Thèse de doctorat
Mention : Informatique
Spécialité : Intelligence artificielle

Graphes et contraintes

Thèse rédigée par

Samy Modeliar Mouny

Soutenue le 22 Mars 2017

Membres du jury

Christophe Lecoutre, professeur des universités à l'université d'Artois, directeur

Gilles Audemard, professeur des universités à l'université d'Artois, codirecteur

Christine Solnon, professeure des universités à l'INSA de Lyon

Eric Monfroy, professeur des universités à l'université de Nantes, rapporteur

Frédéric Koriche, professeur des universites à l'universite d'Artois

Philippe Vismara, maître de conférences à SupAgro, université de Montpellier, rapporteur

CRIL - CNRS UMR 8188 - Université d'Artois

À ma famille,
en particulier
à Rama et Olga.

Introduction

Ce travail a été réalisé à l'université d'Artois au sein du laboratoire CRIL sous la direction de Christophe Lecoutre et Gilles Audemard. Il s'agit de travaux innovants ou de travaux déjà commencés au CRIL et poursuivis dans ce cadre.

Les recherches au CRIL concernent l'intelligence artificielle et ses applications. Deux axes majeurs y sont développés : d'une part, la représentation des connaissances et d'autre part les algorithmes pour l'inférence et les contraintes. Ma recherche se situe dans cette seconde thématique et en particulier dans l'étude des réseaux de contraintes.

Une première thèse en mathématiques et la proximité du laboratoire du CRIL m'ont amené à travailler sur ces thématiques bien avant le début de cette thèse. Certains lecteurs pourront donc retrouver des résultats publiés il y a quelques années.

Le raisonnement par contraintes a été initié au cours des années 70 avec notamment les premiers travaux autour du cadre des problèmes de satisfaction de contraintes (CSP pour Constraint Satisfaction problem) [76]. Le cadre CSP est générique, au sens où il permet de modéliser naturellement de nombreux problèmes combinatoires. Une instance (du cadre) CSP est représentée à l'aide de réseaux de contraintes (CN pour Constraint Network) [43, 69] qui sont simplement constitués de variables discrètes (ou continues) auxquelles il faut assigner une valeur (ou un intervalle) et de contraintes liant certaines de ces variables. Générer une instance CSP est un processus naturel au cours duquel les inconnues du problème sont généralement posées par l'utilisateur dans un premier temps, et les équations représentant les contraintes données dans un second temps. Il est important de souligner que de nombreux problèmes de décision de nature combinatoire se modélisent à l'aide de variables et de contraintes discrètes. Par exemple, on peut citer les problèmes de planification, d'ordonnancement, d'allocation de ressources, de puzzles ou de bio-informatique. Résoudre une instance CSP (ou de manière équivalente résoudre un CN) revient à trouver une solution ou prouver qu'aucune solution n'existe. Un système informatique qui permet de résoudre ce type d'instances est appelé solveur de contraintes. Malgré les progrès en matière de puissance de calcul des ordinateurs et comme nous le verrons page 3, la méthode naïve qui consiste à essayer toutes les combinaisons possibles de l'espace de recherche d'un réseau est vouée à l'échec. Pour de nombreuses instances de problèmes, cela requière un temps quasi-infini à l'échelle humaine (par exemple, plusieurs siècles ou millénaires avant d'aboutir). Il est donc nécessaire de développer des stratégies intelligentes afin d'éliminer un grand nombre de combinaisons sans avoir à les tester.

Une grande majorité des travaux de l'époque des années 70 et 80 considéraient que les contraintes à traiter étaient binaires (c'est-à-dire, portant uniquement sur deux variables) et définies en extension par énumération de l'ensemble des couples de valeurs satisfaisant chaque contrainte. Dans ce contexte, la notion majeure de cohérence (consistency) permettant de caractériser différentes propriétés nécessaires à la présence de solutions dans les réseaux de contraintes a été introduite. Les premières cohérences introduites alors ont été la cohérence de noeud, la cohérence d'arc et la cohérence de chemin [76]. Leur intérêt est qu'elles permettent de filtrer efficacement l'espace de recherche de solutions en éliminant certaines valeurs incohérentes (avec la cohérence de noeud et d'arc) ou certains

couples de valeurs (avec la cohérence de chemin). Conjointement à œs travaux sur les cohérences, et les algorithmes de filtrage qui les accompagnent, il a été rapidement observé que l'uti-

lisation de bonnes heuristiques pour guider un algorithme de recherche arborescent (avec retour-arrière) était primordial [58]. À la fin des années 80, il était généralement admis que l'algorithme de recherche arborescente appelé MAC (qui établit et maintient la cohérence d'arc à chaque pas de la recherche) combinée à min dom, l'heuristique de choix de variables qui choisit systématiquement la variable avec le plus petit domaine comme prochaine variable à instancier, était la solution générique la plus efficace pour résoudre les instances CSP.

Le terme de programmation par contraintes est apparu un peu plus tard (que les années 70 et 80). Il semble admis que ce terme recoupe les développements effectués à partir des années 90 autour des contraintes globales, contraintes qui sont d'arité quelconque (et non plus systématiquement binaires) et de sémantique particulière. Un résultat retentissant, et représentatif, est ainsi l'introduction de la contrainte allDifferent et de son algorithme de filtrage basé sur des éléments de la théorie des graphes (couplage maximum) [92]. Depuis, le nombre de contraintes globales n'a cessé de croître, même si, bien sûr, certaines d'entre elles semblent plus anecdotiques que d'autres. Si on se réfère aux contraintes les plus usitées, c'est-à-dire, celles le plus souvent implantées dans les solveurs et/ou celles le plus souvent utilisées dans la modélisation des problèmes de la littérature, on aboutit à un nombre raisonnable d'une vingtaine de contraintes globales telles que décrites dans le format XCSP3. L'usage de la programmation par contraintes reste encore aujourd'hui réservé aux spécialistes [90], ce qui explique souvent l'utilisation d'autres techniques parfois moins efficaces. Cependant, depuis quelques années des efforts importants sont réalisés par la communauté pour diffuser plus largement ces idées [51]. Citons par exemple le format XCSP3 [96, 68, 77] permettant de définir un problème grâce au format XML ou encore le paquet Python Numberjack [59] qui permet de formaliser un problème grâce à une classe Python.

Un cas particulier[1] représente à lui seul une théorie très vaste. Il s'agit du cas où les variables ne peuvent prendre que les deux valeurs booléennes True ou False (0 ou 1). Cela correspond au **problème SAT** pour problème de satisfiabilité. Se placer dans une algèbre de Boole permet d'utiliser de nombreux outils de logique. Par ailleurs, la base 2 est la plus petite base possible. Des raisonnements fins au digit près sont possibles. Dans un problème SAT, les disjonctions de littéraux sont appelées des clauses. Par exemple, la clause $x \vee y \vee \neg z \vee x$ signifie x ou y ou pas z ou x, ce qui peut se simplifier en $x \vee y \vee \neg z$. La première machine logique est due à Gardner [49] mais les premiers algorithmes efficaces de résolution du problème SAT ont réellement débuté en 1960 avec DP-60 [39, 41] et l'algorithme DPLL [40]. Les solveurs SAT ont progressé très rapidement et l'intégration de l'apprentissage (à chaque échec, on déduit de ce dernier une clause responsable du conflit) en font maintenant des démonstrateurs très efficaces. Ils sont nommés CDCL (pour Conflict Driven, Clause Learning [107, 84, 56, 54, 98, 74]).

Ils sont devenus très efficaces et leur influence dépasse largement le cadre de la programmation par contraintes. Citons par exemple la récente résolution du problème de mathématiques de la bicoloration des triplets de Pythagore [60] grâce aux solvers SAT utilisés sur des super calculateurs. Les principales implémentations des solveurs SAT aujourd'hui sont zChaff (Moskewicz, Madigan, Zhao, Zhang, 2001), Siege (Ryan, 2003), MiniSAT (Eén et Sörensson, 2005), ManySAT (Hamadi, Jabbour et Sais, 2008) et Glucose (Audemard et Simon, 2009), basé sur Minisat.

La programmation par contraintes n'est pas la seule méthode envisageable pour résoudre des problèmes de nature combinatoire. La recherche opérationnelle en est une autre. Elle peut être définie comme l'ensemble des méthodes et techniques rationnelles d'analyse et de synthèse des phénomènes d'organisation utilisables pour élaborer de meilleures décisions [45]. Cette technique semble récente (milieu du siècle dernier) mais cette définition laisse

1. mathématiquement parlant, car certains spécialistes SAT considèrent que c'est l'inverse. En fait, il y a une bijection entre ce cas et la généralité des CSP à domaines finis.

penser qu'elle est beaucoup plus ancienne. L'intelligence artificielle a pour but d'effectuer avec de manière automatique des tâches réalisées habituellement grâce à l'intelligence humaine. La programmation par contraintes se situe ainsi à la jonction entre la recherche opérationnelle et l'intelligence artificielle.

Des méthodes dite d'optimisations linéaires existent également. Le nom de programmation linéaire est également utilisé mais cette appellation tend à être abandonnée afin d'éviter la confusion avec le terme de programmation informatique (PL). Cette théorie a été introduite par Georges Dantiz en 1947 [38] avec son algorithme de résolution du simplexe. Un problème d'optimisation linéaire s'intéresse à la minimisation une fonction linéaire sur un polyèdre convexe.

Dans cette thèse, nous nous intéresserons à deux types de liens entre les graphes et la programmation par contraintes. D'une part, la faculté qu'ont les réseaux de contraintes à résoudre les problèmes classiques de la théorie des graphes. Et d'autre part, l'utilisation de la théorie des graphes pour améliorer l'efficacité des solveurs de contraintes.

Contenu des chapitres

Chapitre 1

Un premier chapitre rappelle les fondements de la programmation par contraintes considéré sous un angle algébrique. Cette approche est assez différente de la présentation usuelle mais le point de vue théorique que l'on adopte facilitera le lien avec certaines théories mathématiques. Nous y introduirons bien sûr la notion de réseaux de contraintes (variables et contraintes). Nous y verrons pourquoi une méthode naïve ne peut aboutir et quelles stratégies ont été développées astucieusement pour contourner ce problème. En particulier, nous présenterons les algorithmes de recherche avec retour arrière et le filtrage par cohérence d'arc (AC). Nous verrons également d'autres filtrages liés à la cohérence d'arc, comme les filtrages par cohérences aux bornes (D-cohérence et Z-cohérence), For-ward Checking (FC) et le filtrage par cohérence d'arc singleton (SAC).

Chapitre 2

Dans le chapitre 2, nous détaillerons le fonctionnement d'un solveur de contraintes utilisant une structure particulière : les sparse set. Il s'agit d'une structure de données bien adaptée à la représentation des domaines des variables mais aussi à la gestion des informations à restaurer concernant l'instantiation courante et la propagation. Nous commencerons par indiquer les avantages et les inconvénients en terme de complexité par rapport à d'autres structures de données classiques. Nous verrons qu'elle est particulièrement efficace pour la restoration des domaines des variables. En revanche, elle est clairement moins performante pour trouver les extremums d'un ensemble ou pour itérer dans l'ordre croissant (ou décroissant). Nous décrirons les différentes opérations possibles sur les sparse set avant de donner un exemple de programmation en Python de cette structure. Nous donnerons ensuite le diagramme de classe d'un solveur utilisant les sparse set, puis nous détaillerons les classes importantes avec le code en Python. Enfin, un algorithme de filtrage pour la contrainte $x = y + z$ sera donné, illustrant les principes généraux de filtrage d'une contrainte. Cet algorithme présente trois phases partiellement imbriquées : la première garantit la propriété de Z-cohérence, le deuxième la propriété de D-cohérence et la dernière assure la propriété de cohérence d'arc.

Chapitre 3

Dans le chapitre 3, nous nous intéressons à la résolution, dans le cadre de la programmation par contraintes, d'un problème classique de la théorie des graphes : l'isomorphisme

de sous-graphe. Il s'agit de trouver une copie d'un graphe motif dans un graphe cible. Après avoir rappelé quelques définitions de théorie des graphes, nous expliquerons comment le problème est modélisé en programmation par contraintes. Les sommets du graphe motif représentent dans ce modèle les variables dont les domaines sont constitués des sommets du graphe cible. Des contraintes permettent d'exprimer les conditions nécessaires pour qu'une instanciation représente un sous-graphe du graphe cible isomorphe au graphe source. Nous décrirons ensuite les différentes méthodes existantes, celles utilisant des algorithmes non dédiés au problème comme FC et GAC et des méthodes plus efficaces car construites dans ce cadre comme LV2002, ILF et LAD. Nous donnerons également une méthode récente utilisant les graphes additionnels. Nous développerons alors notre méthode appelée SND pour score neighborhood dominance. Cette méthode est basée sur des fonctions de voisinage et sur des fonctions de score que nous définirons. Les fonctions de voisinage permettent de sélectionner certains sommets du graphe motif et du graphe cible afin d'évaluer leurs scores. Cela peut permettre de comparer des degrés ou des coefficients du graphe et d'en conclure qu'il n'est pas possible d'affecter un sommet du graphe motif à un sommet du graphe cible. Nous développerons alors un algorithme de filtrage basé sur trois fonctions de voisinage[2] et une fonction de score basée sur les matrices d'adjacence des graphes cible et motif. Nous utiliserons en particulier comme propriété le nombre de chemins d'une certaine longueur d'un sommet à un autre dans un graphe obtenu par le calcul des puissances des matrices d'adjacence. Nous comparerons enfin notre méthode aux méthodes existantes.

Chapitre 4

Dans le chapitre 4, nous exposerons une méthode de filtrage basée sur un algorithme MAC incomplet. En effet, la propagation de contraintes lancée par MAC à chaque étape de la recherche peut être relativement longue car l'obtention du point fixe (de la propagation) peut en particulier prendre beaucoup de temps pour un impact très limité. Nous commencerons donc par décomposer l'algorithme MAC en définissant les filtrages en *n* passes (MAC faisant un nombre de passes indéterminé), ce qui nous amènera à expliquer le mécanisme à l'œuvre derrière le concept de queue de propagation. À chaque étape, le processus de propagation de contraintes lancé par MAC peut renvoyer deux valeurs : `false`, si le réseau est devenu inconsistant ou `true` si le réseau obtenu est toujours arc cohérent. Nous constaterons que la longueur (le nombre de fois où une variable est ajoutée à la queue) de la queue de propagation est généralement plus courte lorsque MAC retourne la valeur `false`. L'idée sera donc d'arrêter la propagation juste après avoir estimé que l'algorithme ne renverrait probablement plus la valeur `false` et considéré que l'effort à produire pour obtenir un réseau arc cohérent ne serait pas rentable. Nous mettrons donc en place une fonction de coût qui nous permettra de prendre cette décision. Nous terminerons ce chapitre par des résultats expérimentaux.

Chapitre 5

Enfin, le chapitre 5 est un compromis entre résultats et perspectives. L'objectif est de faire apparaître rapidement des liaisons cachées entre les variables. Pour cela, nous commencerons par définir un graphe, appelé graphe primal, exprimant des informations explicites sur les contraintes. Les sommets du graphe primal représentent les variables du problème et deux sommets sont liés si les deux variables correspondantes appartiennent au scope d'une même contrainte. Comme dans d'autres représentations, seulement une partie des informations est alors conservée. Nous exposerons donc une méthode pour construire statistiquement un graphe du même type en espérant que de nouvelles liaisons vont apparaître. Nous détaillerons la collecte de données qui est réalisée à partir d'un

2. Id, adj, et all

de recherche en évaluant l'impact d'une décision sur une variable sur le domaine d'une autre. Le graphe obtenu de manière statistique sera construit à partir d'une matrice et les données permettront de construire une de ces matrices. Cette construction permettra alors d'insérer dans le processus des applications capables de sélectionner certaines liaisons. Nous constaterons alors que ce graphe construit avec les meilleures liaisons n'est pas connexe et que, dans une majorité des cas insatisfiables testés, une des composantes connexes contient un noyau insatisfiable. Une variante de cette méthode sera développée. Il s'agit de résoudre des sous-problèmes de tailles réduites, obtenus grâce aux statistiques et dont les variables ont de forts liens. La méthode développée consistera donc à créer de tels graphes avant la recherche afin de détecter de manière précoce l'incohérence ou de créer des contraintes tables. Nous envisagerons enfin d'autres utilisations possibles de ce graphe.

Remerciements

Je tiens à remercier en premier lieu, pour leur aide, mes directeurs de thèse et amis, Christophe Lecoutre et Gilles Audemard, tous les deux Professeurs des universités à l'université d'Artois. J'ai été ravi de travailler en leur compagnie, de profiter de leur bonne humeur et de leurs conseils.

Merci également à Éric Grégoire, Directeur du CRIL et Professeur des universités à l'université d'Artois, pour m'avoir accueilli au sein du CRIL.

Je remercie sincèrement ma femme pour la relecture de certains chapitres de cette thèse.

J'ai aussi une pensée pour Serge Bouc qui m'a enseigné les notions indispensables à tout travail scientifique.

Je tiens à remercier l'ensemble des chercheurs du CRIL et les enseignants du département informatique de l'IUT de Lens pour l'ambiance extraordinaire qu'ils continuent à entretenir.

Enfin je tiens à exprimer ma gratitude aux membres de ma famille pour leur soutien.

Table des matières

Notations **xv**

1 Raisonnement par contraintes **1**
1.1 Réseaux de contraintes . . . 1
1.1.1 Définitions . . . 1
1.2 Résolution . . . 7
1.2.1 Définitions . . . 8
1.2.2 Recherche avec retour arrière (Backtracking) . . . 8
1.2.3 Algorithme de recherche avec retour arrière . . . 10
1.3 Filtrages . . . 11
1.3.1 Cohérence d'arc . . . 12
1.3.2 Cohérences aux bornes . . . 15
1.3.3 Forward checking . . . 20
1.3.4 Singleton arc cohérence . . . 21

2 Squelette d'un mini-Solver **23**
2.1 Sparse set . . . 23
2.1.1 Définition . . . 23
2.1.2 Opérations sur un sparse set . . . 25
2.1.3 Programmation d'un sparse set . . . 28
2.1.4 Diagramme de classe du solveur . . . 29
2.2 Codage Python des domaines des variables . . . 30
2.3 Codage Python des variables . . . 31
2.4 Codage Python des contraintes . . . 31
2.5 Codage Python du problème . . . 32
2.6 Codage Python du solver . . . 33
2.7 Algorithme de filtrage pour la contrainte $x = y + z$ . . . 36

3 Isomorphisme de sous-graphe **41**
3.1 Définitions . . . 41
3.2 Le problème d'isomorphisme de sous-graphe . . . 44
3.3 Modélisation du problème à l'aide de la programmation par contraintes . . . 45
3.4 Évolution des méthodes de résolution . . . 46
3.4.1 Forward Checking . . . 47
3.4.2 GAC . . . 48
3.4.3 LV2002 . . . 48
3.4.4 ILF . . . 49
3.4.5 LAD . . . 50
3.5 Stratégie de dominance de score au voisinage . . . 52
3.5.1 Principe et exactitude . . . 52
3.5.2 Filtrage des contraintes SND . . . 54
3.5.3 Simplification du graphe cible . . . 56

3.6 Étude qualitative 56
3.7 Un algorithme SND incomplet 58
3.8 Résultats expérimentaux 60
3.9 Graphes additionnels 62
3.10 Conclusion 63

4 Contrôle statistique du processus de propagation de contraintes 65
4.1 Passes arc cohérentes 65
4.2 Propagation orientée variable 67
4.3 Contrôle de la propagation 68
4.4 Résultats expérimentaux 76
4.5 Conclusion 77

5 Graphe de contraintes 81
5.1 Graphe de contraintes 81
5.2 Graphe statistique 82
5.2.1 Collecte de données statistiques 82
5.2.2 Construction d'un graphe statistique 83
5.3 Sous-problème inconsistant 86
5.4 Ajout de contraintes tables 90
5.4.1 Principe 90
5.4.2 Résultats expérimentaux 92
5.5 Conclusion 93

Conclusions 93

Bibliographie 101

Index 108

Notations

$\mathrm{M}_n([0,1])$ L'ensemble des matrices à coefficients dans $[0,1]$, page 82

$\llbracket a,b \rrbracket$ L'ensemble des entiers compris entre a et b inclus, page 2

$\mathrm{P}(A)$ L'ensemble des parties de A, page 49

$\mathrm{R}_{|D}$ Le sous-réseau réduit de R pour D, page 7

$\mathit{AllDiff}(\mathrm{X})$ La contrainte AllDifferent pour l'ensemble de variables X, page 5

B L'ensemble $\{0,1\}$, page 4

N L'ensemble des nombres entiers naturels, page 5

N_n L'ensemble des entiers naturels au plus égaux à n, page 21

$\mathrm{O}(n)$ Notation grand O de Landau, page 21

$\mathrm{rel}(C)$ L'ensemble des tuples autorisés pour la contrainte C, page 4

$C_{|D}$ La contrainte induite de C pour le réseau $\mathrm{R}_{|D}$, page 7

$\phi^{\prec}_{AC(1)}$ Le filtrage arc cohérent en une passe, page 64

$\phi^{\prec}_{AC(n)}$ Le filtrage arc cohérent en n passes, page 64

ϕ^{C}_{AC} Le fitrage arc cohérent de la contrainte C, page 63

$|A|$ Le cardinal de l'ensemble A, page 8

Z L'ensemble des nombres entiers relatifs, page 6

$A \cap B$ L'intersection des ensembles A et B, page 6

$A \cup B$ La réunion des ensembles A et B, page 11

$A \times B$ Le produit cartésien de A par B, page 1

$A \mathbin{\mathrm{t}} B$ La réunion disjointe de A et B, page 49

$adj(i)$ L'ensemble des sommets adjacents au sommet i d'un graphe., page 39

$B_V(i_s, i_c)$ La contrainte issue du graphe biparti pour SND, page 52

$deg(i)$ Le degré du sommet i d'un graphe, page 47

$dom(X)$ Le domaine de la variable X, page 7

$g \circ f$ La composition de f par g, page 8

$ILF(k)$ L'algorithme Iterated Labelling Filtering de paramètre k, page 46

MAC L'algorithme Maintaining Arc Consistency, page 14

$scp(C)$ Le scope de la contrainte C, page 8

$sols(\mathrm{R})$ L'ensemble des solutions de R, page 6

$vars(I)$ L'ensemble des variables de l'instanciation I, page 7

$SND_{S,V}$ L'algorithme SND pour un ensemble de fonctions de score S et un ensemble de fonctions de voisinage V, page 50

FC L'algorithme Forward Checking, page 19

Chapitre 1

Raisonnement par contraintes

Ce chapitre comporte trois sections. Dans la première, nous introduirons la notion de réseaux de contraintes en utilisant l'exemple du carré magique pour en illustrer les différents aspects. Nous définirons la notion de variable (en observant les différences avec l'usage en mathématiques) et la notion de contrainte.

Après avoir constaté que des méthodes de résolutions naïves des réseaux de contraintes demandaient trop de temps, nous expliquerons dans la seconde section les mécanismes de recherche dans les réseaux de contraintes : branchement, propagation, retour arrière et apprentissage. Nous verrons également que les choix effectués au cours d'une recherche, appelés heuristiques, avaient des conséquences importantes sur le temps de recherche d'une solution.

Dans la troisième section, nous expliquerons le fonctionnement du processus de filtrage qui permet de réduire le réseau de contraintes à un sous-réseau moins difficile à résoudre et en conservant les solutions (ou une partie) ou l'absence de solution le cas échéant. Nous aborderons enfin une propriété classique des réseaux de contraintes, la cohérence d'arc et des propriétés dérivées comme la cohérence aux bornes et la cohérence d'arc singleton (SAC). Nous en déduirons la construction des filtrages ϕ_{AC}, ϕ_{DC}, ϕ_{ZC}, ϕ_{FC} et ϕ_{SAC}.

