arc consistency algorithms ». Dans *Proceedings of IJCAI'05*, pages 54–59, 2005.

[BESSIÈRE ET DEBRUYNE 2008] C. Bessière et R. Debruyne. « Theoretical Analysis of Singleton Arc Consistency and Its Extensions ». *Artificial Intelligence*, 172(1) :29–41, 2008.

[BESSIÈRE ET RÉGIN 1996] C. Bessière et J.-C. Régin. « MAC and combined heuristics : two reasons to forsake FC (and CBJ ?) on hard problems ». Dans *Proceedings of CP'96*, pages 61–75, 1996.

[BESSIÈRE ET RÉGIN 1997] C. Bessière et J.-C. Régin. « Arc consistency for general constraint networks : preliminary results ». Dans *Proceedings of IJCAI'97*, pages 398–404, 1997.

[BESSIÈRE ET VAN HENTENRYCK 2003] C. Bessière et P. van Hentenryck. « To be or not to be... a global constraint ». Dans *Proceedings of CP'03*, pages 789–794, 2003.

[BESSIÈRE *et al.* 1999] C. Bessière, E.C. Freuder, et J. Régin. « Using constraint metaknowledge to reduce arc consistency computation ». *Artificial Intelligence*, 107 :125–148, 1999.

[BESSIÈRE *et al.* 2001] C. Bessière, A. Chmeiss, et L. Saïs. « Neighborhood-based variable ordering heuristics for the Constraint Satisfaction Problem ». Dans *Proceedings of CP'01*, pages 565–569, 2001.

[BESSIÈRE *et al.* 2005] C. Bessière, J.-C. Régin, R.H.C. Yap, et Y. Zhang. « An optimal coarse-grained arc consistency algorithm ». *Artificial Intelligence*, 165(2) :165–185, 2005.

[BLIEK 1997] C. Bliek. « Wordwise Algorithms and Improved Heuristics for Solving Hard Constraint Satisfaction Problems ». Rapport technique, ERCIM, 1997. `http://citeseer.ist.psu.edu/152567.html`.

[BLIEK ET SAM-HAROUD 1999] C. Bliek et D. Sam-Haroud. « Path Consistency on Triangulated Constraint Graphs ». *Proceedings of IJCAI'99*, 16 :456–461, 1999.

[BORDEAUX *et al.* 2001] L. Bordeaux, E. Monfroy, et F. Benhamou. « Improved bounds on the complexity of kB-consistency ». Dans *Proceedings of IJCAI'01*, pages 303–308, 2001.

[BOUSSEMART *et al.* 2004A] F. Boussemart, F. Hemery, C. Lecoutre, et L. Saïs. « Boosting systematic search by weighting constraints ». Dans *Proceedings of ECAI'04*, pages 146–150, 2004.

[BOUSSEMART *et al.* 2004B] F. Boussemart, F. Hemery, et C. Lecoutre. « Revision ordering heuristics for the Constraint Satisfaction Problem ». Dans *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.

[BOUSSEMART *et al.* 2004C] F. Boussemart, F. Hemery, C. Lecoutre, et L. Saïs. « Support inference for generic filtering ». Dans *Proceedings of CP'04*, pages 721–725, 2004.

[BOUTALEB *et al.* 2006] K. Boutaleb, P. Jégou, et C. Terrioux. « (No)good Recording and ROBDDs for Solving Structured (V)CSPs ». *Proceedings of ICTAI'06*, pages 297–304, 2006.

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

[CABON *et al.* 1999] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, et J.P. Warners. « Radio Link Frequency Assignment ». *Constraints*, 4(1) :79–89, 1999.

[CHMEISS ET JÉGOU 1998] A. Chmeiss et P. Jégou. « Efficient path-consistency propagation ». *International Journal on Artificial Intelligence Tools*, 7(2) :121–142, 1998.

[CODOGNET ET DIAZ 1996] P. Codognet et D. Diaz. « Compiling constraints in clp(FD) ». *Journal of Logic Programming*, 27(3) :185–226, 1996.