1.1 Réseaux de contraintes

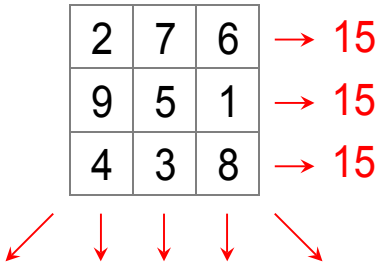
1.1.1 Définitions

Variables

Pour introduire la notion de réseaux de contraintes, nous allons nous intéresser au carré magique dit normal.

Définition 1.1. Pour un entier naturel n, un **carré magique normal** de taille n est un tableau d'entiers naturels de taille $n \times n$ contenant tous les entiers compris entre 1 et n^2 et dont la somme de chaque ligne, chaque colonne et chaque diagonale est égale à une même valeur.

Exemple 1.1. Le tableau suivant est un carré magique de taille 3.



15 15 15 15 15

Remarque 1.1. Imposer au tableau de contenir tous les entiers compris entre 1 et n^2 implique que tous les entiers soient différents (il s'agit donc d'une permutation sur $\llbracket 1, n^2 \rrbracket$).

Considérons maintenant le problème suivant qui consiste à compléter un tableau préremplit afin d'obtenir un carré magique normal :

2	7	
		1
4		

Pour modéliser ce problème, nous pouvons commencer par poser cinq inconnues, également appelées variables, X_1, X_2, X_3, X_4 et X_5 où

$$\forall i \in \llbracket 1, 5 \rrbracket,\ X_i \in \{3, 5, 6, 8, 9\}$$

avec :

2	7	X_1
X_2	X_3	1
4	X_4	X_5

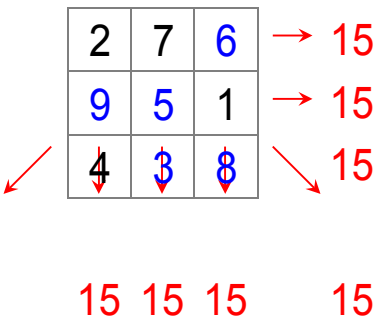
Les équations suivantes doivent alors être vérifiées :

$$\begin{aligned} 2 + 7 + X_1 &= X_2 + X_3 + 1 \\ &= 4 + X_4 + X_5 \\ &= 2 + X_2 + 4 \\ &= 7 + X_3 + X_4 \\ &= X_1 + 1 + X_5 \\ &= X_1 + X_3 + 4 \\ &= 2 + X_3 + X_5 \end{aligned}$$

telles que :

$$\forall i, j \in \llbracket 1, 5 \rrbracket \text{ avec } i = j,\ X_i = X_j.$$

Un exemple de solution pour notre grille préremplie est donnée par :



Il s'agit d'une affectation des cinq variables :

$$X_1 = 6,\ X_2 = 9,\ X_3 = 5,\ X_4 = 3,\ X_5 = 8.$$

Remarque 1.2. En mathématiques, une variable désigne une valeur inconnue appartenant à un ensemble. Pour pouvoir réaliser des opérations, un symbole lui est généralement affecté et l'ensemble auquel elle appartient est fixé. En informatique et plus particulièrement en programmation par contraintes, l'objectif n'est pas de réaliser du calcul littéral mais d'affecter valeur à une variable. Ainsi, l'ensemble de définition varie au cours du temps.

Formellement, si l'ensemble de définition n'est plus le même, alors la variable n'est plus la même. Cependant, par commodité, le nom de la variable est conservé.

C'est pourquoi en mathématiques, l'ensemble de définition d'une variable n'est pas rappelé alors qu'en informatique il est indispensable d'avoir des méthodes permettant de le retrouver. Ces considérations motivent la définition suivante.

Définition 1.2. Nous appellerons variable un couple (X, D), telle que X est un mot dans un alphabet quelconque (au sens de la théorie des langage) et D est un ensemble. L'ensemble D sera appelé **domaine** de la variable X.

Notation. Pour une variable (X, D) et lorsque le contexte sera clair, nous noterons simplement X et non (X, D) la variable. Nous noterons $dom(X)$ le domaine D de la variable X.

Exemple 1.2. Dans notre exemple, $\forall i \in [\![1, 5]\!]$, $dom(X_i) = \{3, 5, 6, 8, 9\}$

Pour trouver une solution ou toutes les solutions à notre problème, nous pouvons essayer toutes les affectations de variables possibles. La figure 1.1 illustre la méthode d'affectations successives. Il y a donc 5^5 cas à étudier sur notre exemple. Un ordinateur moderne avec un processeur i7 a une puissance d'environ 200 gigaFLOPS. Il permet de réaliser 2×10^{11} opérations en virgule flottante par seconde. Avec cette puissance, les 5^5 cas sont étudiés en environ 15 nanosecondes. Cependant, un carré magique comportant

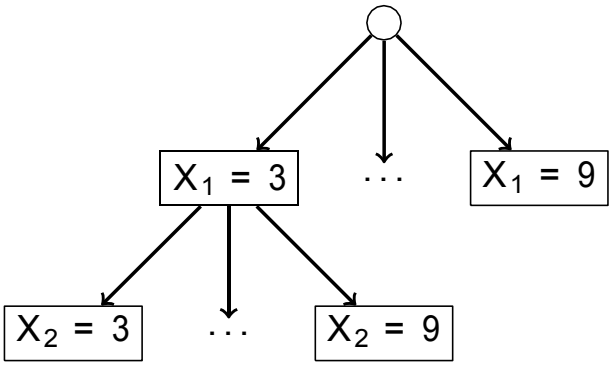


Figure 1.1 : *Arbre de recherche.*

40 variables nécessiterait 5^{40} opérations, ce qui demanderait avec le même ordinateur environ 1,4 milliards de siècles. Il est donc indispensable de réfléchir à des méthodes de résolution moins naïves. Par exemple, il est possible d'accélérer le calcul en identifiant des symétries au niveau du problème ou en simplifiant le problème par déduction de nouvelles informations. Définissons maintenant une structure pour la résolution de ce type de problèmes.

Au cours de la recherche de solutions, des valeurs vont être éliminées. Les domaines des variables sont donc amenés à évoluer au cours du temps. Nous aurons donc besoin des notations suivantes.

Notation. Soit X une variable et D un ensemble tel que $D \subseteq dom(X)$. Nous noterons $X_{|_D}$ la variable (X, D).

Remarque 1.3. En programmation par contraintes, le domaine des variables est constamment modifié. Une classe est généralement créée pour cet objet comme nous pourrons le voir dans le chapitre 2 page 31. En fait, au cours de la recherche ce sont uniquement les domaines des variables qui varient.

Définition 1.3. Nous dirons qu'une **variable est fixée** si son domaine est un singleton. Nous dirons que nous **assignons une variable** X à la valeur $a \in dom(X)$ lorsque X est remplacée par $X_{|_{\{a\}}}$. La nouvelle variable, nommée X par abus, aura alors un domaine égal à $\{a\}$.

Contraintes

Notation. Nous noterons l'ensemble $\{0, 1\}$.

Définition 1.4. Soit $\{X_1, \ldots, X_n\}$ un ensemble de n variables ordonné par une relation d'ordre strict $<$ tel que $X_1 < X_2 < \ < X_n$. Une **contrainte** C sur ces variables est une application de $\Pi_{i=1}^{n} dom(X_i)$ dans B. Chaque élément de $\Pi_{i=1}^{n} dom(X_i)$ est appelé **tuple sur** C.
L'ensemble ordonné $\{X_1, \ldots, X_n\}$ sera appelé **portée (scope)** de c et nous noterons

$$scp(C) = \{X_1, \ldots, X_n\}.$$

Nous appellerons **arité** le cardinal de la portée. Une contrainte est dite :
- **unaire** si son arité est égale à 1
- **binaire** si son arité est égale à 2
- **ternaire** si son arité est égale à 3
- **non-binaire** si son arité est au moins égale à 3.

Remarque 1.4. Lorsque rien ne sera précisé, l'ordre d'un scope donné sous la forme $\{X_1, \ldots, X_n\}$ sera implicitement $X_1 < X_2 < \ < X_n$.

Définition 1.5. Soit C une contrainte telle que $scp(C) = \{X_1, \ldots, X_n\}$. Un tuple $I \in \Pi_{i=1}^{n} dom(X_i)$ sera dit **autorisé** par C si $C(I) = 1$.
Nous noterons $rel(C)$ l'ensemble des tuples autorisés pour une contrainte C.

Remarque 1.5. Une contrainte permet donc de décider si un tuple est autorisé ou non.

Remarque 1.6. Il existe principalement deux manières de définir une contrainte. La première est dite en **intention**, et consiste à représenter la sémantique d'une contrainte par une expression booléenne (prédicat) intégrant les opérateurs classiques arithmétiques, relationnels et logiques. Par exemple, la contrainte $X_2 + X_3 + 1 = 15$ est une contrainte en intention.
La seconde représentation, en **extension**, revient à lister les tuples autorisés ou interdites pour une contrainte. Par exemple, la contrainte relative à l'équation $X_2 + X_3 + 1 = 15$ pourrait être définie de cette manière en donnant son scope, ici $\{X_2, X_3\}$, et en énumérant les tuples autorisés :

$$(5, 9), (6, 8), (8, 6), (9, 5).$$

Dans la suite, nous noterons simplement

$$(X_2, X_3) \in \{(5, 9), (6, 8), (8, 6), (9, 5)\}$$

ce qui nous donnera directement le scope et l'ordre dans ce scope.
En pratique, il peut être intéressant ou nécessaire d'utiliser une forme plutôt qu'une autre. Les deux exemples qui suivent vont illustrer ce propos.

Exemple 1.3. Considérons trois variables X, Y et Z dont les domaines sont

$$dom(X) = dom(Y) = dom(Z) = \llbracket 2, 100 \rrbracket.$$

Considérons la contrainte C définie par $X^2 \times Y^3 + 4 = Z$. La contrainte C n'autorise que deux tuples : (2, 3, 6) et (3, 2, 76). Il est donc plus simple de définir C en extension. Et par ailleurs, il sera sans doute plus efficace pour un système de résolution de manipuler deux tuples à la place d'une équation complexe.

Exemple 1.4. Considérons deux variables X et Y dont les domaines sont

$$dom(X) = dom(Y) = \llbracket 1, 100 \rrbracket.$$

Considérons la contrainte C définie par $X < Y$. Le nombre de tuples autorisés est $\frac{100 \times 99}{2} = 4950$. Il est possible de définir la contrainte en utilisant les tuples interdits mais dans ce cas il y a $4950 + 100 = 5050$ tuples. De manière générale, pour des domaines égaux à $\llbracket 1, n \rrbracket$ ($n \in \mathbb{N}$), le nombre de tuples autorisés serait $\frac{n \times (n-1)}{2} \underset{+\infty}{\sim} n^2$. Dans ce cas, il est plus intéressant de définir la contrainte en intention car les systèmes de résolution sont capables de traiter efficacement ce type de contraintes.

Définition 1.6. Une contrainte est dite **globale** si sa sémantique est applicable à un nombre quelconque de variables (voir [116, 7]).

Exemple 1.5. Dans notre exemple du carré magique normal, la contrainte principale impose que toutes les valeurs soient différentes. Cette contrainte est une contrainte globale appelée *AllDiff* pour *all different*.

Définition 1.7. Soit $X_1, \ldots, X_n$ n variables. La contrainte *AllDiff* est définie par :

$$AllDiff(X_1, \ldots, X_n) : \begin{cases} \prod_{i=1}^{n} dom(X_i) & \longrightarrow \quad \mathbb{B} \\ (x_1, \ldots, x_n) & \longrightarrow \quad \begin{array}{l} 1 \text{ si } \forall i, j \in \llbracket 1, n \rrbracket \text{ avec } i = j,\ x_i = x_j \\ 0 \text{ sinon.} \end{array} \end{cases}$$

La contrainte *AllDiff* est très importante en programmation par contraintes et particulièrement étudiée depuis longtemps et pour des domaines d'applications très variés [91, 9, 88, 5, 93, 57, 115, 110, 108, 55, 94, 64, 8]. Cela semble naturel car tout problème comportant une affectation d'objet (affectation de reines dans le problème des 8-reines, de nombres différents dans le problème du carré magique, de salles en plannification, voir [69]) nécessitera une contrainte *AllDiff* dans le modèle. La vérification de cette contrainte peut être très exigeante en espace mémoire et en temps, c'est pourquoi des méthodes de décomposition sont étudiées [16].

Réseau de contraintes

Définition 1.8. Nous appellerons **réseau de contraintes** un couple (X, C) de deux ensembles finis où :

- X est un ensemble de variables strictement ordonné,
- C est un ensemble de contraintes dont les scopes sont des sous-ensembles de X ordonnés par l'ordre induit de X.

Nous appellerons **réseau de contraintes binaire** un réseau de contraintes dont toutes les contraintes sont binaires.

Remarque 1.7. Comme dans le cas des contraintes, si rien n'est précisé, l'ordre d'un ensemble de variables donné par $\{X_1, \ldots, X_n\}$ est $X_1 < \ < X_n$.

Remarque 1.8. Dans toute la suite,

— (X, C) désignera un réseau de contraintes avec X = $(X_1, \ldots, X_n)$,
— s'il n'y a pas d'ambiguïté, D désignera le produit cartésien des domaines des va-riables selon l'ordre imposé sur X. Pour l'ordre $X_1 < \ < X_n$, nous aurons D = $\Pi_{i=1}^{n} dom(X_i)$,
— les domaines des variables seront tous des sous-ensembles finis de Z.

Exemple 1.6. Reprenons le carré magique de la page 2. Nous avons :
— X = $\{X_1, X_2, X_3, X_4, X_5\}$ avec $\forall i \in$J1, 5K, $dom(X_i) = \{3, 5, 6, 8, 9\}$.
— C = $\{C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$.
Nous avons $C_0 = AllDiff(X_1, X_2, X_3, X_4, X_5)$. La contrainte C_1 peut être définie de la manière suivante :

$$C_1 : \begin{array}{ccc} dom(X_1) \times dom(X_2) \times dom(X_3) & \longrightarrow & \mathbb{B} \\ (x_1, x_2, x_3) & \longrightarrow & \begin{cases} 1 & si\ 2 + 7 + x_1 = x_2 + x_3 + 1 \\ 0 & sinon. \end{cases} \end{array}$$

La représentation en intention de la contrainte C_1 est $2 + 7 + X_1 = X_2 + X_3 + 1$. Nous avons alors $scp(C_1) = \{X_1, X_2, X_3\}$.
De même :
— $C_2 : 2 + 7 + X_1 = 4 + X_4 + X_5$ et $scp(C_2) = \{X_1, X_4, X_5\}$.
— $C_3 : 2 + 7 + X_1 = 2 + X_2 + 4$ et $scp(C_3) = \{X_1, X_2\}$.
— $C_4 : 2 + 7 + X_1 = 7 + X_3 + X_4$ et $scp(C_4) = \{X_1, X_3, X_4\}$.
— $C_5 : 2 + 7 + X_1 = X_1 + 1 + X_5$ et $scp(C_5) = \{X_1, X_5\}$.
— $C_6 : 2 + 7 + X_1 = X_1 + X_3 + 4$ et $scp(C_6) = \{X_1, X_3\}$.
— $C_7 : 2 + 7 + X_1 = 2 + X_3 + X_5$ et $scp(C_7) = \{X_1, X_3, X_5\}$.

Notation. Soit C une contrainte d'un réseau de contraintes (X, C) dont le scope est $\{X_{i_1}, \ldots, X_{i_p}\}$. Nous noterons $\pi_{scp(C)}$ la projection naturelle de D sur $\Pi_{k=1}^{p} D_{X_{i_k}}$

Définition 1.9. Soit R = (X, C) un réseau de contraintes. Nous appellerons **solution de ce réseau de containtes** un élément $I \in$D tel que

$$\forall C \in C,\ C \circ \pi_{scp(C)}(I) = 1.$$

L'ensemble des solutions de R sera noté $sols$(R).

Remarque 1.9. Une solution est donc un élément vérifiant toutes les contraintes.

L'ensemble des solutions d'un réseau de contraintes est ainsi donné par :

$$sols(\mathrm{R}) = \bigcap_{C \in \mathrm{C}} \left(C \circ \pi_{scp(C)} \right)^{-1} (1).$$

Définition 1.10. Nous dirons qu'un réseau est **consistant** s'il possède une solution. Il sera dit **inconsistant** dans le cas contraire.

Définition 1.11. Nous dirons qu'un réseau de contraintes (X^0, C^0) est un **sous-réseau** de (X, C) s'il existe
— $X_0 \subseteq$X,
— $(D_X)_{X \in \mathrm{X}} \subseteq$D,
— $C_0 \subseteq$C
tel que
— $X^0 = \ X\ , X \in X_0\ ,$
— $C_t = \{C\ |_{D^X}\ , C \in C_0\}$ et la relation d'ordre sur C_t est induite par celle de C.
$|_{\Pi^{X \in scp(C)} D^X}$

Lorsque $X_0 = X$ et $C^0 = C$, nous dirons que (X^0, C^0) est un **sous-réseau réduit** de (X, C).

Notation. Nous noterons $R_{|_D}$ le sous-réseau réduit de $R = (X, C)$ pour la collection d'ensembles $D = (D_X)_{X \in X} \subseteq D$.

Par abus de notation et lorsque cela ne prêtera pas à confusion, nous noterons $C_{|_D}$ la contrainte induite de C pour le réseau $R_{|_D}$.

Exemple 1.7. Considérons le réseau de contraintes (X, C) défini de la manière suivante :
— $X = \{X, Y\}$ avec $dom(X) = dom(Y) = [\![1, 3]\!]$ et avec l'ordre $X < Y$,
— $C = \{C\}$ avec $C : X = Y$.
Soit $D = [\![1, 2]\!]^2$. Le sous-réseau réduit $R_{|_D}$ a un ensemble de variables constitué de deux éléments $X_{|_{[\![1,2]\!]}}$ et $Y_{|_{[\![1,2]\!]}}$, c'est-à-dire $(X, [\![1, 2]\!])$ et $(Y, [\![1, 2]\!])$. L'ensemble des contraintes de $R_{|_D}$ a un unique élément $C_{|_D}$. Nous avons

$$C_{|_D} : \begin{array}{ccc} [\![1,2]\!]^2 & \longrightarrow & \{0,1\} \\ (x,y) & \longrightarrow & \begin{cases} 1 & \text{si } x = y \\ 0 & \text{sinon.} \end{cases} \end{array}$$

Remarque 1.10. Une contrainte peut être considérée comme un sous-réseau en ne gardant du réseau initial que cette contrainte. Ainsi résoudre un réseau de contraintes revient à résoudre conjointement ces sous-réseaux.

La définition suivante va introduire un autre type de sous-réseau important.

Définition 1.12. Nous dirons qu'un réseau (X^0, C^0) est un **sous-réseau induit** de (X, C) si
— $X^0 \subseteq X$,
— $C^0 = \{C \in C \mid scp(C) \in X^0\}$.

Exemple 1.8. Considérons le réseau de contraintes (X, C) défini de la manière suivante :
— $X = \{X, Y, Z\}$ avec $dom(X) = dom(Y) = dom(Z) = [\![1, 3]\!]$ et avec l'ordre $X < Y < Z$,
— $C = \{C_1, C_2\}$ avec $C_1 : X = Y$ et $C_2 : X + Y + Z = 1$.
Le réseau induit par les variables X et Y est donné par :
— $X^0 = \{X, Y\}$ avec $dom(X) = dom(Y) = [\![1, 3]\!]$ et avec l'ordre $X < Y$,
— $C = \{C_1\}$ avec $C_1 : X = Y$.

L'intérêt des sous-réseaux induits est que si un réseau de contraintes a un sous-réseau induit inconsistant alors il est inconsistant. La recherche de ces sous réseaux peut donc permettre de réduire sensiblement le temps de recherche. Plus le sous-problème est petit, plus le bénéfice est important.

Définition 1.13. Nous dirons qu'un sous-réseau induit N d'un réseau de contrainte inconsistant est un **noyau inconsistant** s'il est inconsistant et s'il ne contient pas de sous-réseau induit inconsistant différent de N.

La recherche de noyaux inconsistants est très étudiée [4, 21, 78] étant donné le gain potentiel. Cependant, en général, leur détection est difficile. Une méthode sera donnée au chapitre 5.

1.2 Résolution

Pour trouver les solutions d'un réseau de contraintes, nous avons déjà vu qu'il fallait explorer l'espace de recherche, par exemple en utilisant une structure d'arbre (figure 1.1).

Cette exploration peut s'avérer extrêmement coûteuse. Afin de palier ce problème, une technique usuelle consiste à remplacer progressivement le réseau pendant l'exploration par des réseaux plus simples à résoudre. De nombreuses stratégies existent et s'appuient souvent sur une technique de retour arrière (backtracking). Dans cette partie, nous allons expliquer les mécanismes généraux des algorithmes de recherche avec retour arrière.

1.2.1 Définitions

Pour décrire un algorithme arborescent avec retour arrière, nous aurons besoin tout d'abord de quelques définitions. Comme nous l'avons déjà évoqué, le réseau initial dont nous recherchons les solutions est modifié au cours de la recherche. Ainsi, de nombreuses définitions font apparaître le réseau initial et un de ses sous-réseaux réduits.

Définition 1.14. Considérons un réseau de contraintes R = (X, C). Posons X = $\{X_1, \ldots, X_n\}$. — Une **instanciation** I d'un ensemble de variables $\{X_{i_1}, \ldots, X_{i_k}\} \subseteq$X pour le réseau
R est un ensemble $\{X_{i_1}|_{\{a_1\}}, \ldots, X_{i_k}|_{\{a_k\}}\}$ tel que

$$\forall i \in \text{J1}, k\text{K}, \; a_k \in dom(X_{i_k}).$$

Nous noterons alors $vars(I) = \{X_{i_1}, \ldots, X_{i_k}\}$

— Soit $D = (D_X)_{X \in \text{X}} \subseteq \Pi_{i=1}^{n} D_{X_i}$. Une instanciation I de R est **valide** pour $R_{|_D}$ si $\forall (X, a) \in I, \; a \in D_X$.

— Une instanciation I pour R est **complète** si $vars(I) =$ X. Elle sera dite **incomplète** sinon.

— Nous appellerons **contrainte couverte par l'instanciation** I une contrainte C telle que $scp(C) \subseteq vars(I)$.

— Pour une instanciation I de R, nous noterons $R_{|_I}$ le sous-réseau $R_{|_{\Pi_{i=1}^{n} A_i}}$ où

$$A_i = \begin{cases} \{a_i\} & \text{si } i \in \{i_1, \ldots, i_k\} \\ dom(X_i) & \text{sinon} \end{cases}$$

— Nous noterons $R_{|_{X_{i_1} = a_1, \ldots, X_{i_k} = a_k}}$ le réseau $R_{|_I}$ pour l'instanciation $I = \{X_{i_1}|_{\{a_1\}}, \ldots, X_{i_k}|_{\{a_k\}}\}$ — Nous dirons qu'une contrainte $C \in$C est **violée** pour une instanciation I de R si $C \circ \pi_{scp(C)}(I) = 0$.

— Soit I une instanciation pour R avec $r := |vars(I)|$. Nous dirons alors que le **niveau** pour $R_{|_I}$ est r.

1.2.2 Recherche avec retour arrière (Backtracking)

Un algorithme de recherche avec retour arrière peut être assimilé à un parcours en profondeur d'abord. Il s'agit de trouver une solution en complétant progressivement une instanciation et en vérifiant à chaque étape qu'aucune contrainte n'est violée. Dans le cas contraire, on revient à l'étape précédente, ce qui justifie le terme de recherche avec retour arrière . L'intérêt de cette méthode réside dans le fait de ne pas explorer des parties entières de l'arbre de recherche. Si un retour arrière est effectué assez haut dans l'arbre de recherche, le gain peut être énorme.

Une recherche arborescente (avec retour arrière) se compose principalement en quatre parties :

— **Branchement (branching) :** La manière dont les décisions sont prises pour ins-tancier les variables et trouver une solution.

— **Propagation :** Il s'agit du filtrage utilisé à chaque étape afin de réduire l'espace de recherche. Le niveau de filtrage a un impact sur le temps de calcul.

— **Retour arrière (backtracking) :**
La manière dont on revient à une étape antérieure lorsqu'une incohérence est détectée.
— **Apprentissage (learning) :** Il s'agit de l'ensemble des informations apprises durant la recherche.

Reprenons l'exemple du carré magique de la page 2 et les notations de l'exemple 1.6 de la page 6. Rappelons la structure du problème.

2	7	X_1
X_2	X_3	1
4	X_4	X_5

Une recherche avec retour arrière est alors illustrée sur la figure 1.2.

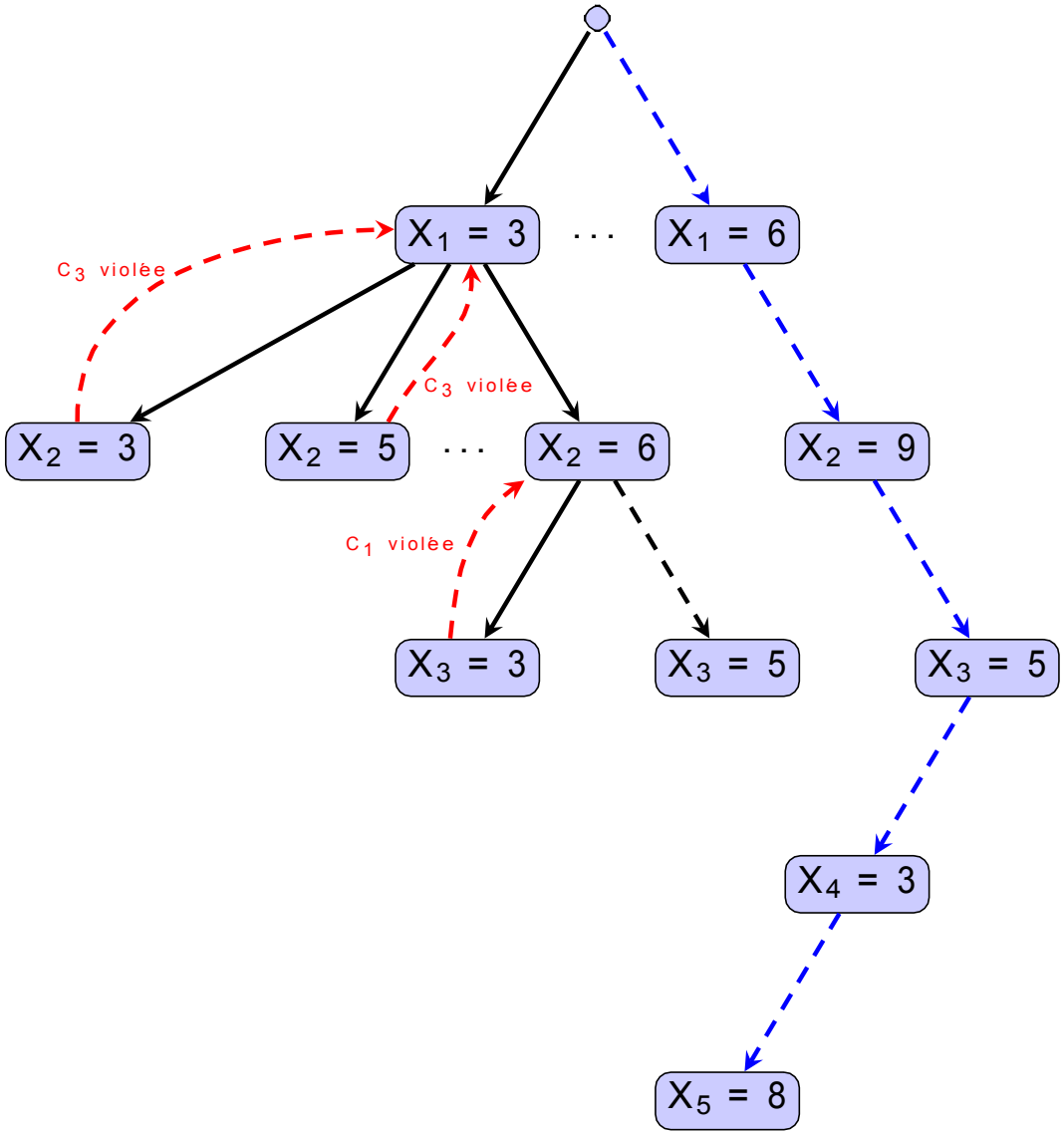


Figure 1.2 : *Illustration de l'algorithme BT sur l'exemple du carré magique de la page 2.*

Premières étapes de la recherche :

— Nous commençons par choisir une variable non affectée, ici X_1.

— Nous choisissons un élément du domaine courant de X_1. Nous choisissons ici la valeur 3. La variable X_1 est affectée à 3, ce qui signifie que $dom(X_1) = \{3\}$.
— Nous n'avons pas d'instanciation complète et aucune contrainte n'est violée, nous pouvons continuer.
— Nous choisissons une variable non affectée, ici X_2.
— Nous choisissons un élément du domaine courant de X_2. Nous choisissons ici la valeur 3. Ainsi, $dom(X_2) = \{3\}$.
— La contrainte C_3 est violée puisque $2 + 7 + 3 = 2 + 3 + 4$. Il faut alors effectuer un retour arrière, c'est-à-dire, restaurer le domaine de X_2 à ce qu'il était au niveau précédent et choisir une autre valeur pour X_2.
— Nous ferons un retour arrière jusqu'à ce que nous choisissions la valeur 6 pour affecter X_2. La contrainte C_3 n'est alors plus violée.
— Nous choisissons une variable non affectée, ici X_3.
— Nous choisissons un élément du domaine courant de X_3. Nous choisissons ici la valeur 3. Ainsi, $dom(X_3) = \{3\}$.
— La contrainte C_1 est alors violée puisque $2 + 7 + 3 = 6 + 3 + 1$. Il faut alors effectuer un retour arrière, etc.

Remarque 1.11. Nous voyons qu'à chaque étape, un choix nous est donné. Pour gérer l'ensemble de ces choix nous faisons généralement appel à une heuristique. C'est une composante importante des systèmes de résolution qui influence de manière substancielle l'efficacité de la résolution d'un problème.

Définition 1.15. Nous appellerons **heuristique de choix de variables** l'ordre dans lequel nous choisissons les variables dans un algorithme de recherche arborescente.

Nous appellerons **heuristique de choix de valeurs** l'ordre dans lequel nous choisissons les valeurs des domaines des variables dans un algorithme de recherche arborescente.

1.2.3 Algorithme de recherche avec retour arrière

Définition 1.16. Nous appellerons **filtrage** une application ϕ qui à tout réseau de contraintes R associe un sous-réseau réduit de R noté ϕ(R).

Nous dirons que le filtrage ϕ sera appliqué lorsqu'un réseau R sera remplacé par ϕ(R). Le filtrage sera dit :
— **prudent** si quelque soit le réseau, ϕ conserve toutes les solutions,
— **équisatisfiable** si quelque soit le réseau, ϕ conserve au moins une solution (lorsqu'il en existe une),
— **arbitraire** s'il existe un réseau R consistant tel que ϕ(R) est inconsistant.

Remarque 1.12. On s'intéresse principalement aux filtrages prudents ou équisatisfiables.

Remarque 1.13. Des exemples de processus de filtrage seront donnés dans la section 1.3.

Définition 1.17. Pour un filtrage ϕ, nous appellerons **ϕ-recherche avec retour arrière** une recherche avec retour arrière pour laquelle le filtrage ϕ est appliqué après chaque décision.

Notation. Pour un réseau de contraintes R, nous noterons R = $\perp$ si un domaine d'une des variables du réseau R est vide. Dans le cas contraire, nous noterons R = $\perp$.

C'est un abus de notation puisqu'il s'agit d'une égalité de classes d'équivalences et non de réseaux de contraintes.

L'algorithme suivant donne une ϕ-recherche binaire de manière récursive. On suppose que ϕ représente un filtrage qui vérifie au minimum que toutes les contraintes couvertes

soient satisfaites. Dans le cas contraire, $\phi(R)$ retourne $\perp$.

Algorithme 1 : ϕ-recherche

Entrée(s): $R = (X, C)$: réseau de contraintes
Sortie(s): true ssi R est satisfiable

```
1: R ← φ(R)
2: si R = ⊥ alors
3:    retourner false
4: fin si
5: si ∀x∈vars(R), |dom(x)| = 1 alors 6:
      afficher la solution
7:    retourner true
8: fin si
9: selectionner une paire (x, a) dans R telle que |dom(x)| > 1
10: retourner φ-recherche(R|x = a) ∨ φ-recherche(R|x = a)
```

La figure 1.3 illustre une ϕ-recherche. Nous y voyons en particulier que cette arbre de recherche est binaire. Il ne peut y avoir, en effet, que deux possibilités : $X = a$ ou $X = a$. En pratique cette recherche est beaucoup plus efficace que la recherche sur un arbre n-aire (figure 1.2).

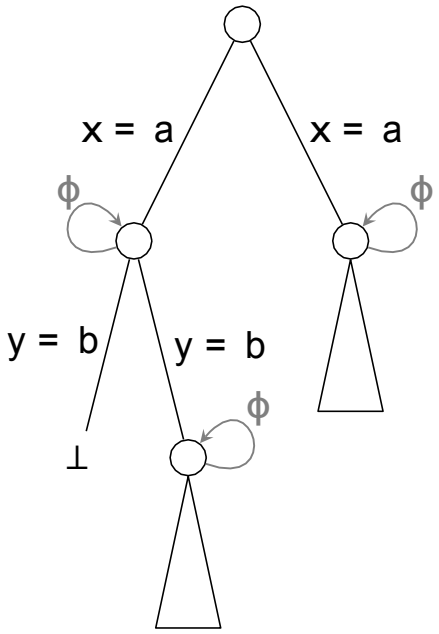


Figure 1.3 : *ϕ-recherche.*

Le filtrage est un aspect important d'un algorithme de recherche car il va lui permettre de réduire l'espace de recherche et rendre l'exploration de l'arbre de recherche plus efficace.

1.3 Filtrages

Comme nous l'avons vu dans la partie précédente, les filtrages représentent une partie essentielle des techniques de recherche de solutions dans les réseaux de contraintes. Le filtrage peut se produire avant de commencer la recherche ou en cours de recherche après une affectation. Dans cette partie, nous allons définir des filtrages classiques de la théorie des réseaux de contraintes. Ils sont très utilisés et servent de références pour les comparaisons. Il est parfois possible d'établir qu'un filtrage est strictement meilleur qu'un autre mais dans la plupart des cas l'efficacité d'un filtrage dépend du type de problème.

1.3.1 Cohérence d'arc

Le filtrage par cohérence d'arc est très populaire. Cela tient au fait que c'est une des premières méthodes qui vient à l'esprit et qu'elle est efficace. En effet, il s'agit de vérifier que les valeurs des domaines sont compatibles avec les contraintes. La communauté se penche sur ce filtrage depuis longtemps, tout d'abord avec [47, 48, 33] et plus récemment avec [3, 95, 75, 19]. Ce filtrage est présent dans la grande majorité des solveurs de réseaux de contraintes. Généralement, il est associé à d'autres filtrages comme nous pourrons le voir au chapitre 3. Les méthodes incomplètes de filtrage par cohérence d'arc sont également beaucoup étudiées (voir chapitre 4 ou [79]).

Définition 1.18. Nous dirons qu'une contrainte C avec $scp(C) = \{X_1, \ldots, X_r\}$, est **arc cohérente** si $\forall i_0 \in J1, rK$, $\forall v \in dom(X_{i_0})$:

$$\exists (v_1, \ldots, v_n) \in \Pi_{i \in J1,rK \setminus \{i_0\}} dom(X_i) \text{ tel que } C(v_1, \ldots, v_{i_0-1}, v, v_{i_0+1}, \ldots, v_n) = 1.$$

Un réseau de contraintes est dit **arc cohérent (AC)** si toutes ses contraintes sont arc cohérentes. On notera qu'historiquement dans la littérature, lorsque les contraintes sont non binaires on parle d'arc cohérence généralisée (GAC, pour generalized arc consistency).

Remarque 1.14. Une solution représente nécessairement un sous-réseau réduit arc cohérent.

Proposition 1. Soit R = (X, C) et D et D^0 deux sous ensembles de D tels que $R|_D$ et $R|_{D^0}$ soient arc cohérents alors $R|_{D \cup D^0}$ est arc cohérent.

La démonstration de cette proposition est simple et nous permet d'établir la propositio n suivante.

Proposition 2. Tout réseau de contraintes R possède un sous-réseau réduit arc cohérent maximum pour l'ordre naturel d'inclusion sur D

Démonstration. Considérons l'ensemble.

$$E = \bigcup \{ D \subseteq D \mid R|_D \text{ est arc cohérent} \}.$$

Le sous-réseau réduit maximum est alors $R|_E$. □

Remarque 1.15. Le sous-réseau arc cohérent maximum vérifie toujours :

$$R|_{sols(R)} \subseteq R|_E.$$

Définition 1.19. Nous appellerons **filtrage par arc cohérence** et nous noterons ϕ_{AC} le filtrage qui à un réseau de contraintes R associe le plus grand sous-réseau réduit arc cohérent de R pour l'ordre naturel d'inclusion sur D.

Remarque 1.16. En pratique, qu'un domaine soit vide ou que tous les domaines soient vides ne change rien. Ces cas sont traités de manière identique.

Pour trouver le sous-réseau réduit arc cohérent maximum, l'algorithme supprime récursivement des domaines les valeurs qui ne vérifient pas l'arc cohérence afin de rendre le réseau arc cohérent. Il agit contrainte par contrainte jusqu'à l'obtention d'un point fixe. Pour chaque contrainte, il faut vérifier que chaque valeur des domaines possède bien un support. C'est-à-dire que, pour ne pas être supprimée, une valeur doit avoir un support dans toutes les contraintes. Le mécanisme de filtrage et l'obtention du point fixe seront détaillés au chapitre 4.

Remarque 1.17. Le filtrage par arc cohérence considère chaque contrainte comme un sous-problème. Il s'agit du filtrage maximum lorsqu'on examine les contraintes une à une.

Exemple 1.9. Modifions un peu le problème de la page 2 pour obtenir le réseau de contraintes R correspondant au carré magique suivant :

2	7	
		1
		8

Les variables sont données alors par :

2	7	X_1
X_2	X_3	1
X_4	X_5	8

avec tous les domaines égaux à $\{3, 4, 5, 6, 9\}$. Les contraintes du problèmes seront :

$$
\begin{aligned}
C_0 &= AllDiff(X_1, X_2, X_3, X_4, X_5)\\
C_1 &: 2 + 7 + X_1 = 15\\
C_2 &: X_2 + X_3 + 1 = 15\\
C_3 &: X_4 + X_5 + 8 = 15\\
C_4 &: 2 + X_2 + X_4 = 15\\
C_5 &: 7 + X_3 + X_5 = 15\\
C_6 &: X_1 + 1 + 8 = 15\\
C_7 &: 2 + X_3 + 8 = 15\\
C_8 &: X_1 + X_4 + X_5 = 15.
\end{aligned}
$$

En effet, la somme des entiers compris entre 1 et 9 est égale à $\frac{9\times 10}{2} = 45$. Ainsi, la somme de chaque ligne, diagonale ou colonne est 15.

Les variables du scope du la contrainte C_0 seront étudiées au fur et à mesure selon un ordre défini préalablement (heuristique de variable). Dans cet exemple, l'ordre sera donné par $X_1 < X_2 < X_3 < X_4 < X_5$. L'heuristique de choix de valeur sera donné par l'ordre naturel sur N.

Nous commençons donc la recherche avec la variable X_1 et la valeur 3. Immédiatement, la contrainte C_1 est violée et la valeur 3 est supprimée du domaine de X_1. De même, les valeurs 4 et 5 sont supprimées du domaine de X_1. La variable X_1 est donc affectée à 6.

2	7	6	→ 15
X_2	X_3	1	
X_4	X_5	8	
		↓	
		15	

À ce stade, aucune contrainte n'est violée car $2 + 7 + 6 = 6 + 1 + 8$. De plus,

$dom(X_1) = \{6\}$

et

$$dom(X_2) = dom(X_3) = dom(X_4) = dom(X_5) = \{3, 4, 5, 6, 9\}.$$

Commençons par appliquer le filtrage par cohérence d'arc à la contrainte C_0 La variable X_1 n'a que la 6 dans son domaine. Cette valeur a un support donc n'est pas filtrée (par exemple $X_1 = 6, X_2 = 3, , X_3 = 4, X_4 = 5$ et $X_5 = 9$). Nous passons alors à la variable X_2. Dans le domaine de X_2 les valeurs 3, 4, 5 et 9 ont des supports pour la contrainte C_0. En revanche, la valeur 6 n'a pas de support. Elle est donc supprimée du domaine de X_2. De même, cette valeur est supprimée des domaines de X_3, X_4 et X_5. Le domaines sont donc :

$$dom(X_1) = \{6\}$$

et

$$dom(X_2) = dom(X_3) = dom(X_4) = dom(X_5) = \{3, 4, 5, 9\}.$$

Appliquons le filtrage par arc cohérence à la contrainte $C_2 : X_2 + X_3 + 1 = 15$.

2	7	6
X_2	X_3	1
X_4	X_5	8

La valeur 3 peut être supprimée du domaine de X_2 car aucune valeur du domaine de X_3 ne permettra d'avoir

$$3 + X_3 + 1 = 15.$$

Cette valeur n'a pas de support, elle peut donc être supprimée du domaine de X_2. De même, par arc cohérence, la valeur 4 peut être supprimée du domaine de X_2. Par symétrie, nous pouvons appliquer le même raisonnement à la variable X_3. Ainsi, après filtrage par arc cohérence sur la contrainte $X_2 + X_3 + 1 = 15$, les domaines sont

$$dom(X_1) = \{6\}$$

$$dom(X_2) = dom(X_3) = \{5, 9\}.$$

et

$$dom(X_4) = dom(X_5) = \{3, 4, 5, 9\}.$$

Le filtrage est terminé lorsque toutes les contraintes vérifient les conditions de cohérence d'arc. Parfois la modification d'un domaine à cause d'une contrainte implique de revenir sur une contrainte déjà étudiée. Nous cherchons un point fixe. Ce point sera spécifiquement étudié dans le chapitre 4. Ce filtrage permet parfois de montrer rapidement qu'une affectation n'aboutira pas et d'éviter ainsi de parcourir inutilement une partie de l'arbre de recherche. Dans notre exemple, la contrainte C_3 permet de réduire les domaines de X_4 et X_5 à $\{3, 4\}$.

2	7	6
5, 9	5, 9	1
3, 4	3, 4	8

Enfin, la contrainte C_4 permet d'obtenir $dom(X_2) = 9$ et $dom(X_4) = 4$. La contrainte C_5 permet d'obtenir $dom(X_3) = 5$ et $dom(X_5) = 3$.

Remarque 1.18. L'algorithme de ϕ_{AC}-recherche maintient la propriété d'arc cohérence tout au long de la recherche. C'est pourquoi il est appelé algorithme *MAC* pour Maintaining Arc Consistency [100]. L'algorithme est traité de manière exhaustive dans [101] et des variantes sont données dans [50, 85]. De nombreux algorithmes ont été proposés pour établir la propriété de cohérence d'arc : AC1 [76], AC3 [76], AC4 [83], AC5 [113], AC6 [11], AC7 [15], AC8 [26], AC2001 [18] et AC3^{rm}[72]. Le tableau 1.1 compare les complexités de ces différents algorithmes sur des réseaux binaires.

Algorithme	Complexité en temps	Complexité en espace
AC1 [76]	$O(end^2)$	$O(n^2)$
AC3 [76]	$O(ed^3)$	$O(e + nd)$
AC4 [83]	$O(ed^2)$	$O(ed^2)$
AC5 [113]	$O(ed^2)$	$O(ed)$
AC6 [11]	$O(ed^2)$	$O(ed)$
AC7 [15]	$O(ed^2)$	$O(ed)$
AC8 [26]	$O(ed^2)$	$O(n)$
AC2001 [18]	$O(ed^2)$	$O(ed)$
AC3^{rm}[72]	$O(end^3)$	$O(ed)$
AC3^{bit}[73]	$O(end^3)$	$O(ed)$

Table 1.1: *Complexité de différents algorithmes établissant la cohérence d'arc pour des réseaux binaires comportant n variables, e contraintes et dont les tailles des domaines des variables sont au plus égales à d.*

1.3.2 Cohérences aux bornes

Le filtrage par cohérence d'arc n'est possible en général que lorsque les domaines des variables ne sont pas trop grands. Sinon, le fait de vérifier chaque valeur des variables peut amener à une complexité en temps et en espace trop grande (voir table 1.1). Il est possible d'utiliser un filtrage moins lourd qui va s'appliquer aux bornes des domaines des variables, c'est-à-dire, aux valeurs maximum et minimum des domaines. Nous allons donc aborder dans les deux sections suivantes deux notions de cohérence moins fortes en terme de filtrage que l'arc cohérence. La première, la D-cohérence, vérifie l'existence de supports dans les domaines des variables et uniquement pour les bornes d'un domaine. La seconde, la Z-cohérence, vérifie également l'existence de support pour les bornes d'une variable. En revanche avec ce filtrage, le support doit appartenir au réseau initial et peut très bien ne plus appartenir au sous-réseau réduit en cours de recherche.Il peut arriver sur certains problèmes que les filtrages induits par ces propriétés soient aussi efficaces que ϕ_{AC}. Le gain en temps et en espace est alors conséquent. Ces notions ont commencé à voir le jour avec [30]. Des définitions plus proches de celles communément admises sont données dans [106, 3]. Des travaux plus récents utilisent ces propriétés et étudient leur propagation [20, 3, 114].

D-cohérente aux bornes

Définition 1.20. Soit C une contrainte d'un réseau (X, C) avec $scp(C) = \{X_1, \ldots, X_n\}$. Nous dirons que C est **D-cohérente aux bornes** si

$$\forall i_0 \in [1, n], \ \forall a \in \{\min(dom(X_{i_0})), \max(dom(X_{i_0}))\},$$

$$\exists\,(x_1,\ldots,x_{i_0-1},x_{i_0+1},\ldots,x_n)\in\Pi_{i=1,i=i_0}^{n}\,dom(X_i)$$

tel que $C(x_1, \ldots, x_{i_0-1}, a, x_{i_0+1}, \ldots, x_n) = 1$.

Définition 1.21. Un réseau (X, C) est dit **D-cohérent** si toutes ses contraintes sont D-cohérentes.

Comme précédemment, cette notion permet de définir un filtrage.

Proposition 3. Soit $R = (X, C)$ et D et D^0 deux sous ensembles de D tels que $R_{|_D}$ et $R_{|_{D^0}}$ soient D-cohérents alors $R_{|_{D \cup D^0}}$ est D-cohérent.

La preuve de cette proposition est simple et permet de montrer comme pour la proposition 1 page 12 la proposition suivante :

Proposition 4. Tout réseau de contraintes R possède un sous-réseau réduit D-cohérent maximum pour l'ordre naturel d'inclusion sur D.