[COLLAVIZZA *et al.* 1999] H. Collavizza, F. Delobel, et M. Rueher. « Comparing partial consistencies ». *Reliable computing*, 5 :213–228, 1999.

[COOK 1971] S.A. Cook. « The complexity of theorem-proving procedures ». *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[COOPER 1997] M.C. Cooper. « Fundamental properties of neighbourhood substitution in constraint satisfaction problems ». *Artificial Intelligence*, 90 :1–24, 1997.

[DAVIS ET PUTNAM 1960] M. Davis et H. Putnam. « A Computing Procedure for Quantification Theory ». *Journal of the ACM (JACM)*, 7(3) :201–215, 1960.

[DAVIS *et al.* 1962] M. Davis, G. Logemann, et D. Loveland. « A machine program for theorem-proving ». *Communications of the ACM*, 5(7) :394–397, 1962.

[DE KLEER 1989] J. de Kleer. « A comparison of ATMS and CSP techniques ». Dans *Proceedings of IJCAI'89*, pages 290–296, 1989.

[DEBRUYNE 1999] R. Debruyne. « A strong local consistency for constraint satisfaction ». *Proceedings of ICTAI'1999*, pages 202–209, 1999.

[DEBRUYNE ET BESSIÈRE 1997A] R. Debruyne et C. Bessière. « From restricted path consistency to max-restricted path consistency ». Dans *Proceedings of CP'97*, pages 312–326, 1997.

[DEBRUYNE ET BESSIÈRE 1997B] R. Debruyne et C. Bessière. « Some practicable filtering techniques for the constraint satisfaction problem ». *Proceedings of the 15th IJCAI*, 412417, 1997.

[DEBRUYNE ET BESSIÈRE 2001] R. Debruyne et C. Bessière. « Domain filtering consistencies ». *Journal of Artificial Intelligence Research*, 14 :205–230, 2001.

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

[DUBOIS *et al.* 1993] O. Dubois, P. Andre, Y. Boufkhad, et J. Carlier. « SAT versus UNSAT ». Dans *Second DIMACS Challenge*, pages 299–314, 1993.

[EISENBERG ET FALTINGS 2003] C. Eisenberg et B. Faltings. « Using the Breakout Algorithm to Identify Hard and Unsolvable Subproblems ». *the Proceedings of Principles and Practice of Constraint Programming CP-2003, LNCS*, 2833 :822–826, 2003.

[FAHLE *et al.* 2001] T. Fahle, S. Schamberger, et M. Sellman. « Symmetry Breaking ». Dans *Proceedings of CP'01*, pages 93–107, 2001.

[FALTINGS 1994] B. Faltings. « Arc consistency for continuous variables ». *Artificial Intelligence*, 65 :363–376, 1994.

[FOCACCI ET MILANO 2001] F. Focacci et M. Milano. « Global Cut Framework for Removing Symmetries ». Dans *Proceedings of CP'01*, pages 77–92, 2001.

[FREEMAN 1995] J.W. Freeman. « *Improvements to Propositional Satisfiability Search Algorithms* ». PhD thesis, University of Pennsylvania, 1995.

[FREUDER 1991] E.C. Freuder. « Eliminating interchangeable values in constraint satisfaction problems ». Dans *Proceedings of AAAI'91*, pages 227–233, 1991.

[FROST ET DECHTER 1994] D. Frost et R. Dechter. « Dead-end Driven Learning ». *Proceedings of AAAI'94*, pages 294–300, 1994.

[FROST ET DECHTER 1995] D. Frost et R. Dechter. « Look-ahead value ordering for constraint satisfaction problems ». Dans *Proceedings of IJCAI'95*, pages 572–578, 1995.

[GALINIER ET HAO 1997] P. Galinier et J.K. Hao. « Tabu search for maximal constraint satisfaction problems ». Dans *Proceedings of CP'97*, pages 196–208, 1997.

[GALINIER ET HAO 2004] P. Galinier et J.K. Hao. « A General Approach for Constraint Solving by Local Search ». *Journal of Mathematical Modelling and Algorithms*, 3(1) :73–88, 2004.