Remarque 1.19. Le sous-réseau réduit D-cohérent maximum vérifie

$$R_{|_{sols(R)}} \subseteq R_{|_E}.$$

Définition 1.22. Nous appellerons **filtrage par D-cohérence** et nous noterons ϕ_{DC} le filtrage qui à un réseau associe son plus grand sous-réseau réduit D-cohérent pour l'ordre naturel d'inclusion sur $\Pi_{x \in X} dom(X)$.

Remarque 1.20. Une solution donne bien un sous-réseau D-cohérent. Ce qui permet de filtrer des valeurs au cours de la recherche. Nous maintenons la D-cohérence aux bornes en supprimant les valeurs des domaines des variables qui ne permettent pas d'avoir la D-cohérence pour toutes les contraintes. L'ordre dans lequel les variables et les contraintes sont prises n'a pas d'impact sur le résultat final puisque le point fixe est $\phi_{DC}(R)$.

Exemple 1.10. Reprenons l'exemple 1.9 de la page 13. Nous avons le problème

2	7	
		1
		8

Avec les variables :

2	7	X_1
X_2	X_3	1
X_4	X_5	8

et tous les domaines égaux à {3, 4, 5, 6, 9}.

Supposons toujours qu'au cours de la recherche, la variable X_1 soit affectée à 6.

2	7	6	→ 15
X_2	X_3	1	
X_4	X_5	8	

↓

15

La contrainte C_0 correspondant au *AllDiff* est *D*-cohérente. En effet, prenons X_2 dans le scope de C_0. Les bornes de son domaines sont 3 et 9. Il existe un support pour cette contraintes pour ces deux valeurs. À la différence du filtre par arc cohérence, la valeur 6 n'est pas supprimée des domaines de X_2, X_3, X_4 et X_5.

La contrainte $X_2 + X_3 + 1 = 15$ n'est pas D-cohérente. En effet, le domaine de la variable X_2 a pour bornes 3 et 9. La valeur 3 n'a pas de support, elle est donc supprimée du domaine de X_2. la nouvelle borne inférieure du domaine de X_2 est 4 qui n'a pas non plus de support. La valeur 4 est donc supprimée du domaine de X_2. La nouvelle borne inférieure du domaine de X_2 est 5. Le domaine de X_2 est donc {5, 6, 9}. De même, $dom(X_3) = \{5, 6, 9\}$.

La contrainte $X_4 + X_5 + 8 = 15$ permet de réduire les domaines de X_4 et X_5 à {3, 4}. La contrainte $2 + X_2 + X_4 = 15$ permet d'obtenir $dom(X_2) = \{9\}$ et $dom(X_4) = \{4\}$. De même, la contrainte $7 + X_3 + X_5 = 15$ permet d'obtenir $dom(X_3) = \{5\}$ et $dom(X_5) = \{3\}$. Sur cet exemple, nous obtenons le même niveau de filtrage qu'avec le filtrage par arc cohérence mais nous voyons qu'à chaque étape nous filtrons moins de valeur.

Ainsi, le filtrage par cohérence d'arc filtre plus de valeurs mais sa complexité temporelle peut s'avérer beaucoup plus grande. Pour certaines contraintes, un filtrage par ϕ_{AC} est NP-dur ou pseudo polynomiale alors qu'un filtrage par ϕ_{DC} est polynomiale (voir [36] chapitre 34, page 933 pour les notions de complexité). Par exemple, le filtrage ϕ_{AC} de la contrainte $\sum c_i x_i = k$ est pseudo polynomial alors que ϕ_{DC} est polynomial.

Z-cohérente aux bornes

Définition 1.23. Considérons le réseau de contraintes $R = (X, C)$ et $C \in C$ et un ensemble $D = (D_X)_{X \in X} \subseteq D$. Nous dirons qu'une contrainte C avec $scp(C) = \{X_1|_{D_{X_1}}, \ldots, X_n|_{D_{X_n}}\}$ du réseau $R|_D$ est **Z-cohérente aux bornes** si

$$\forall i_0 \in [1, n], \ \forall a \in \{\min(D_{X_{i_0}}), \max(D_{X_{i_0}})\},$$

$$\exists (x_1, \ldots, x_{i_0-1}, x_{i_0+1}, \ldots, x_n) \in \prod_{i=1, i=i_0}^{n} [\min(D_{X_i}), \max(D_{X_i})]$$

$$\text{tel que } C(x_1, \ldots, x_{i_0-1}, a, x_{i_0+1}, \ldots, x_n) = 1.$$

Ainsi, par rapport à la D-cohérence, nous élargissons le filtrage en demandant l'existence d'une valeur sur un intervalle comprenant le domaine initial. Nous pourrons donc éventuellement accepter des valeurs n'appartenant peut-être pas au domaine courant des variables.

Remarque 1.21. Notons que la D-cohérence aux bornes est plus exigeante que la Z-cohérence aux bornes dans la mesure où la recherche d'un support pour la Z-cohérence se fait sur un ensemble plus grand que la même recherche pour la D-cohérence En effet, $\forall X \in X$,

$$D_X \subseteq [\min(D_X), \max(D_X)].$$

Comme précédemment, cette notion permet d'utiliser un filtrage.

Proposition 5. Soit $R = (X, C)$ et D et D^0 deux sous ensembles de D tels que $R|_D$ et $R|_{D^0}$ soient Z-cohérents alors $R|_{D \cup D^0}$ est Z-cohérent.

Comme pour l'arc cohérence et la D-cohérence, la preuve de cette proposition est évidente et permet de montrer la stabilité par réunion.

Proposition 6. Tout réseau de contraintes R possède un sous-réseau réduit Z-cohérent maximum pour l'ordre naturel d'inclusion sur D.

Remarque 1.22. Comme dans les cas d'arc cohérence et de D-cohérence, le sous-réseau réduit Z-cohérent maximum vérifie :

$$R|_{sols(R)} \subseteq R|_{E}.$$

où

$$E = \bigcup \{ D \subseteq D \mid R|_{D} \text{ est Z-cohérent} \}.$$

Définition 1.24. Nous appellerons **filtrage par Z-cohérence** et nous noterons ϕ_{ZC} le filtrage qui à un réseau associe son plus grand sous-réseau réduit Z-cohérent pour l'ordre naturel d'inclusion sur $\Pi_{x \in X} dom(X)$.

Remarque 1.23. Une solution donne bien un sous-réseau Z-cohérent. Ce qui permet de filtrer des valeurs au cours de la recherche. La Z-cohérence aux bornes est maintenue en supprimant les valeurs des domaines des variables qui ne permettent pas d'avoir la Z-cohérence pour toutes les contraintes. L'ordre dans lequel les variables et les contraintes sont prises n'a pas d'impact sur le résultat final puisque le point fixe est $\phi_{ZC}(R)$.

Exemple 1.11. Reprenons l'exemple 1.9 de la page 13 mais modifions un peu la modélisation :

X_0	7	X_1	→ 15
X_2	X_3	1	
X_4	X_5	8	
↓			
15			

$$\begin{aligned}
C_0 &= AllDiff(X_0, X_1, X_2, X_3, X_4, X_5)\\
C_1 &: X_0 + 7 + X_1 = 15\\
C_2 &: X_2 + X_3 + 1 = 15\\
C_3 &: X_4 + X_5 + 8 = 15\\
C_4 &: X_0 + X_2 + X_4 = 15\\
C_5 &: 7 + X_3 + X_5 = 15\\
C_6 &: X_1 + 1 + 8 = 15\\
C_7 &: X_0 + X_3 + 8 = 15\\
C_8 &: X_1 + X_4 + X_5 = 15.
\end{aligned}$$

$dom(X_0) = \{2\}$ et

$$\forall i \in [1,5],\ dom(X_i) = \{2, 3, 4, 5, 9\}.$$

Affectons X_1 à 6. Appliquons le filtre ϕ_{ZC} à la contrainte C_0. Aucune valeur n'est filtrée bien sûr sur le domaine de X_0. En revanche 2 est la borne inférieure des domaines des variables X_2, X_3, X_4 et X_5. N'ayant pas de support pour cette contrainte, la valeur 2 est supprimée du domaine de X_2, X_3, X_4 et X_5. Comme dans le cas du filtrage ϕ_{DC} l'affectation $X_1 = 6$ n'a pas d'incidence directe sur les autres contraintes. Nous avons alors :

$$\begin{aligned}
dom(X_0) &= \{2\}\\
dom(X_1) &= \{6\}\\
\forall i \in [2,5],\ dom(X_i) &= \{3, 4, 5, 9\}.
\end{aligned}$$

Considérons le filtrage sur la contrainte $X_4 + X_5 + 8 = 15$. Les bornes du domaine de X_4 sont 3 et 9. La valeur 3 n'est pas filtrée puisque $3 + 4 + 8 = 15$. En revanche, la valeur 9 est supprimé du domaine de X_4 car il n'existe pas de support pour cette valeur. Ainsi $dom(X_4) = \{3, 4, 5\}$. La valeur 5 n'est pas supprimée car $5 + 2 + 8 = 15$. En effet, la valeur 2 était présente dans tous les domaines à l'initialisation du problème. Le filtrage ϕ_{ZC} sur cette contrainte donne $dom(X_4) = dom(X_5) = \{3, 4, 5\}$.

Le filtrage sur la contrainte C_2 donne $dom(X_2) = dom(X_3) = \{5, 9\}$. Le filtrage sur la contrainte C_4 donne $dom(X_2) = \{9\}$ et $dom(X_4) = \{4\}$. Le filtrage sur la contrainte C_5 donne $dom(X_3) = \{5\}$ et $dom(X_5) = \{3, 4, 5\}$. Le réseau est Z-cohérent et il faut procéder à une nouvelle affectation. Le filtrage est donc beaucoup plus rapide à obtenir qu'avec ϕ_{AC} ou ϕ_{DC} mais le nombre de valeurs filtrées est nettement réduit.

Nous pouvons nous interroger sur l'utilité de ces contraintes qui filtrent dans certaines situations beaucoup moins que le filtrage par cohérence d'arc. En fait, comme cela a déjà été évoqué, le filtrage par cohérence d'arc peut avoir des complexités spatiale et temporelle élevées. De plus, pour certains types de contraintes les cohérences d'arc et aux bornes peuvent être équivalentes.

Intéressons-nous à un mécanisme de filtrage générique qui consiste à chercher un support pour chaque valeur en parcourant le domaine de la seconde variable. Considérons, par exemple, le problème suivant : $X < Y$ $dom(X) = dom(Y) = [0, 10]$.

Par souci de simplicité, nous considèrerons ici une heuristique de choix de variables donnée par $X \prec Y$ et une heuristique de choix de valeurs donnée par l'ordre naturel sur $\mathbb{N}$.

Appliquons le filtrage ϕ_{AC}. L'algorithme de filtrage va considérer les différentes valeurs du domaines de X puis de Y est va chercher un support. Ainsi

- Pour $X = 0$, nous testons $Y = 0$ qui n'est pas un support puis $Y = 1$ convient.
- Pour $X = 1$, les valeurs 0 et 1 sont testées avant de trouver $Y = 2$
- Pour $i \in [0, 9]$, $X = i$ va demander $i + 2$ tests avant de trouver un support à $(Y, i + 1)$.
- Pour $X = 10$, nous avons 11 tests à effectuer avant de supprimer 10 du domaine de X.

En tout, le nombre d'opérations est $2 + 3 + \cdots + 11 + 11 = \frac{12 \times 13}{2} - 1 = 77$. Il reste ensuite à tester toutes les valeurs du domaine de Y.

- $Y = 0$, demande 10 tests pour constater qu'il n'existe pas de support pour cette instanciation et 0 est supprimée du domaine de Y.
- $Y = 1$, demande 1 tests pour trouver un support pour $(X, 0)$.
- Pour $i \in [1, 10]$, $(Y = i, X = 0)$ convient et ne demande qu'un test.

En tout, le passage par la variable Y demande $10 + 10 = 20$ opérations. Au total, cette première passe a couté 97 opérations. Il faut en faire une seconde qui permettra de vérifier que nous avons bien un point fixe. La deuxième passe coûte 65 opérations, soit un coût total pour ϕ_{AC} de 162 opérations.

Appliquons maintenant la filtrage ϕ_{DC} à ce même réseau.

- Initialement, nous avons $\min(dom(X)) = 0$ et $\max(dom(X)) = 10$. La recherche d'un support pour $(X, 0)$ se fait en deux opérations. La détection de l'absence de support de $(X, 10)$ coûte 11 opérations. La valeur 10 est supprimée du domaine de X.
- Nous avons, $\min(dom(Y)) = 0$ et $\max(dom(Y)) = 10$. La recherche du support de $(Y, 10)$ demande 1 opération et la détection de l'absence de support de $(Y, 0)$ demande 10 opérations. La valeur 0 est supprimée du domaine de Y.

Cette première passe demande $11 + 2 + 1 + 10 = 24$ opérations. La deuxième passe ne filtre aucune valeur et permet de savoir qu'un point fixe est atteint. Cette deuxième passe demande $1 + 10 + 1 + 1 = 13$ opérations. Au total, le filtrage ϕ_{DC} a demandé 37 opérations.

Sur cet exemple, la complexité en nombre d'opérations est beaucoup plus faible avec le filtrage ϕ_{DC} pour un résultat identique. Avec un mécanisme de filtrage dédié à la sémantique de la contrainte $X < Y$, le nombre d'opérations serait encore plus limité. Ici, le filtrage ϕ_{ZC} donnerait le même résultat que ϕ_{DC} puisqu'il n'y a pas de trous dans les domaines. Le filtrage ne portant que sur les bornes des domaines, ce résultat est peu étonnant. En revanche, il semble surprenant de voir à quel point le filtrage par arc cohérence est peu efficace étant donné le nombre de couples à supprimer réellement. Cela tient au fait qu'en programmation par contraintes nous travaillons par nature sur des pro-jections. En effet, comme nous l'avons vu page 6, l'ensemble des solutions est donné par une intersection de sous-ensembles du produit cartésien des domaines des variables.

Bilan des filtrages par cohérence

Des trois parties précédentes, nous pouvons conclure que pour tout réseau de contrainte s R, nous avons :

$$R_{|_{sols(R)}} \subseteq \phi_{AC}(R) \subseteq \phi_{DC}(R) \subseteq \phi_{ZC}(R).$$

Plus un filtrage se rapproche de la solution, plus le filtrage est efficace en terme de nombre de valeurs éliminées.

Notation. Soit ϕ et ψ deux filtrages, nous noterons $\phi \leq \psi$ si pour tout réseau de contraintes R, nous avons $\phi(R) \subseteq \psi(R)$.

Nous avons alors :

$$\phi_{AC} \leq \phi_{DC} \leq \phi_{ZC}.$$

1.3.3 Forward checking

L'algorithme qui va être présenté dans cette partie est un filtrage partiel d'arc cohérence . Il a longtemps été utilisé pour des contraintes binaires. Les réseaux non-binaires étaient transformés en réseaux binaires pour pouvoir être utilisés dans un contexte plus large. Le coût en temps et en espace de cette transformation a poussé la communauté à réfléchir à l'extension de cette cohérence pour des réseaux non-binaires. De nombreuses possibilités d'extension ont alors émergé [17].

L'algorithme MAC applique un filtrage de cohérence d'arc en visitant toutes les contraintes. Si un domaine d'une des variables est modifié, alors les contraintes sont visitées à nouveau. Ceci jusqu'à obtenir un point fixe, c'est-à-dire que plus aucune valeur ne soit supprimée d'un domaine par de mécanisme. Une passe sur les contraintes est un passage en revue des contraintes. Ce processus peut demander beaucoup de passes et différentes stratégies ont été élaborées afin de réduire ce calcul quitte à réduire le filtrage [69]. Une de ces stratégies est mise en place dans le chapitre 4. Nous y verrons notamment que l'ordre dans lequel les contraintes sont visitées a une importance dans certains filtrages. L'algorithme MAC cherche un point fixe et le résultat ne dépend ni de l'ordre des variables ni de l'ordre des contraintes. En revanche, le nombre de passes pour y arriver en dépend fortement.

Définition 1.25. Soit $R = (X, C)$ un réseau binaire tel que $R = \perp$. Notons X_p l'ensemble des variables assignées (dont le domaine est réduit à un singleton) et X_f l'ensemble des variables de X non présentes dans X_p. Nous appellerons, filtrage **Forward Checking pour les réseaux binaires**, noté ϕ_{bFC}, le filtrage qui à tout réseau R associe le plus grand sous-réseau $R_{|_D}$ de R tel que $\forall C \in C$ avec $scp(C) = \{X, Y\}$

$$(X, Y) \in (X_p \times X_f) \cup (X_f \times X_p) \Rightarrow C \text{ est arc cohérente.}$$

Comme précédemment cette définition est stable par réunion et préserve les solutions. Il s'agit d'un filtrage prudent.

Comme cela est décrit dans l'article [17], cette définition peut être étendue de diverses manières aux réseaux de contraintes non binaires. Deux solutions semblent les plus naturelles. La première, notée **FC2** dans [17], modifie la propriété en assujettissant celle-ci avec l'existence d'une variable dans chaque ensemble. A tout réseau R le filtrage associe le plus grand sous-réseau $R_{|_D}$ de R tels que $\forall C \in C,\ \forall X, Y \in scp(C),\ X = Y$,

$$(X, Y) \in (X_p \times X_f) \cup (X_p \times X_f) \Rightarrow C \text{ est arc cohérente.}$$

Pour la seconde, notée **FC0** dans [17], après l'instanciation d'une variable X et pour toute contrainte n'ayant plus qu'une variable Y non affectée, cette propriété est appliquée entre X et Y.

Exemple 1.12. Reprenons l'exemple 1.1.1 de la page 2. Nous allons utiliser le filtre **FC0**. Affectons la variable X_1 à 6. Ceci n'a aucune incidence en terme de filtrage car il n'y a pas de contrainte n'ayant plus q'une seule variable non affectée. Affectons alors X_2 à 3. Les contraintes $X_2 + X_3 + 1 = 15$ et $2 + X_2 + X_4 = 15$ sont concernées par le filtrage puisque X_2 est affectée et X_3 et X_4 ne le sont pas. Il n'y a pas de support pour $X_2 = 3$ pour la contrainte $X_2 + X_3 + 1 = 15$. Donc 3 est supprimé du domaine de X_2. Après filtrage de la contrainte $X_2 + X_3 + 1 = 15$, nous avons $dom(X_2) = dom(X_3) = \{5, 9\}$. Après filtrage pour la contrainte $2 + X_2 + X_4 = 15$, nous avons $dom(X_2) = \{9\}$ et $dom(X_4) = \{4\}$. Ensuite X_2 est affectée à 9, le filtrage donne alors $dom(X_3 = 5$. La variable X_3 est alors affectée à $\{5\}$, ce qui va impliquer que $dom(X_5 = 3$. Le problème est alors résolu. Sur cet exemple, le filtrage par forward checking est efficace mais sur problème plus gros, peu de contraintes sont étudiées et le filtrage est moins intéressant.

1.3.4 Singleton arc cohérence

Nous allons étudier ici un filtrage introduit par [42] et basé une fois encore sur la cohérence d'arc. Nous verrons que ce filtrage peut s'avérer très efficace en terme de valeurs supprimées mais extrêmement coûteux en espace et en temps de calcul. Cette méthode de filtrage est très utilisée dans les méthodes d'apprentissage [20].

Définition 1.26. Nous dirons qu'un réseau de contraintes (C, X) est **singleton arc cohérent (SAC)** si $\forall X \in X,\ \forall a \in dom(X),\ \phi_{AC}(R_{|_{X=a}}) = \perp$.

Remarque 1.24. Une solution représente un sous-réseau singleton arc cohérent. En l'absence de traduction satisfaisante, nous laisserons, ici, l'anglicisme du nom de cette propriété.

Clairement, cette propriété va être stable pour la réunion.

Proposition 7. Soit R = (X, C) et D et D^0 deux sous ensembles de D tels que $R_{|_D}$ et $R_{|_{D^0}}$ soient singleton arc cohérents alors $R_{|_{D \cup D^0}}$ est singleton arc cohérent.

Proposition 8. Soit R = (X, C) un réseau de contrainte consistant alors R possède un sous-réseau réduit singleton arc cohérent maximum pour l'ordre naturel d'inclusion sur D

Définition 1.27. Nous appellerons **filtrage par singleton arc cohérence** et nous noterons ϕ_{SAC} le filtrage qui à un réseau de contraintes R associe le plus grand sous-réseau réduit singleton arc cohérent de R pour l'ordre naturel d'inclusion sur D.

Comme pour les autres filtrages, les valeurs sont supprimées des domaines jusqu'à l'obtention d'un réseau SAC.

Ce filtrage vérifie que le réseau est arc cohérent pour toutes les valeurs du domaine de chaque variable. Ainsi, le filtrage SAC est très lourd en terme de calcul mais peut parfois supprimer beaucoup de valeurs des domaines. Il filtre plus de valeurs des domaines

Algorithme	Complexité en temps	Complexité en espace
SAC1 [42]	$O(en^2d^4)$	$O(ed)$
SAC2 [6]	$O(en^2d^4)$	$O(n^2d^2)$
SAC-Opt [13]	$O(end^3)$	$O(end^2)$
SAC-SDS [14]	$O(end^4)$	$O(n^2d^2)$
SAC3 [71]	$O(en^2d^4)$	$O(ed)$

Table 1.2: *Complexité de différents algorithmes établissant la propriété SAC pour des réseaux binaires comportant n variables, e contraintes et dont les tailles des domaines des variables sont au plus égales à d.*

que GAC. De nombreux algorithme permettent d'établir la propriété SAC. Nous pouvons citer par exemple : SAC1 [42], SAC2 [6], SAC-Opt [13], SAC-SDS [14] et SAC3 [71]. La complexité de ces algorithmes en temps et en espace est donné dans le tableau 1.2.

Des méthodes probabilistes ont été étudiées afin de réduire le coût de SAC en gardant un filtrage acceptable [82].

Il existe donc différentes techniques de filtrage représentant souvent un ratio entre filtrage et vitesse. La méthode filtrant le plus n'est pas nécessairement la plus rapide dans le contexte d'une recherche arborescente. Suivant le problème proposé, certains filtrages seront adaptés ou non. Il semble inévitable de proposer des méthodes s'adaptant aux problèmes comme c'est le cas dans le chapitre 5. Il faut également éviter de biaiser les filtrages en les adaptant aux problèmes classiques utilisés dans la communauté des réseaux de contraintes.

Chapitre 2

Squelette d'un mini-Solver

Dans ce chapitre, nous allons nous intéresser à la structure d'un solver de contraintes. Nous utiliserons notamment une structure de données appelée **sparse set** (en anglais) qui sera étudiée dans la première partie de ce chapitre. Nous pourrions traduire sparse set par ensembles épars mais, comme c'est l'usage, nous garderons l'anglicisme sparse set. Les éléments clés d'un solver seront donnés dans la deuxième partie. Nous décrirons un diagramme générique de solver et le code Python des classes essentielles à son fonctionnement. Enfin, nous finirons à titre d'exemple, par un algorithme de filtrage dédié à la contrainte $X = Y + Z$.

2.1 Sparse set

2.1.1 Définition

La notion de sparse set date de la fin du xx^{e} siècle, elle a été introduite par Briggs and Torczon [24]. Elle est utile lorsque les valeurs d'un ensemble sont connues et bornées. L'utilisation des indices des valeurs permet de réduire la complexité de la plupart des opérations de base sur un tel ensemble. Le tableau suivant compare la complexité en nombre d'opérations du sparse set avec différentes structures classiques (voir [67]). Nous considèrerons ici des domaines égaux à un sous-ensemble D de $[\![1, N]\!]$, pour un entier naturel N.

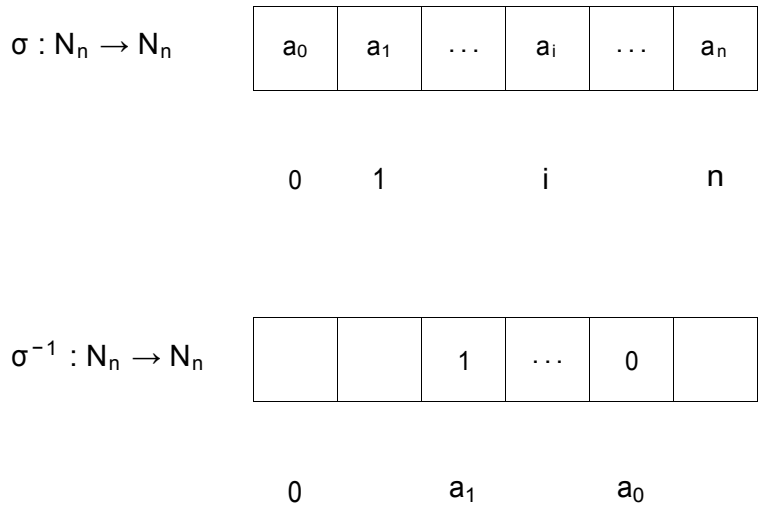
	Vecteur de Bit	Liste chaînée	Sparse Set
Vérifier l'appartenance d'une valeur dans D	$O(1)$	$O(1)$	$O(1)$
Supprimer valeur	$O(1)$	$O(1)$	$O(1)$
Restorer n valeurs	$O(n)$	$\Theta(n)$	$O(1)$
Itérer	$O(N)$	$O(\|D\|)$	$O(\|D\|)$
Trouver min/max	$O(N)$	$O(1)$	$\Theta(\|D\|)$

Pour comprendre comment fonctionne les sparse set, nous allons considérer que les tableaux de taille $n + 1$ contenant les $n + 1$ premiers éléments sont des bijections de N_n dans N_n.

$\sigma : \mathrm{N}_n \to \mathrm{N}_n$

a_0	a_1	a_2	a_3	...	0	...	a_n
0	1	2	3		i_0		n

Cette bijection associe à un indice du tableau, l'élément de ce tableau qui a cet indice. Ainsi trouver l'indice d'un élément revient à déterminer son antécédent par cette bijection. Nous pouvons donc utiliser la bijection inverse sous forme d'un tableau.



Nous garderons une fois encore les termes en anglais pour les définitions qui suivent. Le premier tableau contenant les éléments entre est appelé **tableau dense** et le second **tableau sparse**.

De plus, un élément de $N_n \cup \{-1\}$, appelé **limite** , permet de déterminer si un élément du premier tableau est considéré comme présent . Les éléments dont l'indice est plus petit ou égal à la limite sont présents comme nous pouvons le voir sur la figure 2.1.

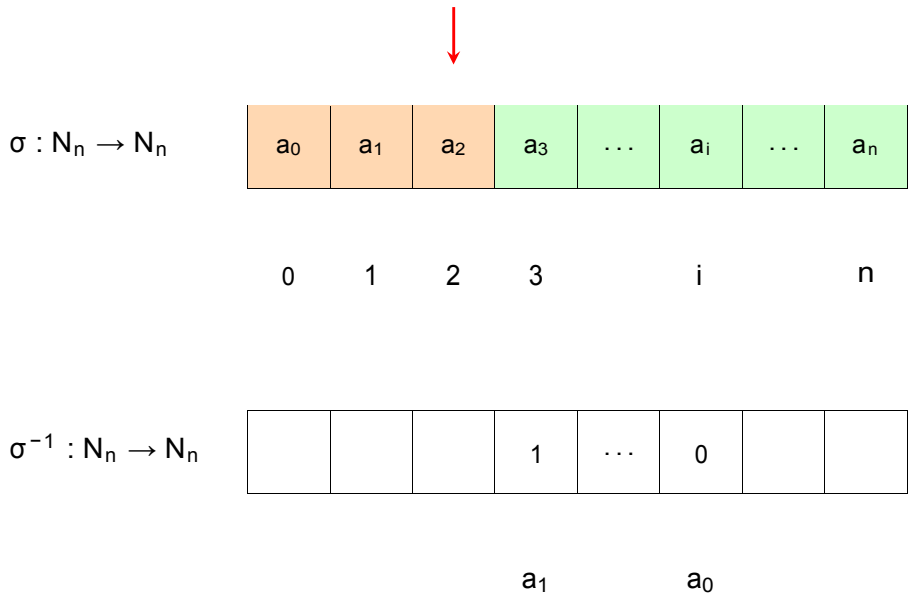


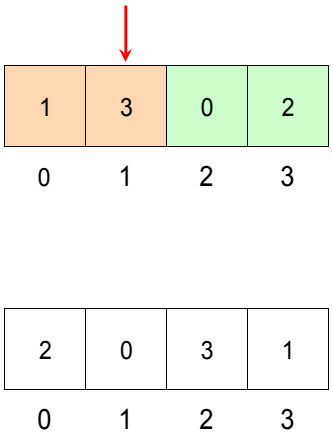
Figure 2.1 : *La limite du sparseSet est* 2. *Les éléments,* a_0, a_1 *et* a_2 *sont présents.*

Ainsi un sparse set de taille $n+1$ est la donnée de deux tableaux de taille $n+1$ et d'un entier de $N_n \cup \{-1\}$. Formellement, un sparse set est représenté par une bijection sur N_n et un entier $N_n \cup \{-1\}$. Les éléments présents sont dans le tableau dense, leur position est donnée par le tableau sparse.

Le tableau dense contient les éléments présents (et absents). Le tableau sparse donne la position

Remarque 2.1. Si la limite d'un sparse set est égale à -1 alors cela signifie que l'ensemble est vide.

Exemple 2.1. La figure suivante représente un sparse set de taille 4 et de limite 1 pour lequel les éléments 1 et 3 sont présents et les éléments 0 et 2 ne le sont pas.



2.1.2 Opérations sur un sparse set

Dans cette section, nous allons décrire les opérations de suppression et d'ajout sur les spare set. Ces opérations sont simples à effectuer et nous verrons plus loin qu'elles seront réalisées de très nombreuses fois au sein du schéma que nous présentons, ce qui justifie l'usage de cette structure.

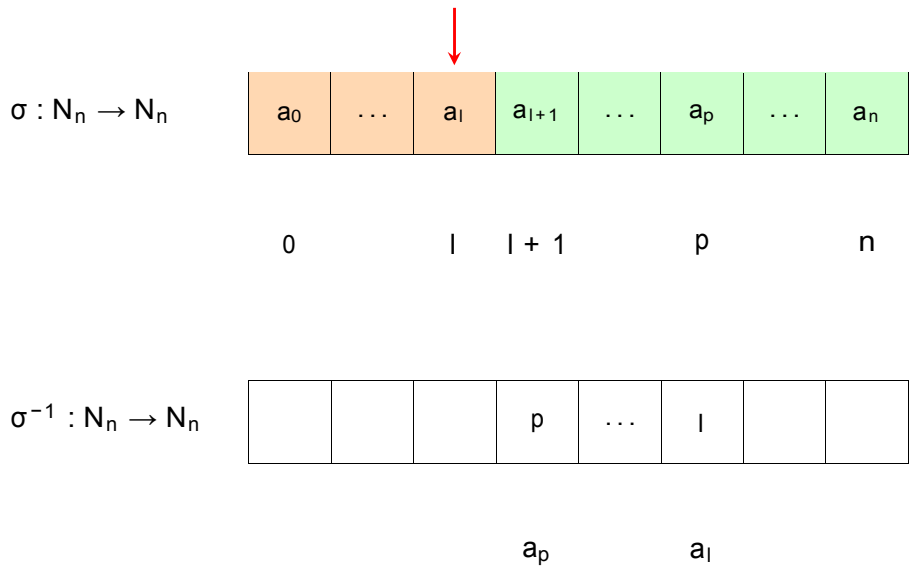
Tester si un élément est présent

Un entier $a \in N_n$ est présent dans un sparse set représenté par une bijection σ et une limite l si $\sigma^{-1}(a) \leq l$.

Ajouter un élément

Pour ajouter un élément absent, il suffit d'incrémenter la limite l de 1, de permuter cet élément avec l'élément d'indice $l + 1$ dans le tableau dense et de permuter les éléments d'indices correspondants dans le tableau sparse.

Considérons un sparse set donné par la figure suivante :



Pour ajouter l'élément a_p, il faut incrémenter la limite l de 1, permuter a_{l+1} et a_p dans le tableau dense et l et p dans le tableau sparse comme dans la figure 2.2.

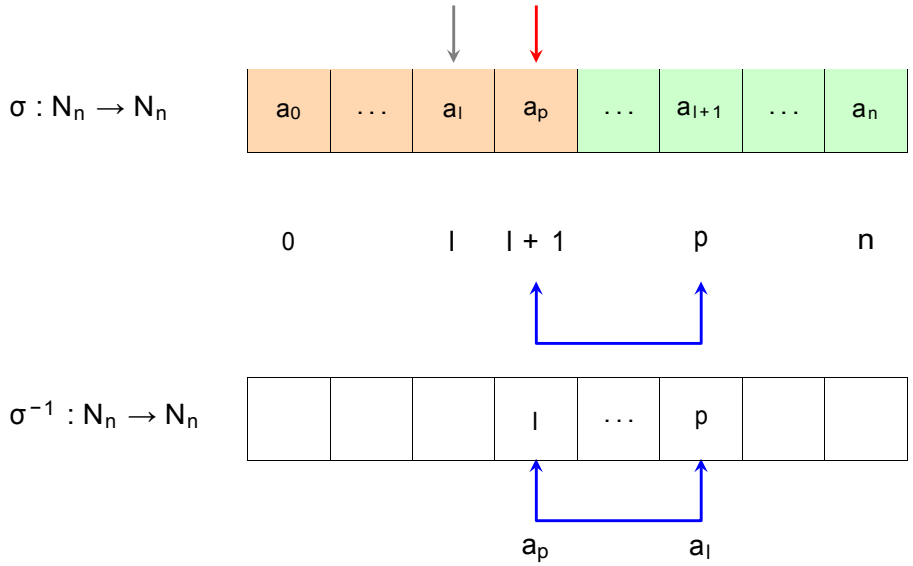
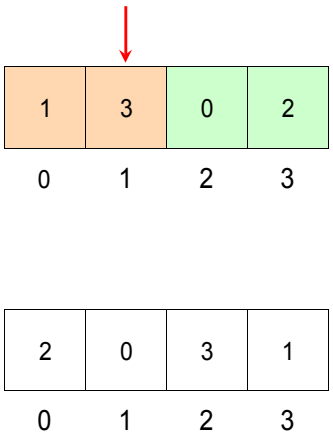
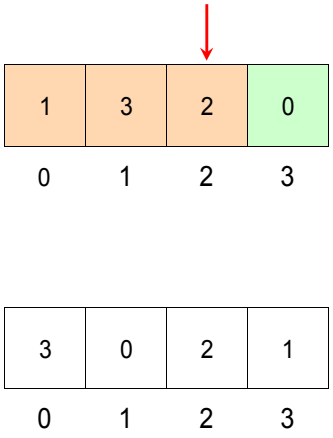


Figure 2.2 : *Ajout de l'élément a_p.*

Exemple 2.2. Reprenons l'exemple précédent :



Pour ajouter l'élément 2. Il faut permuter 0 et 2 et incrémenter la limite, nous obtenons alors l'état suivant :



Remarque 2.2. Considérons un sparse set représenté par une permutation σ et une limite l, alors ajouter un élément a d'indice j donne le sparse set représenté par la limite $l + 1$ et la permutation

$$\sigma \times (\sigma(l + 1), a).$$

De plus,

$$\begin{aligned} \sigma \times (\sigma(l+1), a) &= \sigma \times (\sigma(l+1), \sigma(j)) \\ &= (l+1, j) \times \sigma. \end{aligned}$$

Ainsi, le nouveau tableau sparse s'obtient en permutant $l+1$ et j car

$$((l+1, j) \times \sigma)^{-1} = \sigma^{-1} \times (l+1, j).$$

Supprimer un élément

Supprimer un élément d'un sparse set revient à inverser l'opération précédente. Pour supprimer un élément a à un sparse set de limite l, il suffit de permuter l'élément d'indice l avec a et de décrémenter la limite. Considérons le sparse set de la figure 2.3.

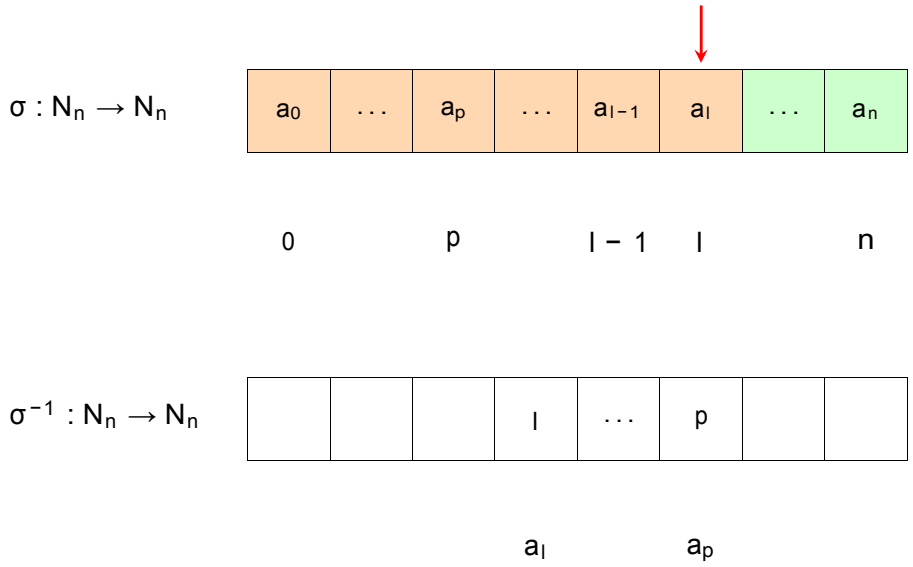


Figure 2.3 : *État avant suppression de l'élément a_p.*

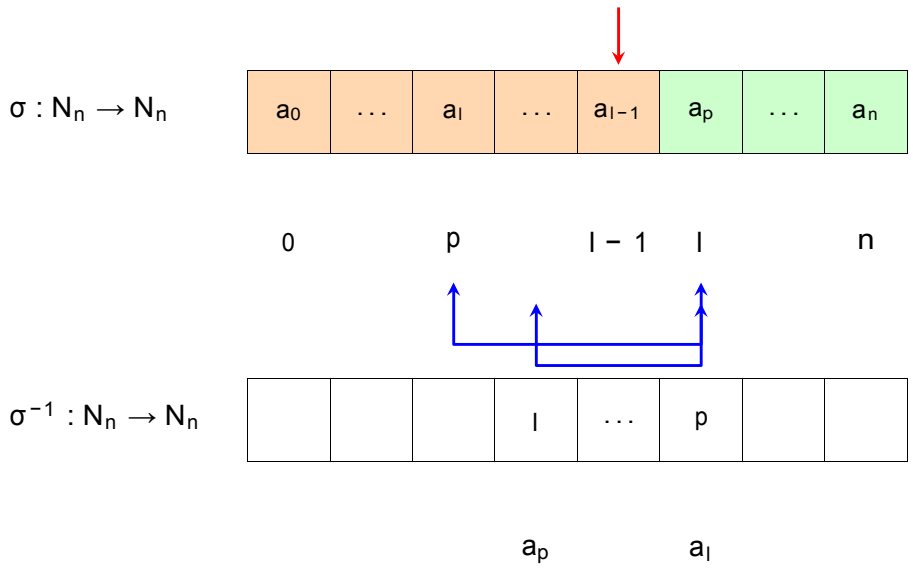


Figure 2.4 : *État après suppression de l'élément* a_p.

Supprimer tous les éléments

Pour supprimer tous les éléments d'un sparse set, il suffit d'affecter la limite à -1.

Ajouter tous les éléments

Pour ajouter tous les éléments à un sparse set, il suffit d'affecter à la limite la taille du tableau des valeurs.

Les sparse set sont donc bien adaptés aux structures avec retour arrière. En particulier pour les domaines des variables dans les réseaux de contraintes. Nous allons maintenant décrire pas à pas la structure d'un solveur avec cette structure.

2.1.3 Programmation d'un sparse set

Pour simplifier la syntaxe, les programmes seront donnés en Python.

```
class SparseSet():

    def __init__(self,size,full=False):
        self.limit = size-1 if full else -1
        self.dense = range(size)
        self.sparse = range(size)
```

Comme nous pouvions nous y attendre, la classe SparseSet a trois attributs : la limite et les deux tableaux, dense et sparse. Un paramètre `full` est ajouté pour pouvoir rendre présents tous les éléments du tableau dense dès l'instanciation.

La méthode la plus utilisée est la méthode swap qui permute deux éléments.

```
    def swap(self,elt):
        elt2 = self.dense[self.limit]
        i = self.sparse[elt]
        self.dense[self.limit],self.dense[i] = self.dense[i],self.dense[
    self.limit]
        self.sparse[elt],self.sparse[elt2] = self.sparse[elt2],self.sparse[
    elt]
```

Savoir si un élément est présent :

```
    def isPresent(self,elt):
        return self.sparse[elt]<=self.limit
```

Il est possible alors de définir les méthodes permettant d'ajouter et de supprimer un élément.

La méthode ajouter :

```
    def add(self,elt):
        if not self.isPresent(elt):
            self.limit +=1
            self.swap(elt)
            return True
        else:
            return False
```

La méthode supprimer :

```
    def delete(self,elt):
        if self.isPresent(elt):
            self.swap(elt)
            self.limit -=1
            return True
        return False
```

Chaque méthode retourne un booléen de manière à indiquer si l'opération a été effectuée ou non.

Quelques méthodes seront très utiles dans le solveur.

L'ensemble des éléments présents :

```
    def presents(self):
        return self.dense[:self.limit+1]
```

Ce qu'on peut optimiser avec un générateur [1] :

```
    def iteratorPresents(self):
        i=0
        while i<=self.limit:
            yield self.dense[i]
            i+=1
```

Le nombre d'éléments présents dans le sparse set est obtenu comme suit :

```
    def size(self):
        return self.limit+1
```

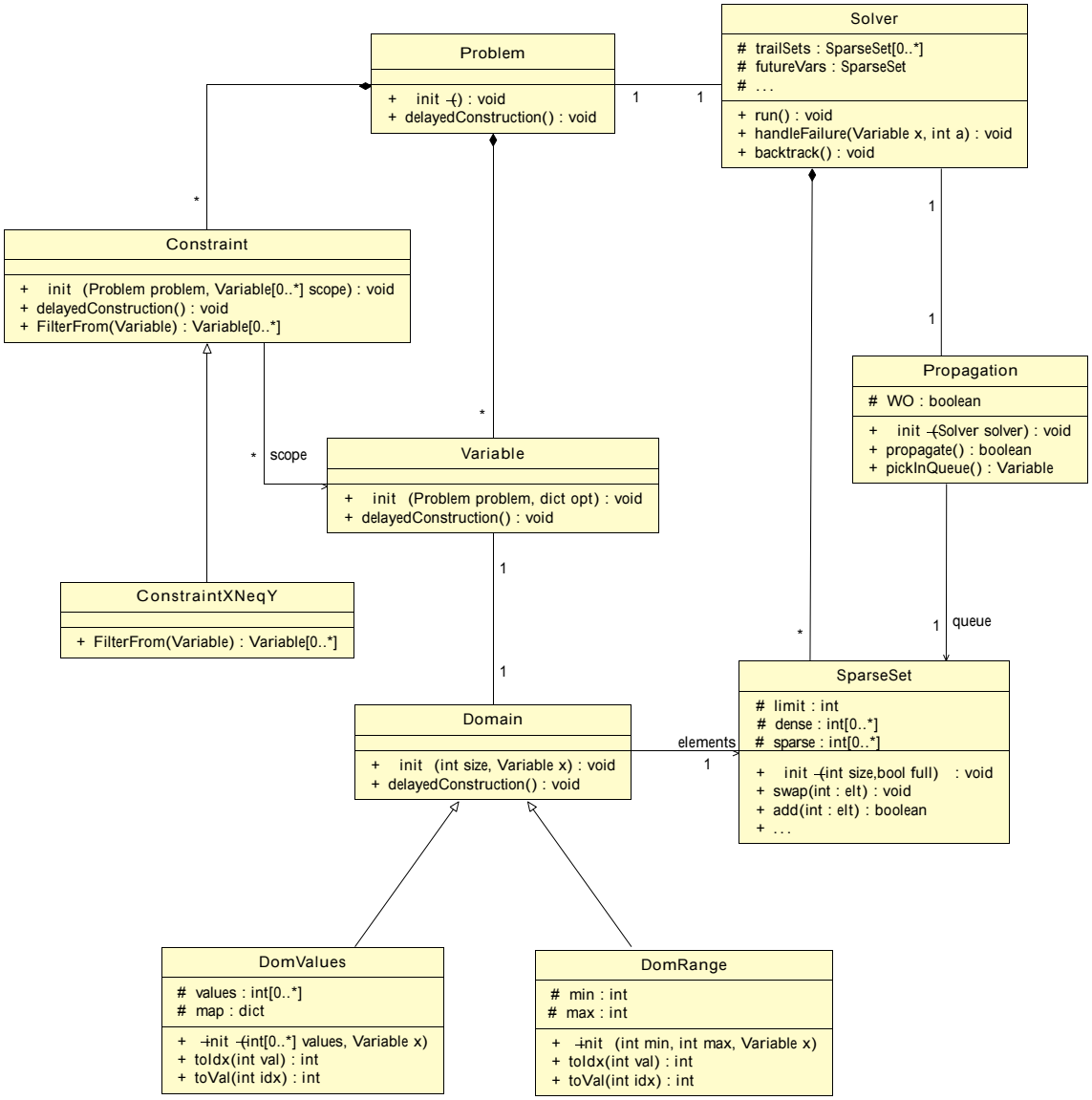
2.1.4 Diagramme de classe du solveur

Le diagramme de classes qui suit permettra de mieux se repérer dans toutes les classes à venir.

Il apparaît que la classe SparseSet est à la base du solveur ainsi construit. En effet, cette classe est utilisée dans les classes Domain, Propagation et Solver.

Ensuite, la classe Variable est créée à partir de la classe Domain et est associée à la classe Problem. Enfin, la classe Problem est construite à partir de la classe Variable et doit connaître son solveur.

1. Un générateur est un objet Python itérable utilisant un chargement à la demande. L'intérêt principal est une économie de la mémoire. Il permet également d'éviter des calculs inutiles.