[GEELEN 1992] P.A. Geelen. « Dual viewpoint heuristics for binary constraint satisfaction problems ». Dans *Proceedings of ECAI'92*, pages 31–35, 1992.

[GENT 2002] I.P. Gent. « Arc Consistency in SAT ». Dans *Proceedings of ECAI'02*, pages 121–125, 2002.

[GENT *et al.* 2006A] I.P. Gent, C. Jefferson, et I. Miguel. « Minion : A Fast, Scalable, Constraint Solver ». Dans *Proceedings of ECAI'06*, 2006.

[GENT *et al.* 2006B] I.P. Gent, C. Jefferson, et I. Miguel. « Watched literals for constraint propagation in Minion ». Dans *Proceedings of CP'06*, pages 182–197, 2006.

[GOMES *et al.* 2000] C.P. Gomes, B. Selman, N. Crato, et H. Kautz. « Heavy-tailed phenomena in satisfiability and constraint satisfaction problems ». *Journal of Automated Reasoning*, 24 :67–100, 2000.

[GUÉRET ET PRINS 1999] C. Guéret et C. Prins. « A new lower bound for the open-shop problem ». *Annals of Operations Research*, 92 :165–183, 1999.

[HARALICK ET ELLIOTT 1980] R.M. Haralick et G.L. Elliott. « Increasing tree search efficiency for constraint satisfaction problems ». *Artificial Intelligence*, 14 :263–313, 1980.

[HEMERY *et al.* 2006] F. Hemery, C. Lecoutre, L. Saïs, et F. Boussemart. « Extracting MUCs from Constraint Networks ». Dans *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'2006)*, pages 113–117, Trento, Italy, August 2006.

[HULUBEI ET O'SULLIVAN 2005] T. Hulubei et B. O'Sullivan. « Search Heuristics and Heavy-tailed behaviour ». Dans *Proceedings of CP'05*, pages 328–342, 2005.

[HWANG ET MITCHELL 2005] J. Hwang et D.G. Mitchell. « 2-way vs d-way branching for CSP ». Dans *Proceedings of CP'05*, pages 343–357, 2005.

[HYVONEN 1992] E. Hyvonen. « Constraint reasoning based on interval arithmetic : the tolerance approach ». *Artificial Intelligence*, 58 :71–112, 1992.

[JANSSEN *et al.* 1989] P. Janssen, P. Jegou, B. Nouguier, et M.C. Vilarem. « A filtering process for general constraint-satisfaction problems : achieving pairwise-consistency using an associated binary representation ». Dans *Proc. of IEEE International Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[JEROSLOW ET WANG 1990] R. G. Jeroslow et J. Wang. « Solving Propositional Satisfiability Problems ». *Annals of Mathematics and Artificial Intelligence*, 1 :167–187, 1990.

[JUSSIEN ET LHOMME 2000] N. Jussien et O. Lhomme. « Local search with constraint propagation and conflict-based heuristics ». Dans *Proceedings of AAAI'2000*, pages 169–174, Austin, TX, USA, août 2000.

[KASIF 1990] S. Kasif. « On the parallel complexity of discrete relaxation in constraint satisfaction networks ». *Artificial Intelligence*, 45 :275–286, 1990.

[KATSIRELOS ET BACCHUS 2005] G. Katsirelos et F. Bacchus. « Generalized NoGoods in CSPs ». Dans *Proceedings of AAAI'05*, pages 390–396, 2005.

[LECOUTRE 2006] C. Lecoutre. « Benchmarks 2.0 - XML representation of CSP instances ». `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks`, 2006.

[LECOUTRE ET CARDON 2005] C. Lecoutre et S. Cardon. « A greedy approach to establish singleton arc consistency ». Dans *Proceedings of IJCAI'05*, pages 199–204, 2005.

[LECOUTRE ET HEMERY 2007] C. Lecoutre et F. Hemery. « A Study of Residual Supports in Arc Consistency ». Dans *Proceedings of IJCAI'2007*, pages 125–130, 2007.