2.2 Codage Python des domaines des variables

La structure de sparse set permet de bien représenter d'un point de vue informatique le domaine des variables dans la mesure où les éléments d'un domaine sont connus initialement. De plus, au cours de la recherche, les valeurs des domaines seront supprimées (lors de la descente) ou ajoutées (lors du retour arrière) à de multiples reprises. Le domaine d'une variable est défini par un ensemble de valeurs.Notons qu'à l'initialisation d'un problème, toutes les valeurs des domaines doivent être considérées comme présentes dans les sparse set correspondants. Nous allons donc utiliser le paramètre `full` des sparse set (lors de l'initialisation).

```
class Domain():

    def __init__(self,size,x):
        self.variable = x
        self.elements = SparseSet(size,True)
```

Les domaines des variables sont des ensembles généraux dont on ne sait à priori rien. La modélisation précédente impose un ensemble de valeurs égale aux n premiers entiers. En

informatique, ce sera souvent le cas et il n'y a pas de perte de généralité dans le mesure où il est toujours possible d'indicer chaque élément d'un domaine. Afin de concrétiser

cela, il suffit de définir une classe qui hérite de la classe Domain. Pour gérer des valeurs quelconques, nous pouvons utiliser la classe suivante :

```
class DomainValues(Domain):

    def __init__(self,values,x):
        super(DomainValues,self).__init__(len(values),x)
        self.values = values
        self.map = {}
        for i,v in enumerate(values):
            self.map[v]=i

    def toIdx(self,val):
        return self._map[val]

    def toVal(self,idx):
        return self._values[idx]
```

Très souvent les valeurs entières des domaines sont des intervalles entiers dont la borne de gauche n'est pas forcément 0. Dans ce cas, il peut être utile d'utiliser une autre classe :

```
class DomainRange(Domain):

    def __init__(self,min,max,x):
        super(DomainRange,self).__init__(max-min+1,x)
        self.min = min
        self.max = max

    def toIdx(self,val):
        return val-self.min if val>=self.min and val<=self.max else -1

    def toVal(self,idv):
        return self.min+idv
```

2.3 Codage Python des variables

Une variable va appartenir à un problème. Elle aura un indice pour ce problème et éventuellement un nom et un domaine associé sous une forme choisie. Bien sûr, certains attributs devront être changés au moment de l'instanciation du problème. Par soucis de concision, le constructeur de notre classe prendra en paramètres un dictionnaire. C'est à dire que les arguments seront données sous la forme d'un tableau associatif.

```
class Variable():

    def __init__(self,problem,**kwargs):
        self.problem = problem
        if "domainRange" in kwargs.keys():
            self.dom = DomainRange(kwargs["domainRange"][0],kwargs["
    domainRange"][1],self)
        elif "domainValues" in kwargs.keys():
            self.dom = DomainValues(kwargs["domainValues"],self)
        else:
            exit("Problem domain declaration")
```

Nous voyons ici que la variable est mise en attente d'une action de la classe Problem.

2.4 Codage Python des contraintes

Comme les variables, la construction des contraintes sera réalisée par la classe Problem. Une contrainte doit connaître le problème auquel elle appartient, avoir un indice dans ce

problème, avoir éventuellement un nom et le scope sera donné sous forme d'une liste (donc un ordre sera fixé).

```
class Constraint():

    def __init__(self,problem,scope):
        self.problem = problem
        self.scope = scope # ordre dans le scope !
```

Sous cette forme, il est difficile de comprendre comment nous allons exprimer des contraintes du type $X < Y$. En fait, la classe Contrainte va posséder une méthode FilterFrom que nous détaillerons plus loin et qui permettra d'exprimer la manière de filtrer avec cette contrainte durant la recherche. Plus précisément, pour créer une contrainte, il faudra créer une classe qui hérite de la classe Constraint et redéfinir la méthode FilterFrom.

2.5 Codage Python du problème

Le problème devra connaître son solver et mettre en place les variables et les contraintes. C'est le rôle de la méthode delayedConstruction.

```
class Problem(object):

    def __init__(self):
        self.solver = Solver()
        self.delayedConstruction()
```

La méthode delayedConstruction de la classe Problem peut donner :

```
    def delayedConstruction(self):
        for i,x in enumerate(self.variables):
            x.delayedConstruction(i)
        for i,c in enumerate(self.constraints):
            c.delayedConstruction(i)
```

Un indice i sera affecté à chaque variable et à chaque contrainte. Il faut donc définir les méthodes delayedConstruction pour les classes Variables et Constraint.

Ainsi, pour la classe Constraint la méthode delayedConstruction est :

```
    def delayedConstruction(self,id):
        self.id = id
```

Cette méthode est assez simple et tout le travail sera réalisé par la méthode delayedConstruction de la classe Variable :

```
1   def delayedConstruction(self,id):
2       self.id = id
3       self.constraints = []
4       for c in self.problem.constraints:
5           if self in c.scope:
6               self.constraints.append(c)
7       self.dom.delayedConstruction(len(self.problem.variables))
```

— Ligne 2, un indice est donné à la variable pour le problème.
— Ligne 3, un nouvelle attribut constraints est ajouté à la classe Variable. Il s'agit d'une liste contenant les contraintes dont le scope contient la variable.
— Ligne 7, nous allons commencer à gérer le niveau des variables. Ceci dans un objectif de retour arrière.

Le processus de retour arrière est un élément important du solver. Détaillons la méthode delayedConstruction de la classe Domain. Ajoutons deux méthodes à cette classe :

```
1     def delayedConstruction(self,nbLevels):
2         self.elements.delayedConstruction(nbLevels)
```

Cette méthode fait appel à une méthode du sparse set.

```
1     def delayedConstruction(self,nbLevels):
2         self.limits = [constantes.UNDEF]*(nbLevels+1)
```

L'attribut `limits` est une liste dont la taille est nbLevels+1. Dans le cas de l'appel par le constructeur de la classe Problem, il s'agit d'un tableau dont la taille est la nombre de variables plus un. **Ce tableau représente l'application qui à un niveau associe la limite du sparse set à ce niveau**. Ainsi, il est possible de garder en mémoire la limite du sparse set aux niveaux inférieurs, ce qui permettra de réaliser des retours arrières, par exemple des valeurs des domaines des variables. La structure de spare set permet donc de restorer un domaine en temps constant. La constante UNDEF doit être une valeur négatives strictement inférieure à −1. C'est une valeur particulière qui représente une absence de modification depuis le début de la recherche. **Cette méthode est essentielle**.

Supposons que le tableau d'un domaine ressemble au tableau suivant :

limite du Sparse set du domaine	6	4	4	2	···	UNDEF
niveau	0	1	2	3		30

Nous pouvons en conclure

- qu'au niveau 0 (aucune variable instanciée), la limite du sparse set du domaine était 6,
- qu'au niveau 1 (une variable instanciée) la limite du sparse set du domaine était 4,
- qu'au niveau 2 (deux variables instanciées) la limite du sparse set du domaine était 4,
- qu'au niveau 3 (trois variables instanciées) la limite du sparse set du domaine était 2,
- etc.
- que jamais toutes les variables n'ont été instanciées (niveau 30).

Nous avons vu que pour supprimer une valeur d'un sparse set, il suffisait d'effectuer une permutation dans les valeurs présentes et de décrémenter la limite. Ainsi, pour retrouver un domaine dans l'état qui était le sien à un niveau donné, il suffit remettre la limite de ce niveau. Seul l'ordre dans le sparse set peut avoir changé.

Dans notre exemple, si nous voulons retrouver le domaine dans l'état du niveau 2, il faut changer la limite correspondante en 4. Sachant, bien sûr que 4 est la dernière limite au niveau 2.

2.6 Codage Python du solver

Nous allons passer maintenant à la dernière partie du solveur avec la classe Solver. Cette dernière étape va mettre en lumière l'utilisation de tous les concepts définis précédemment.

```
1  class Solver():
2
3      def __init__(self,problem):
4      self.problem = problem
5      self.problem.setSolver(self)
6      self.trailSets = []
7      for i in xrange(self.problem.nbVariables()+1):
8          self.trailSets.append(SparseSet(self.problem.nbVariables()))
9      self.futureVars = SparseSet(self.problem.nbVariables(),True)
```

```
10        self.heuristicVar = HeuristicVarDom(self)
11        self.heuristicVal = HeuristicValLexico(self)
12        self.status = const.RUNNING
13        self.nbSolutions = 0
```

```
14      self.nbWishedSolutions = 10**5
15      self.propagation = Propagation(self)
```

Le solver se construit à partir du problème.

- Ligne 5, l'attribut `solver` du problème est modifié afin que le problème connaisse son solver.
- La ligne 6 est très importante. L'attribut `trailSets` est une liste de sparse set. Elle représente l'application qui à un niveau associe les variables dont le domaine est modifié à ce niveau. Cette liste sera très utile au cours de la recherche.
- Ligne 8, l'attribut `futureVars` est la liste des variables non affectées.
- Lignes 9 et 10, deux objets représentent les choix d'heuristique de choix de variables et de valeurs qui seront utilisés par le solver. Ici, à titre illustratif, nous considérons l'heuristique de choix de variable minDom et l'heuristique de choix de valeur `lexico`. Bien sûr d'autres choix seraient possibles [2, 65, 112, 23, 122, 32].
- Ligne 12, une variable permettant de connaître le status de la rechercher est initialisée avec la valeur RUNNING permettant d'indiquer que la recherche commence.
- La ligne 15 est importante, elle concerne le processus de propagation de contraintes. Une classe lui est consacrée.

La classe propagation connait son solver et utilise une queue de propagation.

```
class Propagation():

    def __init__(self,solver):
        self.solver = solver
        nbVars = len(solver.problem.variables)
        self.queue = SparseSet(nbVars)
        self.WO = False
```

Pendant la propagation des variables sont placées sur la queue en attente d'être utilisées pour lancer le filtrage sur les contraintes auxquelles elles appartiennent. Il est à noter qu'il est possible qu'une variable soit replacée plusieurs fois dans le queue (voir chapitre 4). La structure de sparse set est donc à nouveau utilisée.

Il va donc falloir utiliser les filtres de chaque contrainte pour chaque variable jusqu'à ce qu'il n'y ait plus aucune valeur des domaines supprimée.

```
    def propagate(self,x):
      self.queue.add(x)
        while self.queue.size()!=0:
            y = self.pickInQueue()   # choix possible
            for c in y.constraints:
                S = c.filterFrom(y)
                if S = const.WO:
                    return const.WO
                else:
                  self.queue.add(S)
    return const.OK
```

Après l'appel de la procédure de filtrage pour la contrainte c, si la valeur de retour S est égale à la constante WOsymbolisant la détection d'un conflit alors la même constante est retournée par la méthode propagate afin de produire un retour arrière. Sinon, la valeur S représentant l'ensemble des variables dont les domaines ont été modifiés pendant le filtrage, elles sont donc ajoutées à la queue de propagation.

```
    def pickInQueue(self):
        x = self.solver.problem.variables[self.queue.dense[0]]
        self.queue.delete(self.queue.dense[0])
        return x
```

La méthode pickInQueue permet de dépiler une variable de la queue de propagation. Il nous reste à définir une méthode filterFrom générique dans la classe Constraint sachant

qu'elle pourra être redéfinie pour chaque contrainte. La méthode suivante correspond à l'établissement de la propriété GAC.

```
    def filterFrom(self,x):
        evts = []
        for y in (y for y in self.scope if y!=x):
            size = y.dom.size()
            for a in y.dom.iteratorPresent():
                if not self.isThereASupport(y,a):
                    self.problem.solver.delete(y,a)
                    if y.dom.isEmpty():
                        return const.WO
            if size != y.dom.size():
                evts.append(y)
        return evts
```

La méthode a pour paramètre une variable, filtre en vérifiant les supports avec les variables liées par une contrainte et renvoie une liste des variables dont le domaine a été modifié. La méthode isThereASupport cherche s'il y a un support entre deux variables et ne sera pas détaillée ici.

La procédure de recherche se fait dans la classe Solver de la manière suivante :

```
1   def run(self):
2       while self.status == const.RUNNING:
3           x = self.heuristicVar.select()
4           a = self.heuristicVal.select(self.problem.variables[x])
5           self.assignToValueId(x, a)
6           if self.propagation.propagate(x)==const.WO:
7               self.handleFailure(x,a)
8           elif self.futureVars.size()==0:
9               self.nbSolutions += 1
10              if self.nbSolutions == self.nbWishedSolutions:
11                  self.status = const.Succes
12              self.handleFailure(x,a)
13      self.fullBacktrack()
```

La boucle de la ligne 2 s'execute tant que l'attribut status n'est pas modifié. Les lignes 3 et 4 utilisent les heuristiques de variables et de valeurs des domaines afin d'effectuer un branchement.

Ligne 5, une variable est affectée à une valeur. Il faut penser à faire toutes les modifications nécessaires :

```
    def assignToValueId(self,x,a):
        self.futureVars.delete(x)
        self.trailSets[self.getLevel()].clear()
        self.reduceDomain(x, a)
        self.trailSets[self.getLevel()].add(x)
```

Pour affecter une variable X à une valeur v, il faut donc supprimer X des variables non affectées, réduire le domaine de X à v et ajouter X aux variables affectées pour le niveau courant. Après propagation, il y a plusieurs cas.

Premier cas, ligne 8, un des domaines est vide. Il faut alors organiser le retour arrière avec la méthode handleFailure :

```
    def handleFailure(self,x,a):
        self.propagation.queue.clear()
        self.backtrack()
        self.deleteValue(x,a)
        if self.problem.variables[x].dom.size()==0
          or self.propagation.propagate(x) == const.WO:
            if self.getLevel()==0:
                self.status = const.FULL_EXPLORATION
            else:
```

```
            y = self.futureVars.firstAbsent()
            b = self.problem.variables[y].dom.getFirstVal()
            self.handleFailure(y,b)
```

La queue de propagation est vidée, le retour arrière est lancé, la valeur a est supprimée du domaine de X et si le niveau est 0 alors l'exploration est complète.

Nous avons :

```
def backtrack(self):
   for x in self.trailSets[self.getLevel()].iteratorPresents():
  self.problem.variables[x].dom.elements.restoreLimit(self.getLevel())
    self.futureVars.limit += 1
```

Nous voyons que le retour arrière restore la limite du niveau précédent Pour en finir avec la méthode run, la méthode fullBacktrack réalise un retour arrière sur tous les niveaux inférieurs au niveau courant.

```
def fullBacktrack(self):
    for i in xrange(self.getLevel()):
        self.backtrack()
```

La difficulté dans la modélisation d'un problème réside souvent dans la programmation des contraintes. Par exemple, la contrainte binaire $X = Y$ revient à créer une nouvelle classe Contrainte et à surcharger la méthode filterFrom :

```
def filterFrom(self,dummy):
    if self.x.dom.size() == 1:
        v = self.x.dom.getFirstVal()
        self.deleteValue(self.y,v)
    if self.y.dom.size() == 1:
        v = self.y.dom.getFirstVal()
        self.deleteValue(self.x,v)
```

Dans la partie suivante, nous allons étudier un algorithme de filtrage dédié à une contrainte ternaire classique.

2.7 Algorithme de filtrage pour la contrainte $x = y + z$

La contrainte $x = y + z$ est très utilisée. Dans cette partie, nous allons étudier cette contrainte à titre d'exemple ce qui nous permettra de revenir et d'illustrer les mécanismes généraux introduits auparavant pour la propagation de contraintes. Considérons x, y et z trois variables, la contrainte $x = y + z$ est une contrainte ternaire définie sur $dom(x) \times dom(y) \times dom(z)$. Soit $n \in$N, nous nous placerons dans le cas où les domaines sont inclus dans J0, nK.

Pour une variable X, nous noterons $\overline{X}$ et $\underline{X}$ respectivement l'élément maximum et l'élément minimum du domaine courant de X.

Nous allons utiliser trois niveaux de filtrage. La remarque suivante, nous permettra d'obtenir le premier.

La contrainte est $x = y + z$, ce qui implique que $\forall v \in dom(x)$, si v a un support pour cette contrainte alors $v \leq \overline{y} + \overline{z}$. Nous pouvons donc supprimer du domaine de x les valeurs strictement plus grande que $\overline{y} + \overline{z}$. De même, $\forall w \in dom(y)$ si w a un support pour cette contrainte alors $w \leq \overline{x} - \underline{z}$. Nous pouvons donc supprimer du domaine de y les valeurs strictement plus grandes que $\overline{x} - \underline{z}$. En appliquant ces remarques à toutes les variables, nous pouvons obtenir le filtrage donné par l'algorithme 2. Pour simplifier la présentation, on suppose qu'aucun domaine wipe-out ne peut être rencontré. Autrement dit, on considère qu'il existe au moins un support sur la contrainte $x = y + z$ que l'on filtre.

La méthode delValsGT supprime du domaine d'une variable les valeurs strictement supérieures à un entier. De même, la méthode delValsLT supprime du domaine d'une

Algorithme 2 : ZC($x = y + z$)

tant que true **faire**
  x.delValsGT($\overline{y} + \overline{z}$)
  y.delValsGT($\overline{x} - \underline{z}$)
  z.delValsGT($\overline{x} - \underline{y}$)
  x.delValsLT($\underline{y} + \underline{z}$)
  y.delValsLT($\underline{x} - \overline{z}$)
  z.delValsLT($\underline{x} - \overline{y}$)
  si aucune valeur supprimée **alors**
    break
  fin si
fin tant que

variable les valeurs strictement inférieures à un entier. Le filtrage de l'algorithme 2 garantit que le réseau est Z-cohérent aux bornes après son application (voir chapitre 1 section 1.3.2 page 17). En effet, considérons la variable x, instructions x.delValsGT($\overline{y} + \overline{z}$) et x.delValsLT($\underline{y} + \underline{z}$) garantissent que :

$$\underline{y} + \underline{z} \leq \underline{x} \leq \overline{x} \leq \overline{y} + \overline{z}.$$

Ainsi, il existe nécessairement $v \in [\underline{y}, \overline{y}]$ et $w \in [\underline{z}, \overline{z}]$ tel que $\overline{x} = v + w$. Nous pourrions procéder de la même manière avec $\underline{x}$.

Nous allons maintenant passer à un deuxième niveau du filtrage en complétant le premier filtrage et afin de maintenir la D-cohérence aux bornes (voir chapitre 1 section 1.3.2 page 15) .

Nous allons considérer (algorithme 3) un filtrage qui permet de filtrer ou non $\overline{x}$.

Algorithme 3 : $DC_{x,max}(x = y + z)$: Booléen

1: found ← false
2: **pour** $v \in dom(y)$ & **faire**
3: **si** $\overline{x} - v \in dom(z)$ **alors**
4: found ← true
5: break
6: **sinon si** $\overline{x} - v > \overline{z}$ **alors**
7: break
8: **fin si**
9: **fin pour**
10: **si** found = false **alors**
11: supprimer $\overline{x}$ de $dom(x)$
12: **fin si**
13: **retourner** found

Nous pouvons réaliser le même type d'opération pour $\underline{x}$, $\overline{y}$, $\underline{y}$, $\overline{z}$ et $\underline{z}$. Ainsi, nous pouvons compléter le filtrage précédent de manière à obtenir en réseau D-cohérent aux bornes.

Nous voyons dans les algorithmes 3 et 4 qu'un support est cherché à chaque étape. L'algorithme 3 cherche un support pour $\overline{x}$. Ligne 2, une boucle dans l'ordre décroissant sur le domaine de y est implémentée. Ligne 3, nous testons si un support existe dans $dom(z)$. Si c'est le cas, alors il n'y a pas de filtre possible et l'algorithme renvoie immédiatement True. Ligne 6, si $\overline{x} - v > \overline{z}$ alors il n'y a pas support pour ce v et comme les valeurs v

sont prises dans l'ordre décroissant, il n'y en aura pas non plus pour les valeurs suivantes de *v*. La boucle peut donc être arrêtée et la valeur `False` est renvoyée. Ce principe est

Algorithme 4 : $DC_{y,min}(x = y + z)$: Booléen

```
found ← false
pour v ∈dom(z) & faire
  si y + v ∈dom(x) alors
    found ← true
    break
  sinon si y + v < x alors
    break
  fin si
fin pour
si found = false alors
  supprimer y de dom(y)
fin si
retourner found
```

Algorithme 5 : $DC_{all}(x = y + z)$: Booléen

```
ZC(x = y + z)
si DC_{x,max}(x = y + z) = false alors
  retourner false
fin si
si DC_{x,min}(x = y + z) = false alors
  retourner false
fin si
si DC_{y,max}(x = y + z) = false alors
  retourner false
fin si
si DC_{y,min}(x = y + z) = false alors
  retourner false
fin si
si DC_{z,max}(x = y + z) = false alors
  retourner false
fin si
si DC_{z,min}(x = y + z) = false alors
  retourner false
fin si
```

appliqué pour tous les cas dans l'algorithme 5. Lorsqu'un point fixe est trouvé, deux cas se présentent. Un domaine est vide et le problème est inconsitant. Sinon, la D-cohérent aux bornes est atteinte. Par exemple, l'algorithme 3, garantit que pour $\overline{x}$, il existe $v \in dom(y)$ et $w \in dom(z)$ tel que $\overline{x} = v + w$.

Nous allons maintenant ajouter encore un niveau à ce filtrage pour atteindre GAC. Il s'agit simplement d'un filtrage par arc cohérence en une passe sans tester les bornes des domaines. Ce filtrage permet d'obtenir un réseau arc cohérent. En effet, si un triplet (u, v, w) vérifie $u = v + w$ avec $(u, v, w) \in dom(x) \times dom(y) \times dom(z)$ alors (u, x), (v, y) et (w, z) ont un support. Les valeurs supprimées dans les différents domaines ne peuvent amener à filtrer de nouvelles valeurs lors d'une deuxième passe. Ainsi, une seule passe suffit à obtenir un réseau arc cohérent. L'algorithme 6 donne ce dernier niveau de filtrage et l'algorithme 7 donne l'algorithme complet.

Algorithme 6 : passGAC($x = y + z$)

```
pour ∀ξ∈{x, y, z} faire _
  pour ∀v∈dom(ξ) \ {ξ, ξ}_ faire
    si il n'existe pas de support pour (ξ, v) alors
      ξ.delVal(v)
    fin si
  fin pour
fin pour
```

Algorithme 7 : GAC($x = y + z$)

```
tant que true faire
  ZC(x = y + z)
  si DC(x = y + z) = false alors
    break
  fin si
fin tant que
passGAC(x = y + z)
```

Voici donc comment une contrainte particulière peut être étudiée pour optimiser un processus de filtrage. Le gain peut être considérable lorsque la contrainte est fréquemment utilisée.

Cependant, cette optimisation n'est pas sensiblement meilleure que l'utilisation actuelle des contraintes linéaires. Il s'agissait dans cette partie de montrer les différentes étapes qui permettent d'optimiser une contrainte.

Ce chapitre a donc permis de mettre en évidence différentes méthodes intervenant dans les solveurs de contraintes. Le solveur Mistral du projet NumberJack [59] est basé également sur des sparse set et les méthodes exposées ici peuvent permettre de créer et comparer des solveurs, par exemple, dans NumberJack.

Chapitre 3

Isomorphisme de sous-graphe

En informatique, de nombreux objets sont représentés à l'aide de la théorie des graphes (arbres, réseaux, cartes combinatoires, bioinfomatique, etc.) [46, 97, 102, 62, 103]. De très nombreux ouvrages sont consacrés à cette thématique [29, 10, 87, 52, 44]. Très souvent, une relation entre ces structures est cherchée. Mathématiquement, il s'agit de morphismes de graphes. Dans ce chapitre, nous nous intéresserons plus particulièrement à la recherche d'un graphe donné dans un autre graphe. Plus précisément, nous nous intéresserons à la résolution d'une version du problème d'isomorphisme de sous-graphe en utilisant la programmation par contraintes. Les graphes étudiés seront des graphes simples et non orientés. Nous poserons le problème, nous expliquerons la modélisation et nous proposerons un filtrage nouveau pour le problème d'isomorphisme de sous-graphe.

3.1 Définitions

Définition 3.1. Dans cette thèse, un graphe (non-orienté) désignera un couple (S, A) où S est un ensemble dit de **sommets** et où A est un ensemble, dit **d'arêtes**, dont les éléments sont des paires $\{i, j\}$ tel que i et j sont des éléments distincts de S.

Nous dirons que les sommets i et j sont **voisins** ou **adjacents** si $\{i, j\}$ est une arête du graphe.

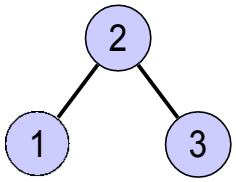
Pour un sommet i, nous noterons $adj(i)$ l'ensemble des sommets voisins de i.

Nous appellerons **degré** d'un sommet le nombre de ses voisins. Pour un sommet i, nous noterons $deg(i)$ sont degré.

Exemple 3.1. Un graphe $G = (S, A)$ avec

$$S = \{1, 2, 3\} \textit{ et } A = \{\{1, 2\}, \{2, 3\}\}$$

sera représenté par le digraphe suivant :



Le sommet 2 a 2 voisins (1 et 3). Ainsi $deg(2) = 2$.