[LECOUTRE ET SZYMANEK 2006] C. Lecoutre et R. Szymanek. « Generalized Arc Consistency for Positive Table Constraints ». Dans *Proceedings of CP'06*, pages 284–298, 2006.

[LECOUTRE ET VION 2005] C. Lecoutre et J. Vion. « Bound Consistencies for the Discrete CSP ». Dans *Proceedings of the Second International Workshop on Constraint Propagation And Implementation*, pages 17–31, 2005.

[LECOUTRE ET VION 2008] C. Lecoutre et J. Vion. « Enforcing Arc Consistency using Bitwise Operations ». *Constraint Programming Letters*, 2 :21–35, 2008.

[LECOUTRE *et al.* 2004] C. Lecoutre, F. Boussemart, et F. Hemery. « Backjump-based techniques versus conflict-directed heuristics ». Dans *Proceedings of ICTAI'04*, pages 549–557, 2004.

[LECOUTRE *et al.* 2005] C. Lecoutre, F. Boussemart, et F. Hemery. « Abscon 2005 ». Dans *Proceedings of CPAI'05*, volume II, pages 67–72, 2005.

[LECOUTRE *et al.* 2007A] C. Lecoutre, S. Cardon, et J. Vion. « Conservative Dual Consistency ». Dans *Proceedings of AAAI'07*, pages 237–242, 2007.

[LECOUTRE *et al.* 2007B] C. Lecoutre, S. Cardon, et J. Vion. « Path Consistency by Dual Consistency ». Dans *Proceedings of CP'2007*, 2007.

[LECOUTRE *et al.* 2007C] C. Lecoutre, L. Saïs, S. Tabary, et V. Vidal. « Nogood Recording from Restarts ». Dans *Proceedings of IJCAI'07*, 2007.

[LECOUTRE *et al.* 2007D] C. Lecoutre, L. Saïs, et J. Vion. « Using SAT Encodings to Derive CSP Value Ordering Heuristics ». *JSAT Special Issue on SAT/CP Integration*, 1 :169–186, 2007.

[LHOMME 1993] O. Lhomme. « Consistency techniques for numeric CSPs ». Dans *Proceedings of IJCAI'93*, pages 232–238, 1993.

[LHOMME 1994] O. Lhomme. « *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles* ». PhD thesis, Université de Nice-Sophia Antipolis, 1994.

[LI ET ANBULAGAN 1997] C.M. Li et Anbulagan. « Heuristics Based on Unit Propagation for Satisfiability Problems ». Dans *Proceedings of IJCAI'97*, pages 366–371, 1997.

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

[MAZURE *et al.* 1998] B. Mazure, L. Saïs, et É. Grégoire. « Boosting Complete Techniques thanks to local search methods ». *Annals of Mathematics and Artificial Intelligence*, 22 :319–331, 1998.

[MCGREGOR 1979] J.J. McGregor. « Relational Consistency Algorithms and their application in finding subgraph and graph isomorphisms ». *Information Sciences*, 19 :229–250, 1979.

[MEHTA ET VAN DONGEN 2005A] D. Mehta et M. R. C. van Dongen. « Static Value Ordering Heuristics for Constraint Satisfaction Problems ». Dans *Proceedings of the 2nd Itl Workshop on Constraint Propagation And Implementation, Volume I*, 2005.

[MEHTA ET VAN DONGEN 2005B] D. Mehta et M.R.C. van Dongen. « Reducing checks and revisions in coarse-grained MAC algorithms ». Dans *Proceedings of IJCAI'05*, pages 236–241, 2005.

[METLOV 2006] K. L. Metlov. « Java Expressions Library ». http ://kinetic.ac.donetsk.ua/JEL/, 2006.

[MINTON *et al.* 1992] S. Minton, M.D. Johnston, A.B. Philips, et P. Laird. « Minimizing conflicts : a heuristic repair method for constraint-satisfaction and scheduling problems ». *Artificial Intelligence*, 58(1-3) :161–205, 1992.

[MOHR ET HENDERSON 1986] R. Mohr et T.C. Henderson. « Arc and path consistency revisited ». *Artificial Intelligence*, 28 :225–233, 1986.

[MOHR ET MASINI 1988] R. Mohr et G. Masini. « Good old discrete relaxation ». Dans *Proceedings of ECAI'88*, pages 651–656, 1988.

[MOLLOY 2003] M. Molloy. « Models for random constraint satisfaction problems ». *SIAM Journal of computing*, 32(4) :935–949, 2003.

[MONTANARI 1974] U. Montanari. « Network of constraints : Fundamental properties and applications to picture processing ». *Information Science*, 7 :95–132, 1974.

[MORRIS 1993] P. Morris. « The breakout method for escaping from local minima ». Dans *Proceedings of AAAI'93*, pages 40–45, 1993.

[NEVEU ET PRCOVIC 2002] B. Neveu et N. Prcovic. « Progressive Focusing Search ». Dans *Proceedings of ECAI'02*, pages 126–130, 2002.

[PERLIN 1991] M. Perlin. « Arc consistency for factorable relations ». *Tools for Artificial Intelligence, 1991. TAI'91., Third International Conference on*, pages 340–345, 1991.

[PRESTWICH 2001] S. Prestwich. « Local Search and Backtracking versus Non-Systematic Backtracking ». Dans *Papers from the AAAI 2001 Fall Symposium on Using Uncertainty Within Computation*, pages 109–115, North Falmouth, Cape Cod, MA, November 2-4 2001. The American Association for Artificial Intelligence, The AAAI Press.

[PUGET 2004] J.F. Puget. « Constraint programming next challenge : Simplicity of use ». Dans *Proceedings of CP'04*, pages 5–8. Springer, 2004.

[RICHAUD *et al.* 2006] G. Richaud, H. Cambazard, B. O'Sullivan, et N Jussien. « Automata for Nogood Recording in Constraint Satisfaction Problems ». Dans *Proceedings of SAT/CP Workshop held with CP'06*, 2006.

[ROSSI *et al.* 1990] F. Rossi, C. Petrie, et V. Dhar. « On the Equivalence of Constraint Satisfaction Problems ». Dans Luigia Carlucci Aiello, éditeur, *ECAI'90 : Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, Stockholm, 1990. Pitman.

[RÉGIN 1994] J.-C. Régin. « A filtering algorithm for constraints of difference in CSPs ». Dans *Proceedings of AAAI'94*, pages 362–367, 1994.

[SABIN ET FREUDER 1994] D. Sabin et E. Freuder. « Contradicting conventional wisdom in constraint satisfaction ». Dans *Proceedings of CP'94*, pages 10–20, 1994.

[SCHAERF 1997] A. Schaerf. « Combining Local Search and Look-Ahead for Scheduling and Constraint Satisfaction Problems ». Dans *Proceedings of IJCAI'97*, pages 1254–1259, 1997.

[SCHIEX ET VERFAILLIE 1993] T. Schiex et G. Verfaillie. « Nogood Recording for static and dynamic constraint satisfaction problems ». *Tools with Artificial Intelligence, 1993. TAI'93. Proceedings. Fifth International Conference on*, pages 48–55, 1993.

[SCHULTE 1999] C. Schulte. « Comparing Trailing and Copying for Constraint Programming ». Dans *Proceedings of ICLP'99*, pages 275–289, 1999.

[SELMAN *et al.* 1992] B. Selman, H. Levesque, et D. Mitchell. « A new method for solving hard satisfiability problems ». Dans *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446. AAAI Press, 1992.

[SELMAN *et al.* 1997] B. Selman, H. Kautz, et D. McAllester. « Ten challenges in propositional reasoning and search ». Dans *Proceedings of IJCAI'97*, 1997.

[SINGH 1995] M. Singh. « Path consistency revisited ». *Tools with Artificial Intelligence, 1995. Proceedings., Seventh International Conference on*, pages 318–325, 1995.