Définition 3.2. Soit $G = (S, A)$ et $G^0 = (S^0, A^0)$ deux graphes.

Nous dirons que G^0 est un **graphe partiel** de G si

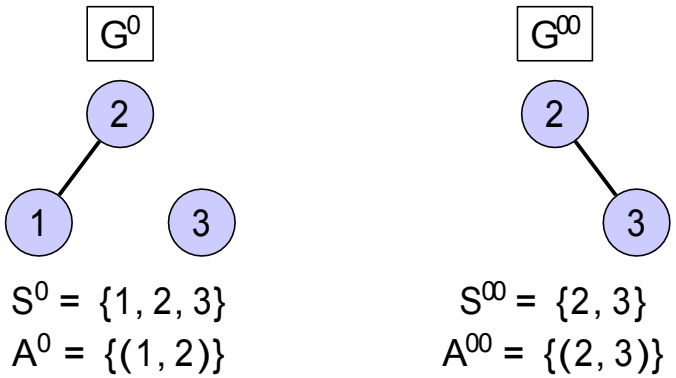
— $S = S^0$,

— $A^0 \subseteq A$.

Nous dirons que G^0 est un **sous-graphe induit** de G si

— $S^0 \subseteq S$,
— $A^0 = \{(a, b) \in A \mid a, b \in S^0\}$.
Nous dirons que G^0 est un **sous-graphe partiel** si G^0 est un graphe partiel d'un sous-graphe induit de G.

Exemple 3.2. Reprenons l'exemple 3.1 et considérons les graphes G^0 et G^{00} définis de la manière suivantes :



Le graphe G^0 est un sous-graphe partiel de G et G^{00} est un sous-graphe induit de G.

Définition 3.3. Soit $G = (S, A)$ un graphe, nous appellerons **chemin** du graphe G une suite finie de sommets du graphe $c = (i_1, \ldots, i_n)$ telle que

$$\forall j \in \llbracket 1, n-1 \rrbracket, \ (i_j, i_{j+1}) \in A.$$

Le sommet i_1 est alors appelé **origine** du chemin c et i_n est appelé **but** du chemin c.

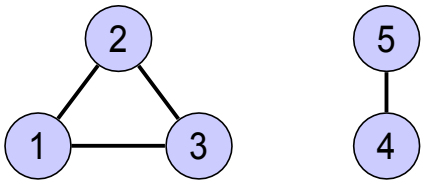
Remarque 3.1. La définition précédente n'est valable que pour la catégorie de graphes définis ici.

Exemple 3.3. Dans l'exemple 3.1, la suite finie $c = (1, 2, 3)$ est un chemin d'origine 1 et de but 3.

Définition 3.4. Nous dirons qu'un graphe $G = (S, A)$ est connexe si deux sommets donnés sont toujours reliés par un chemin.

Nous appellerons **composante connexe** de G les sous-graphes maximaux pour la propriété de connexité.

Exemple 3.4. Le graphe suivant a deux composantes connexes données par les ensembles de sommets $\{1, 2, 3\}$ et $\{4, 5\}$.



Définition 3.5. Soit $G = (S, A)$ un graphe avec $n := |S|$. Nous appellerons **matrice d'adjacence** et nous noterons M_G la matrice carrée de taille $n \times n$ dont le coefficient correspondant à la ligne i et à la colonne j est égale à 1 si $(i, j) \in A$ et 0 sinon.

Notons, qu'avec cette définition, la matrice dépend d'un ordre donnée aux sommets.

Exemple 3.5. Dans l'exemple 3.1 et avec l'ordre 1, 2, 3, la matrice d'adjacence est

$$\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$$

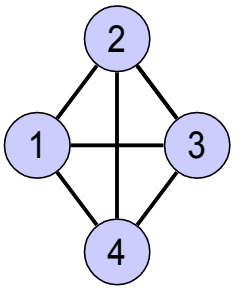
$$\begin{matrix} \square & 1 & 0 & 1 & \square \\ & 0 & 1 & 0 & \end{matrix}.$$

Définition 3.6. Nous dirons qu'un graphe $G = (S, A)$ est une **clique** si

$$\forall i,j \in S,\ (i,j) \in A.$$

Nous parlerons alors de clique d'ordre n, où $n = |S|$.

Exemple 3.6. Le graphe suivant est une clique d'ordre 4.



Nous voyons qu'une clique est très dense en terme d'arête. Il existe un coefficient, appelé coefficient de clustering qui mesure cette densité localement ou globalement.

Définition 3.7. Dans un graphe $G = (S, A)$, nous appellerons **triangles** les sous-graphes qui sont des cliques d'ordre 3. Nous appellerons **triplets connectés** les sous-graphes connexes de trois sommets.

Nous appellerons **coefficient de clustering global** de G le nombre

$$\frac{3 \times nbT}{nbC}$$

où nbT est le nombre de triangles du graphe et nbC le nombre de triplet.connectés.

Nous appellerons **coefficient de clustering** du sommet i le nombre

$$\frac{2 \times nbT(i)}{deg(i) \times (deg(i) - 1)}$$

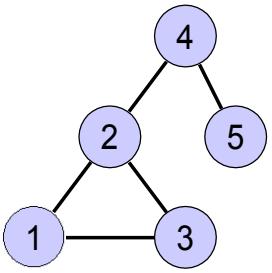
où $nbT(i)$ est le nombre de triangles du graphe contenant i .

Nous appellerons **coefficient de clustering moyen** le nombre

$$\frac{1}{|S|} \sum_{i \in S} c_i$$

où c_i est le coefficient de clustering de i.

Exemple 3.7. Considérons le graphe suivant :



Les triplets connectés sont donnés par les ensembles de sommet $\{1, 2, 3\}$, $\{1, 2, 4\}$ et $\{2, 4, 5\}$, $\{2, 3, 4\}$. Seul l'ensemble $\{1, 2, 3\}$ représente un triangle. Le coefficient global $\frac{3}{4}$ $\frac{2}{4}$ $\frac{1}{4}$

de clustering est alors $_4$. De même le coeffi cient de clustering du sommet 2 est $_6 = _3$.

Le coeffi cient de clustering moyen est alors $\frac{1}{5}(1 + 1 + \frac{1}{3} + 0 + 0) = \frac{7}{15}$.

3.2 Le problème d'isomorphisme de sous-graphe

Définition 3.8. Un morphisme d'un graphe $G_s = (S_s, A_s)$ dans un graphe $G_c = (S_c, A_c)$ est une application $f : S_s \rightarrow S_c$ tel que :

$$\forall \{i, j\} \in A_s, \{f(i), f(j)\} \in A_c$$

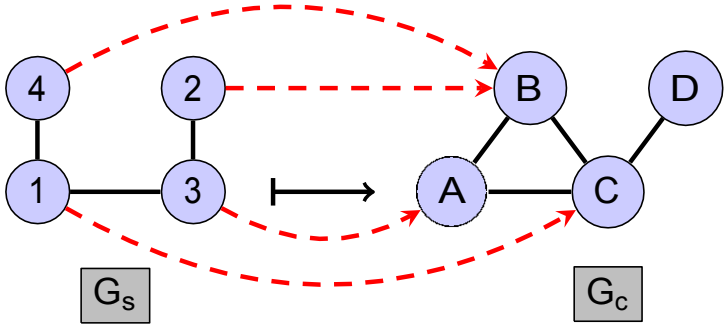
Exemple 3.8. Considérons le graphe $G_s = (S_s, A_s)$ défini par :

$$S_s = \{1, 2, 3, 4\} \text{ et } A_s = \{\{4, 1\}, \{1, 3\}, \{2, 3\}\}$$

et $G_c = (S_c, A_c)$ défini par :

$$S_c = \{A, B, C, D\} \text{ et } A_c = \{\{A, B\}, \{B, C\}, \{A, C\}, \{C, D\}\}.$$

Le morphisme qui associe 1 à C, 2 à B, 3 à C et 4 à B est un morphisme du graphe G_s dans le graphe G_c.



Définition 3.9. Un morphisme de graphe f d'un graphe $G_s = (S_s, A_s)$ dans un graphe $G_c = (S_c, A_c)$ est dit **injectif** si tout élément de S_c a au plus un antécédent par f.

Exemple 3.9. Considérons le graphe $G_s = (S_s, A_s)$ défini par :

$$S_s = \{1, 2, 3\} \text{ et } A_s = \{\{1, 2\}, \{1, 3\}, \{2, 3\}\}$$

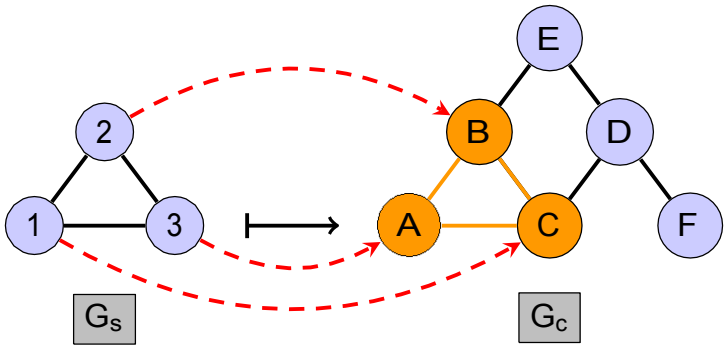
et le graphe $G_c = (S_c, A_c)$ défini par :

$$S_c = \{A, B, C, D, E, F\}$$

et

$$A_c = \{\{A, B\}, \{B, C\}, \{A, C\}, \{B, E\}, \{D, E\}, \{C, D\}, \{D, F\}\}.$$

Le morphisme qui associe 1 à A, 2 à B et 3 à C est un morphisme injectif du graphe G_s dans le graphe G_c.



Nous voyons dans l'exemple précédent qu'un morphisme de graphe injectif implique un isomorphisme du graphe source dans un sous-graphe du graphe cible. Autrement dit, un sous-graphe du graphe cible est isomorphe au graphe source. Le problème d'isomorphisme de sous-graphe est la recherche dans un graphe cible d'un sous-graphe isomorphe à un graphe source.

3.3 Modélisation du problème à l'aide de la programmation par contraintes

Nous allons nous intéresser aux morphismes de graphes injectifs. C'est-à-dire les morphismes tels qu'un sommet image a au plus un antécédant. L'objectif est de reconnaître un graphe donné dans un graphe plus grand.

Notons qu'il s'agit, ici, du problème d'isomorphisme de sous-graphe partiel et non induit. Comme nous pouvons le voir sur la figure 3.1, cette condition relâche un peu les contraintes sur le problème puisqu'il n'est pas nécessaire pour le graphe source d'avoir toutes les arêtes d'un sous-graphe induit cible.

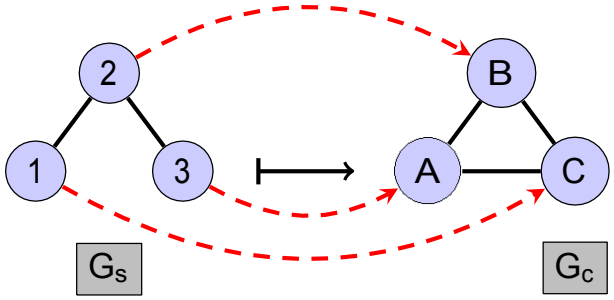


Figure 3.1 : *Convient au problème de graphe partiel mais pas au problème de graphe induit.*

Dans la suite de ce chapitre, nous noterons $G_s = (S_s, A_s)$ un graphe appelé **graphe source** et $G_c = (S_c, A_c)$ un graphe appelé **graphe cible**. Nous nous placerons dans le cas où nous cherchons à trouver un sous-graphe du graphe cible isomorphe au graphe source. Le morphisme de graphe induit sera noté f.

Nous modéliserons ce problème à l'aide du réseau de contraintes $R = (X, C)$ défini par :

$$X = \{X_i \mid i \in S_s\}$$

où $\forall i \in S_s$, la variable X_i a pour domaine S_c, c'est-à-dire $dom(X_i) = S_c$. De plus,

$$C = \{C_{i,j} \mid \{i, j\} \in A_s\} \cup \text{AllDiff}(X)$$

où $C_{i,j}$ est la contrainte de scope $\{X_i, X_j\}$ telle que $C_{i,j}(x, y) = 1$ ssi $\{f(x), f(y)\} \in A_c$.

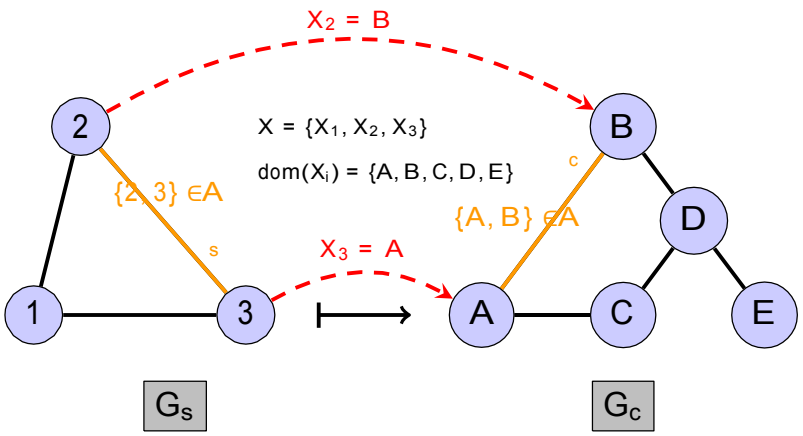
Les contraintes $C_{i,j}$ garantissent que f est un morphisme de graphe. C'est-à-dire,

$$\{i, j\} \in A_s \Rightarrow \{f(i), f(j)\} \in A_c.$$

La contrainte AllDiff(X) garantit l'injectivité de f. En effet, si i et j sont deux sommets distincts de S_s alors les variables X_i et X_j sont distinctes et donc les affectations $f(i)$ et $f(j)$ sont distinctes.

Exemple 3.10. Dans la figure ci-dessous, le graphe G_s est une clique contenant les sommets 1, 2 et 3. Le graphe $G_c = (S_c, A_c)$ est défini par :

— $S_c = \{A, B, C, D, E\}$
— $A_c = \{\{A, B\}, \{A, C\}, \{B, D\}, \{D, C\}, \{D, E\}\}$.



Le graphe G_s contient trois sommets, donc le réseau de contraintes contient trois variables indexées par ces sommets : X_1, X_2 et X_3. Chaque domaine des variables est égale à A_c. De plus,

$$C = \{C_{1,2}, C_{2,3}, C_{1,3}, \text{AllDiff}(X_1, X_2, X_3)\}.$$

Par exemple, la contrainte $C_{2,3}$ est définie par :

— $scp(C_{2,3}) = \{X_2, X_3\}$.

— $(X_2, X_3) \in A_c$, autrement dit $\forall x,y \in S_c$, $C_{2,3}(x, y) = 1$ si et seulement si $(x, y) \in A_c$.

Il existe d'autres manières de modéliser le problème de sous-isomorphisme de graphe à l'aide de la programmation par contraintes. Cependant cette méthode est la plus généralemen t utilisée (voir [111, 80, 61, 66, 121, 108]).

Dans la suite de ce chapitre, nous nous placerons dans le problème isomorphisme de sous-graphe de $G_s = (S_s, A_s)$ dans $G_c = (S_c, A_c)$ pour lequel le réseau de contraintes, noté $R = (X, C)$, sera modélisé comme ci-dessus.

3.4 Évolution des méthodes de résolution

Depuis de nombreuses années, des travaux ont été réalisés pour améliorer les performances de résolution du problème de sous-isomorphisme de graphe avec le programmation par contraintes.

Le premier algorithme d'isomorphisme de sous-graphe date des années 70 [37, 111]. De manière générale, les méthodes utilisées construisent le couplage entre deux sommets de manière récursive et utilisent des techniques de filtrage de type look-ahead pour supprimer les valeurs incompatibles. Des progrès très important ont été réalisés depuis les années 70. L'article [35] montre une accélération au moins polynomiale comparée à l'ap-proche de [111]. Pour montrer cela, pour chaque méthode, des problèmes sont représentés comme des points dont l'abscisse correspond au nombre de sommets du graphe cible et dont l'ordonnée correspond au temps mis pour résoudre le problème avec une échelle lo-garithmique en abscisse et en ordonnée. Ces nuages de points montrent des alignements avec une pente pour l'algorithme présenté, nommé VF2, nettement meilleure que ce qui existait jusqu'alors.

D'autres méthodes sont basées sur le calcul des automorphismes de graphes ou encore sur les techniques de raffi nement par coloriage (voir [119]) qui sont utilisées pour partitionner les sommets par rapport à des invariants (comme le degré).

Les méthodes incomplètes basées sur des (méta-)heuristiques sont également très po-pulaires pour résoudre des problèmes diffi ciles ou de grande taille. Elles sont par exemple très largement utilisées dans la littérature liée aux problèmes de *matching* de graphes (des centaines de références sont données dans [25, 31]). Par ailleurs, certains de

œs algo-rithmes peuvent être adaptés au cas du pur problème d'isomorphisme de (sous-)graphe.

Par exemple, l'heuristique définie dans [89] essaie de minimiser la mesure de similarité entre deux graphes, et atteindre la valeur 0 correspond en conséquence à trouver un isomorphisme. Dans cette thèse, les méthodes incomplètes ne seront pas développées.

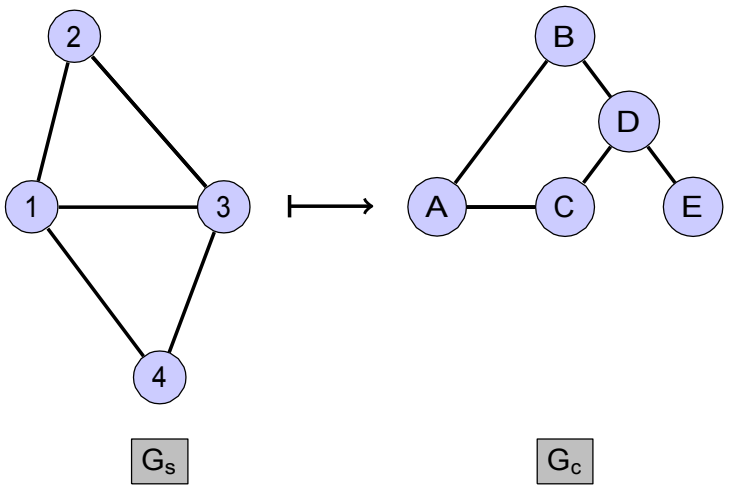
En ce qui concerne les méthodes complètes, une direction de recherche consiste à utiliser des techniques de filtrage issus de la programmation par contraintes (voir [108, 121, 86] pour des développements récents). En effet, le problème d'isomorphisme de (sous-)graphe peut facilement être transformé en un problème de satisfaction de contraintes. On peut alors exploiter des propriétés liées aux contraintes et réseaux de contraintes comme par exemple la cohérence d'arc (AC) ou la singleton cohérence d'arc (SAC) [69]. Afin de renforcer le processus d'inférence, des techniques de filtrage spécifique au problème d'isomorphisme de (sous-)graphes ont été proposées, comme par exemple ILF [121] et LAD [108] qui utilisent des informations liées aux degrés, des multi-ensembles ou encore une relation de voisinage pour partitionner les sommets et ainsi filtrer les domaines.

Dans la suite de ce chapitre, nous allons décrire quelques méthodes existantes de résolution du problème d'isomorphisme de sous-graphe par la programmation par contraintes. Puis nous proposons une approche générale pour raisonner sur la dominance entre sommets selon deux axes principaux : tout d'abord, en associant un ensemble de scores à chaque couple de sommets et en considérant ensuite pour raisonner diverses façons de définir le voisinage entre sommets. Nous nommons cette approche SND pour *Scoring-based Neighborhood Dominance*. Nous montrons que LAD est un cas particulier de SND et nous étudions des spécialisations de SND en considérant le nombre de chemins de longueur k dans les graphes et trois façons de relier les ensembles de sommets. Avec ces spécialisations, nous montrons que SND est plus fort que LAD et incomparable à SAC. Enfin, l'étude expérimentale que nous avons menée démontre l'efficacité de notre approche.

3.4.1 Forward Checking

Comme nous l'avons vu au chapitre 1 page 20, l'algorithme Forward Checking peut être utilisé de manière naturelle sur les contraintes binaires. Sur les problèmes non binaires, les définitions sont multiples et l'efficacité n'est souvent pas évidente. Notre modélisation fait apparaître deux types de contraintes binaires. Il a donc été naturel d'utiliser l'algorithme Forward Checking. Cependant, la contrainte $AllDiff$ est une contraintes globale. Dans ce cadre, nous considèrerons la contraintes $AllDiff$ comme une clique de contraintes binaires de la forme $X = Y$.

Exemple 3.11. Considérons le problème de sous-isomorphisme de graphe schématisé par la figure suivante :



avec $X = \{X_1, X_2, X_3, X_4\}$ et $dom(X_1) = dom(X_2) = dom(X_3) = dom(X_4) = \{A, B, C, D, E\}$. Supposons que l'algorithme assigne X_2 à A. Alors Forward Checking va passer en revue

les contraintes dont le scope contient X_2. Il y a donc les contraintes $C_{1,2}$ et $C_{2,3}$ puis les contraintes $X_2 = X_3$ et $X_1 = X_2$.

Forward Checking appliqué aux contraintes $X_2 = X_3$ et $X_1 = X_2$ supprime A de $dom(X_1)$ et $dom(X_2)$.

Forward Checking appliqué aux contraintes $C_{1,2}$ et $C_{2,3}$ supprime D et E des domaines de X_1 et X_3. Ainsi,

$$dom(X_2) = \{A\},\ dom(X_1) = dom(X_3) = \{B, C\} \text{ et } dom(X_4) = \{A, B, C, D, E\}.$$

Nous pouvons remarquer que Forward Checking a un effet très local sur le graphe. L'heuristique de choix de variables a intérêt à choisir en priorité les sommets ayant un coefficient de clustering élevé.

Comme nous le verrons dans la partie suivante, c'est une remarque qui ne s'applique pas à l'algorithme GAC puisque toutes les variables seront touchées de manière égale.

Nous avons utilisé ici le même algorithme pour filtrer les variables des domaines mais rien ne nous empêche d'utiliser Forward Checking sur un sous-ensemble de contraintes et GAC sur autre sous-ensemble. Nous obtenons alors un niveau de filtrage intermédiaire. Comme dans [108], nous pourrons noter $FC(AllDiff)$ la propagation de Forward Checking pour les contraintes de type $AllDiff$ (en considérant la décomposition de la contrainte $AllDiff$ en contraintes binaires).

La notation $FC(Edges)$ est réservée à la propagation de Forward Checking pour les contraintes $C_{i,j}$. Avec les mêmes notations, nous avons $GAC(AllDiff)$ et $GAC(Edges)$. Ainsi différentes combinaisons sont possibles.

Ici, nous nous sommes intéressés à $FD(AllDiff) + FC(Edges)$.

3.4.2 GAC

L'algorithme GAC va permettre de toucher plus largement les variables que Forward Checking. Nous allons présenter ici $GAC(AllDiff) + GAC(Edges)$.

Exemple 3.12. Considérons à nouveau le problème de problème de l'exemple 3.11. Supposons toujours que l'algorithme assigne X_2 à A. L'algorithme $GAC(AllDiff)$ supprime le valeur A des domaines de X_1, X_3 et X_4. Ensuite, GAC appliqué aux contraintes $C_{1,2}$ et $C_{2,3}$ réduit les domaines de X_1 et X_3 à $\{B, C\}$. La contrainte $C_{1,4}$ supprime E du domaine de X_4. Nous avons en particulier $dom(X_1) = dom(X_3) = \{B, C\}$ et donc la contrainte $C_{1,3}$ est violée. Il faut alors effectuer un retour arrière et supprimer A du domaine de X_2.