[SMITH ET STURDY 2005] B. M. Smith et P. Sturdy. « Value Ordering for Finding All Solutions ». Dans *Proceedings of IJCAI'05*, pages 311–316, 2005.

[STALLMAN 1991] R.M. Stallman. « GNU General Public License ». GNU Project–Free Software Foundation, http ://gnu.org/licenses, 1991.

[STALLMAN 1999] R.M. Stallman. « GNU Lesser General Public License ». GNU Project–Free Software Foundation, http ://-gnu.org/licenses, 1999.

[STERGIOU ET WALSH 2006] K. Stergiou et T. Walsh. « Inverse Consistencies for Non-binary Constraints ». *Proceedings of ECAI*, 6 :153–157, 2006.

[SZYMANEK ET O'SULIVAN 2006] R. Szymanek et B. O'Sulivan. « Guiding search using constraint-level advice ». Dans *Proceedings of ECAI'06*, pages 158–162, 2006.

[TAILLARD 1993] E. Taillard. « Benchmarks for basic scheduling problems ». *European journal of operations research*, 64 :278–295, 1993.

[TERRIOUX 2001] C. Terrioux. « Cooperative Search and Nogood Recording ». *Proceedings IJCAI'2001*, pages 260–265, 2001.

[ULLMANN 1976] J.R. Ullmann. « An algorithm for subgraph isomorphism ». *Journal of the ACM*, 23(1) :31–42, 1976.

[VAN DONGEN 2004] M.R.C. van Dongen. « Saving support-checks does not always save time ». *Artificial Intelligence Review*, 21(3-4) :317–334, 2004.

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

[VAN DONGEN 2006] M.R.C. van Dongen. « Beyond Singleton Arc Consistency ». Dans *Proceedings of ECAI'05*, pages 163–167, 2006.

[VAN DONGEN *et al.* 2006A] M. van Dongen, C. Lecoutre, O. Roussel, R. Szymanek, F. Hemery, C. Jefferson, et R. Wallace. « Second International CSP Solvers Competition ». `http://cpai.ucc.ie/06/Competition.html`, 2006.

[VAN DONGEN *et al.* 2006B] M. van Dongen, C. Lecoutre, O. Roussel, R. Szymanek, F. Hemery, C. Jefferson, et R. Wallace. « XML Representation of Constraint Networks, Version 2.0 ». http ://cpai.ucc.ie/06/Representation.pdf, 2006.

[VAN HENTENRYCK *et al.* 1992] P. van Hentenryck, Y. Deville, et CM. Teng. « A generic arc-consistency algorithm and its specializations ». *Artificial Intelligence*, 57 :291–321, 1992.

[VAN HENTENRYCK *et al.* 1998] P. van Hentenryck, V. Saraswat, et Y. Deville. « Design, implementation and evaluation of the constraint language cc(FD) ». *Journal of Logic Programming*, 37(1-3) :139–164, 1998.

[VAN HOEVE 2001] W. van Hoeve. « The alldifferent constraint : A survey », 2001.

[VION 2006] J. Vion. « Constraint Satisfaction Problem For Java ». http ://cspfj.sourceforge.net/, 2006.

[VION 2007] J. Vion. « Hybridation de prouveurs CSP et apprentissage ». Dans *Actes des troisièmes Journées Francophones de Programmation par Contraintes (JFPC '07)*, 2007.

[WALLACE 1993] R.J. Wallace. « Why AC3 is almost always better than AC4 for establishing arc consistency in CSPs ». Dans *Proceedings of IJCAI'93*, pages 239–245, 1993.

[WALLACE 2005] R.J. Wallace. « Heuristic policy analysis and efficiency assessment in Constraint Satisfaction search ». Dans *Proceedings of CPAI'05 workshop held with CP'05*, pages 79–91, 2005.

[WALSH 2000] T. Walsh. « SAT v CSP ». Dans *Proceedings of CP'00*, pages 441–456, 2000.

[XU ET LI 2000] K. Xu et W. Li. « Exact phase transitions in random constraint satisfaction problems ». *Journal of Artificial Intelligence Research*, 12 :93–103, 2000.

[XU *et al.* 2007] K. Xu, F. Boussemart, F. Hemery, et C. Lecoutre. « A simple model to generate hard satisfiable instances ». *Artificial Intelligence*, 171 :514–534, 2007.

[ZHANG ET YAP 2001] Y. Zhang et R.H.C. Yap. « Making AC3 an optimal algorithm ». Dans *Proceedings of IJCAI'01*, pages 316–321, 2001.

[ZHANG *et al.* 2001] L. Zhang, C.F. Madigan, M.W. Moskewicz, et S. Malik. « Efficient Conflict Driven Learning in a Boolean Satisfiability Solver ». Dans *Proceedings of ICCAD'01*, pages 279–285, 2001.

[ZHOU ET WALLACE 2005] N.F. Zhou et M. Wallace. « A Simple Constraint Solver in Action Rules for the CP'05 Solver Competition ». Dans *Proceedings of the CP workshop on Constraint Propagation and Implementation*, 2005.

Résumé

Nous proposons plusieurs techniques visant à résoudre en pratique le problème NP-complet de satisfaction de contraintes de manière générique. Nous distinguons deux grands axes de techniques de résolution de CSP : l'inférence et la recherche. Nous avons contribué à l'amélioration des techniques d'inférence en nous concentrant sur la propriété centrale qu'est la consistance d'arc : optimisations des algorithmes de consistance d'arc, comportement de plusieurs algorithmes d'inférence aux bornes de domaines discrets, et enfin une alternative intéressante à la consistance de chemin : la consistance duale. Cette propriété nous a amenés à concevoir des algorithmes de consistance de chemin forte très efficaces. La variante conservative de cette consistance est de plus plus forte que la consistance de chemin conservative, tout en restant plus rapide à établir en pratique.

Par ailleurs, nous avons également cherché à améliorer MGAC, tout d'abord en équipant celui-ci d'heuristiques de choix de valeurs. Nous nous sommes pour cela basés sur l'heuristique de Jeroslow-Wang, issue du problème SAT. En utilisant deux techniques de conversion de CSP vers SAT, nous montrons comment cette heuristique se comporterait sur un CSP. Enfin, nous avons cherché à utiliser une hybridation entre un algorithme de recherche locale basé sur la pondération des contraintes et un algorithme MGAC équipé de l'heuristique $dom/wdeg$, en exploitant les possibilités d'apprentissage de l'un et l'autre algorithmes.

De manière transversale, l'ensemble des techniques développées dans le cadre de cette thèse a amené à la résolution d'une API pour le langage Java, capable de résoudre un CSP au sein d'une application Java quelconque. Cette API a été développée dans l'optique "boîte noire" : le moins de paramètres et d'expertise possibles sont demandés à l'utilisateur. Un prouveur basé sur CSP4J a concouru lors les compétitions internationales de prouveurs CSP avec des résultats encourageants.

Abstract

We propose several techniques aimed to solve the NP-complete Constraint Satisfaction Problem in practice. We distinguish two main approaches : search and inference. We have contributed to improve inference techniques by evolving the central Arc Consistency property. We optimized the best AC algorithms, we studied their behavior on the bounds of the discrete domains, and we finally studied an interesting alternative to Path Consistency : Dual Consistency. This property leads us to design new algorithms that can establish Path Consistency very efficiently. Conservative Dual Consistency is stronger than Conservative Path Consistency, and is much faster to establish.

Besides, we have tried to improve MGAC, first by equipping it of Value Ordering Heuristics. We studied how the well known Jeroslow-Wang heuristic, from the SAT problem, would behave if applied to the translation of a CSP problem in SAT. Finally, we studied a hybridization between a local search algorithm based on constraint weighting and MGAC, by exploiting the learning abilities of both algorithms.

All techniques developed during this PhD thesis lead to the development of a new API for the Java language, namely CSP4J, able to solve a CSP as part of any Java application. This API is a "black box" : as less parameters and expertise are required from a user point of view. A solver based on CSP4J took part in International Solved Competitions with promising results.