Nous voyons ici l'algorithme GAC agit de manière plus globale sur le graphe, supprime plus de valeurs et va donc trouver des états inconsistants plus rapidement.

3.4.3 LV2002

Larrosa et Valiente exploitent dans [66] le fait qu'un sommet i du graphe source ne peut être affecté à un autre sommet j du graphe cible que si le nombre de voisins de i est inférieur ou égal au nombre de voisins de j qui sont au moins dans un domaine d'une variable indexée par un voisin de i.

Pour cela, pour tout sommet i du graphe source et tout sommet j du graphe cible, ils considèrent l'ensemble

$$F(i, j) = \cup_{i_0 \in adj(i)} (dom(X_{i_0}) \cap adj(j)).$$

On peut supprimer j du domaine de X_i si $|F(i, j)| < |adj(i)|$.

Exemple 3.13. Considérons à nouveau le problème de l'exemple 3.11 mais avec les domaines suivants :

$$dom(X_1) = dom(X_2) = \{A, B, C, D, E\},\ dom(X_3) = \{B, C, D, E\} \text{ et } dom(X_4) = \{B, D, E\}.$$

Nous avons :

$$\begin{aligned} F(1, D) &= dom(X_2) \cap dom(X_3) \cap dom(X_4) \cap adj(D) \\ &= \{A, B, C, D, E\} \cap \{B, C, D, E\} \cap \{B, D, E\} \cap \{B, C, E\} \\ &= \{B, E\} \end{aligned}$$

Nous avons donc $|adj(1)| = |\{2, 3, 4\}| = 3$ et $|F(1, D)| = 2$. Ainsi, nous pouvons supprimer D du domaine de X_1.

3.4.4 ILF

Dans cette section, nous allons présenter l'algorithme $ILF(k)$ pour Iterated Labelling Filtering. Dans un problème d'isomorphisme de sous-graphe, différents critères comme le degré, le coefficient de clustering, le nombre de cliques passant par un sommet peuvent être utilisés pour filtrer les domaines. Par exemple un sommet ne peut être assigné à un sommet de degré inférieur. Zampelli et *al* ont proposés dans [121] de généraliser ce principe en utilisant ce qu'ils nomment des étiquetages. Ils itèrent ensuite ce principe. Par exemple, si un sommet est affecté à un autre avec un degré compatible, il faut que les sommets voisins soient également compatibles. Il est donc possible de propager ce raisonnement à tous les sommets du graphe cible. Dans cette partie, nous expliquerons les principes de $ILF(k)$. Pour plus de détails, le lecteur pourra consulter [121].

Tout d'abord, nous avons besoin de la notion d'étiquetage.

Définition 3.10. Un **étiquetage** relatif à G_s et G_c est un triplet $(L, , \alpha)$ tel que :
- $(L,)$ est un ensemble partiellement ordonné, dit d'étiquettes,
- α est une application de $S_s \cup S_c$ dans L.

Un étiquetage permet de prendre en compte différents critères. L'application α peut représenter le degré, le nombre de cliques, le nombre de chemins, etc.

Si l'étiquetage est bien défini, pour un sommet i de G_s, les sommets j de $dom(X_i)$ pourront être supprimés lorsque $\alpha(i) \quad \alpha(j)$.

Exemple 3.14. Considérons à nouveau le problème de problème de l'exemple 3.11. Si l'application α représente le degré alors A est supprimé du domaine de X_1. En effet, $deg(1) = 3$ et $deg(A) = 2$ donc $deg(1) > deg(A)$.

Zampelli et *al* vont plus loin en construisant l'extension d'un étiquetage de manière à pouvoir itérer ce processus. La définition 3.12 donne l'extension d'un étiquetage.

Notation. Nous utiliserons la double accolade pour désigner les multi-ensembles. Par exemple, $\{\{1, 1, 1, 2\}\}$ désignera le multi-ensemble contenant 1 avec une multiplicité de 3 et 2 avec une multiplicité de 1.

Définition 3.11. Soit m et m^0 deux multi-ensembles sur un ensemble E muni d'une relation d'ordre partiel . Un ordre partiel sur les multi-ensembles, que nous noterons encore par abus, est induit par :

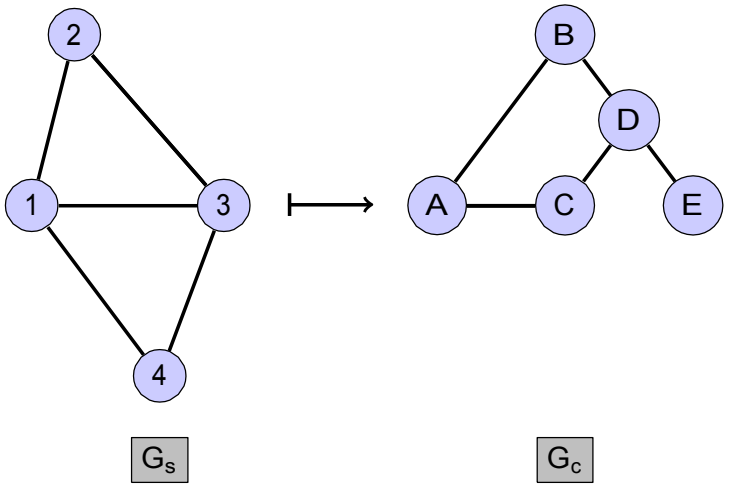
$$m \quad m^0 \iff \text{il existe une injection } t : m \to m^0 \text{ telle que } \forall v \in m, \; v \; t(v).$$

Définition 3.12. Soit $e = (L, , \alpha)$ un étiquetage pour G_s et G_c. L'extension de e est l'étiquetage $e^0 = (L^0, {}^0, \alpha^0)$ défini par :
- L^0 est l'ensemble des $\alpha(v).\{\{\alpha(v^0), \; v^0 \in adj(v)\}\}$ pour $v \in S_s \cup S_c$ et où . désigne la concaténation,
- à tout sommet $v \in S_s \cup S_c$, l'application α^0 associe $\alpha(v).m$ où m est le multi-ensemble constitué des éléments l de L et dont la multiplicité est $|\{u \mid (u, v) \in A_s \cup A_c \text{ et } \alpha(u) = l\}|$,

— la relation d'ordre partiel sur L^0 est définie par $l_1.m_1 \ ^0 l_2.m_2$ si $l_1 \ l_2$ et $m_1 \ m_2$.

Cette extension permet d'étendre l'étiquetage jusqu'à un ordre k. C'est le principe de $ILF(k)$.

Exemple 3.15. Reprenons le problème :



Considérons ici, que l'application α correspond au degré. Nous avons

$$deg(2) = deg(4) = 2 \text{ et } deg(1) = deg(3) = 3.$$

et

$$deg(E) = 1,\ deg(A) = deg(B) = deg(C) = 2 \text{ et } deg(D) = 3.$$

Nous avons alors

$$\begin{aligned} \alpha^0(1) &= 3.\{\{2, 2, 3\}\} \\ \alpha^0(2) &= 2.\{\{2, 3, 3\}\} \\ \alpha^0(3) &= 3.\{\{2, 2, 3\}\} \\ \alpha^0(4) &= 2.\{\{2, 3, 3\}\}. \end{aligned}$$

et

$$\begin{aligned} \alpha^0(A) &= 2.\{\{2, 2\}\} \\ \alpha^0(B) &= 2.\{\{2, 3\}\} \\ \alpha^0(C) &= 3.\{\{2, 3\}\} \\ \alpha^0(D) &= 3.\{\{1, 2, 2\}\} \\ \alpha^0(E) &= 1.\{\{2, 3\}\}. \end{aligned}$$

Nous voyons que le domaine de X_1 est réduit à $\{D\}$. En effet, α (le degré) permet d'éliminer A, B et E. De plus, α^0 (le multi-ensemble) permet d'éliminer C. Mieux encore, le domaine de X_2 est immédiatement vide. En effet α^0 permet d'éliminer D du domaine de X_2.

3.4.5 LAD

Dans [108], C. Solnon utilise des relations de voisinage pour partitionner les sommets et filtrer les domaines. Cela s'appuie sur une première remarque. Si f est un morphisme de graphe injectif de G_s dans G_c alors $\forall i \in S_s$, nous avons :

— $\forall i' \in adj(i),\ f(i') \in adj(f(i))$,
— $\forall i', i'' \in adj(i),\ i' = i'' \Rightarrow f(i') = f(i'')$.

Ces propriétés sont exprimées pour tout sommets $i \in S_s$ et $j \in S_c$ par les contraintes $L_{i,j}$ définies par :

$$X_i = j \Rightarrow \forall i^0 \in adj(i),\ X_{i^0} \in adj(j) \wedge AllDiff(\{X_{i^0} \mid i^0 \in adj(i)\}).$$

Notons que $L_{i,j}$ est une contrainte de scope $X_i \cup \{X_{i^0},\ i^0 \in adj(i)\}$.

Exemple 3.16. Considérons à nouveau le problème de problème de l'exemple 3.11. La contrainte $L_{2,A}$ est de scope $\{X_1, X_2, X_3\}$ et est définie par :

$$X_2 = A \Rightarrow X_1 \in \{B, C\},\ X_3 \in \{B, C\} \wedge AllDiff(X_1, X_3).$$ Ce

qui formellement signifie :

$$L_{2,A}(X_1, X_2, X_3) = 1 \iff X_2 = A \text{ ou } X_2 = A \text{ et } X_1 \in \{B, C\},\ X_3 \in \{B, C\} \text{ et } AllDiff(X_1, X_3) = 1.$$

Une idée forte de cette méthode est la possibilité d'adapter un algorithme classique de filtrage basé sur le calcul d'un couplage maximum.

Définition 3.13. Soit G un graphe non-orienté. Un **couplage** sur G est un ensemble constitué d'arêtes de G sans sommet commun.

Un graphe sera appelé **graphe biparti** si l'ensemble des sommets du graphe peut être partitionné en deux ensembles sous la forme $S_1 \,\dot\cup\, S_2$ et tel que chaque arête soit constituée d'un sommet de S_1 et d'un sommet de S_2. Dans ce cas, nous parlerons de graphe biparti pour la partition $S_1 \,\dot\cup\, S_2$.

Si G est un graphe biparti pour la partition $S_1 \,\dot\cup\, S_2$, nous dirons qu'un couplage c pour G est un **couplage couvrant** S_1 si tout élément de S_1 appartient à une arête de c.

Définition 3.14. Soit $(i, j) \in S_s \times S_c$ tel que $j \in dom(X_i)$ nous noterons $G_{i,j} = (S_{i,j}, A_{i,j})$ le graphe biparti défini par :

— $S_{i,j} = adj(i) \,\dot\cup\, adj(j)$,

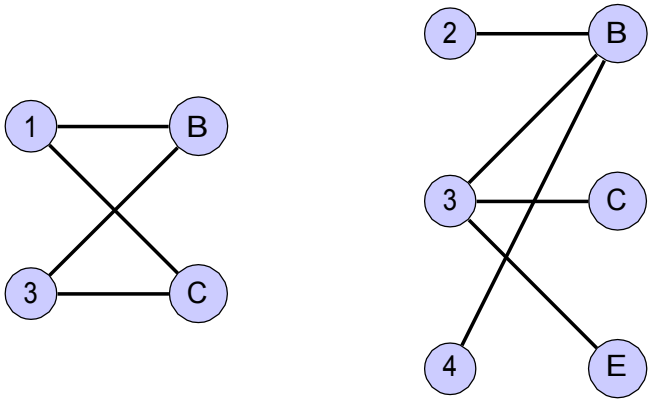
— $A_{i,j} = \{(i^0, j^0) \in adj(i) \times adj(j) \mid j^0 \in dom(X_{i^0})\}$.

Le sommet j est alors supprimé de $dom(X_i)$ s'il n'existe pas de couplage couvrant $adj(i)$ dans le graphe biparti $G_{i,j}$ pour la partition $adj(i) \,\dot\cup\, adj(j)$.

Exemple 3.17. Considérons à nouveau le problème de problème de l'exemple 3.11 avec :

$$dom(X_1) = dom(X_3) = \{A, B, C, D, E\} \text{ et } dom(X_2) = dom(X_4) = \{A, B, D\}.$$

Nous avons, par exemple, $G_{2,A}$ et $G_{1,D}$:



$G_{2,A}$ $G_{1,D}$

Le graphe biparti $G_{2,A}$ admet un couplage couvrant $\{1, 3\}$ donc ne permet pas de réaliser de filtrage. En revanche, $G_{1,D}$ n'admet pas de couplage couvrant $\{2, 3, 4\}$ donc nous pouvons supprimer D du domaine de X_1.

3.5 Stratégie de dominance de score au voisinage

3.5.1 Principe et exactitude

Nous proposons à présent de généraliser le raisonnement sous-jacent à LAD selon deux axes : tout d'abord, en associant un ensemble de scores à chaque couple de sommets d'un graphe et ensuite en considérant différentes formes de voisinage (contrairement à LAD, ces ensembles ne correspondent pas nécessairement aux sommets adjacents d'un sommet donné). Nous nommons cette approche SND (pour Scoring-based Neighborhood Dominance). Pour la suite, nous aurons besoin de quelques définitions.

Définition 3.15. Nous appellerons **fonction de voisinage** une fonction qui à tout graphe $G = (S, A)$ associe une fonction de S dans $P(S)$.

Nous appellerons **fonction de score** qui à tout graphe $G = (S, A)$ associe une fonction de $S \times S$ dans R.

Exemple 3.18. Les fonctions suivantes sont des fonctions de voisinage :
— $\mathrm{Id} : G \to (i \to \{i\})$,
— $\mathrm{adj} : G \to (i \to adj(i))$,
— $\mathrm{all} : G \to (i \to S \setminus \{i\})$.

SND peut être vu comme une approche générique pouvant être paramétrée par deux éléments notés S pour le score et V pour le voisinage. Une spécialisation $\mathrm{SND}_{S,V}$ de SND est donc définie par un ensemble de fonctions de score S et un ensemble de fonctions de voisinage V.

Comme c'est le cas pour LAD, pour chaque couple (i_s, i_c) de sommets motif/cible, nous considérons une contrainte implicite. Elle est notée $\mathrm{SND}_{S,V}(i_s, i_c)$.

Pour un élément $T \in S$, par abus de notations et lorsque cela ne prête pas à confusion, nous noterons simplement $T(i, i^0)$ à la place de $T(G)(i, i^0)$.

Définition 3.16. Nous noterons $\mathrm{SND}_{S,V}(i_s, i_c)$ la contrainte dont la portée est

$$\{X_{i_s}\} \bigcup_{V \in V} \{X_{i^0_s} \mid i^0_s \in V(i_s)\}$$

et dont la sémantique est définie comme suit :

$$\begin{aligned} X_{i_s} = i_c \Rightarrow \bigvee_{V \in V} \quad & AllDiff(\{X_{i^0_s} \mid i^0_s \in V(i_s)\}) \\ & \wedge X_{i^0_s} \in V(i_c), \forall i^0_s \in V(i_s) \\ & \wedge T(i_s, i^0_s) \leq T(i_c, X_{i_s}), \forall i^0_s \in V(i_s), \forall T \in S \end{aligned} \tag{3.1}$$

Définition 3.17. Soient $i_s, i^0_s \in S_s$ et $i_c, i_c{}^0 \in S_c$ alors nous dirons que (i_s, i_s) est **dominé** par (i_c, i^0_c) si $\forall T \in S,\ T(i_s, i_s) < T(i_c, i_c)$.

Remarque 3.2. Lorsque la condition de l'équation 3.1 n'est pas vérifiée (dans le contexte d'une fonction de voisinage V), (i_s, i_s) n'est pas dominé par (i_c, i_c).

S'il n'y a pas d'interprétation possible de $\{X_{i^0_s} \mid i^0_s \in V(i_s)\}$ telle que chaque couple (i_s, i^0_s) soit dominé, on peut alors conclure que $X_{i_s} = i_c$. Bien entendu, la façon dont on calcule les scores et la façon dont on définit les voisinages doivent être cohérentes par rapport au réseau de contraintes. En d'autres termes, chaque contrainte $\mathrm{SND}_{S,V}(i_s, i_c)$ ajoutée à C doit être une conséquence de R.

Définition 3.18. Nous dirons qu'une contrainte de la forme $\mathrm{SND}_{S,V}(i_s, i_c)$ est **impliquée** (par rapport à R) ssi

$$sols(R) = sols(R \oplus \mathrm{SND}_{S,V}(i_s, i_c))$$

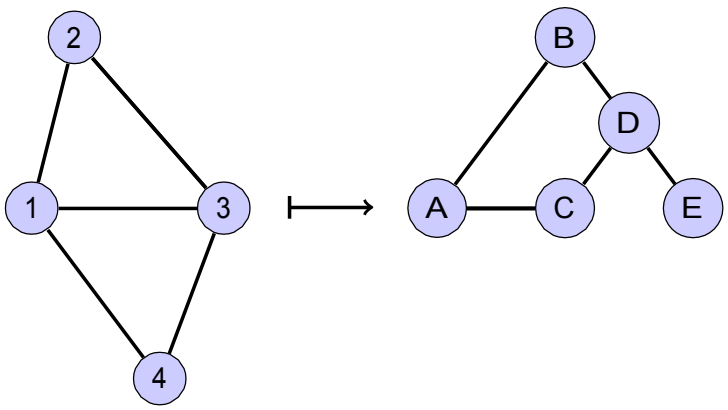
où $R \oplus c$ est le réseau de contraintes R avec la contrainte additionnelle c.

Nous allons donc ajouter ce type de contraintes au réseau de contraintes initial. Dans ce chapitre, des combinaisons originales de fonctions de score et de voisinage sont proposées. Tout d'abord, le nombre de chemins entre deux sommets sera utilisé comme fonction de score.

Il est bien connu que si M_G est la matrice d'adjacence associée à un graphe G, alors $M_G^k[i][i^0]$ indique le nombre de chemins de longueur k dans G qui relient i à i^0.

L'idée sous-jacente est qu'il n'est pas possible d'associer une paire de sommets du graphe motif $\{i_s, i^0_s\}$ à une paire $\{i_c, i^0_c\}$ du graphe cible si pour un certain k on a $M^k_{G_s}[i_s][i^0_s] > M^k_{G_c}[i_c][i^0_c]$. En conséquence, cette observation peut être utilisée lors du filtrage.

Exemple 3.19. Considérons à nouveau le problème suivant :



Nous avons les matrices d'adjacence suivantes :

$$M_{G_s} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad M_{G_c} = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Nous voyons sur ces matrices qu'il n'est, par exemple, pas possible d'associer la paire $\{1, 2\}$ à la paire $\{A, E\}$. En effet, les sommets 1 et 2 sont liés dans G_s alors que A et E ne le sont pas dans G_c. Ce qui se voit matriciellement car $1 = M_{G_s}[1][2] > M_{G_c}[A][E] = 0$. Il est possible d'étendre ce raisonnement aux chemins plus longs. Nous avons :

$$M^2_{G_s} = \begin{pmatrix} 3 & 1 & 2 & 1 \\ 1 & 2 & 1 & 2 \\ 2 & 1 & 3 & 1 \\ 1 & 2 & 1 & 2 \end{pmatrix} \quad M^2_{G_c} = \begin{pmatrix} 2 & 0 & 0 & 2 & 0 \\ 0 & 2 & 2 & 0 & 1 \\ 0 & 2 & 2 & 0 & 1 \\ 2 & 0 & 0 & 3 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Ces matrices donnent le nombre de chemins de longueur 2 dans chaque graphe. Nous voyons qu'il n'est pas possible d'affecter la paire de sommets $\{2, 4\}$ à la paire de sommets $\{B, E\}$. En effet, il y a 2 chemins de longueur 2 entre 2 et 4 alors qu'il n'y a qu'un seul chemin de

longueur 2 entre B et E. Ceci se voit matriciellement car $2 = M_G\,[2][4] > M_G\,[B][E] = 1$. Ce raisonnement est valable quelque soit la longueur des chemins.

Définition 3.19. Soit k un nombre entier naturel. Nous noterons M^k la fonction de score définie par :

$$\mathrm{M}^k : G \rightarrow ((i, i^0) \rightarrow M_G^k[i][i^0]).$$

où M_G est la matrice d'adjacence de G.

Désormais, nous ne considérerons que ces fonctions de score (appelées fonctions de *longueur*). Dans ce contexte, M désignera l'ensemble de toutes les fonctions de longueur, i.e. $\mathrm{M} = \{\mathrm{M}^k \mid k \in \mathrm{N}\}$.

Ensuite, les trois fonctions de voisinage Id, adj et all définies dans l'exemple 3.18 seront utilisées. Remarquons que :

— Id combinée à M peut être utilisée pour filtrer les domaines en considérant le nombre de chemins menant d'un sommet à lui-même.
— adj est la fonction de voisinage utilisée par LAD.
— all est complémentaire à Id.

Remarque 1. LAD est un cas particulier de SND. En effet, LAD est équivalent à $\mathrm{SND}_{\mathrm{S,V}}$, avec $\mathrm{S} = \emptyset$ et $\mathrm{V} = \{\mathrm{adj}\}$.

Proposition 9. Chaque contrainte $\mathrm{SND}_{\mathrm{S,V}}(i_s, i_c)$, avec $\mathrm{S} \subseteq \mathrm{M}$ et $\mathrm{V} \subseteq \{\mathrm{Id}, \mathrm{adj}, \mathrm{all}\}$ est impliquée.

Preuve. Pour tout réseau de contrainte R et pour toute contraintes c sur X, nous avons clairement

$$sol(\mathrm{R} \oplus c) \subseteq sol(\mathrm{R}).$$

Ainsi,

$$sol(\mathrm{R} \oplus \mathrm{SND}_{\mathrm{S,V}}(i_s, i_c)) \subseteq sol(\mathrm{R})$$

Soit $x = (j_1, \ldots, j_n)$ une solution de R et $(i_s, i_c) \in \mathrm{S}_s \times \mathrm{S}_c$. L'ordre des sommets de G_s sera $i_1, \ldots, i_n$. Comme x est une solution de R alors $AllDiff(\mathrm{X})(x) = 1$.

Soit $V \in \mathrm{V}$. Comme $\{X_{i^0_s} \mid i^0_s \in V(i_s)\} \subseteq \mathrm{X}$ alors $AllDiff(\{X_{i^0_s} \mid i^0_s \in V(i_s)\})(x) = 1$. Comme x est une solution de R alors le morphisme f du graphe G_s vers G_c qui à tout i_k associe j_k est un morphisme injectif de graphe.

Ainsi, $f(G_s)$ est en bijection avec G_c. Clairement, si $i^0_s \in V(i_s)$ alors $f(i^0_s) \in V(f(i_s))$. De plus, $\mathrm{M}^k(G_s)(i_s, i^0_s) = \mathrm{M}^k(f(G_s))(f(i_s), f(i^0_s))$. Comme $f(G_s)$ est un sous-graphe de G_c alors

$$\mathrm{M}^k(G_s)(i_s, i^0_s) \leq \mathrm{M}^k(G_c)(i_c), i_c).$$

Pour nos expérimentations décrites plus loin, nous utilisons les fonctions de voisinage définies ci-dessus. Notons toutefois qu'il existe d'autres possibilités. On peut, par exemple, raisonner sur des voisinages définis comme étant les ensembles de sommets à distance 1 ou 2, à distance 1 et 2, et ainsi de suite. De plus, à la place des fonctions de longueur nous pouvons exploiter d'autres fonctions de score, basées par exemple sur le nombre de cliques ou de cycles. L'étude de ces alternatives est une perspective de notre travail à moyen terme.

3.5.2 Filtrage des contraintes SND

Le filtrage des contraintes SND dans le but d'atteindre GAC est similaire à ce qui a été proposé pour les contraintes *AllDiff* [92] et LAD. Ceci est basé sur le concept de couplage couvrant dans un graphe biparti. L'objectif est de filtrer les contraintes de type $\mathrm{SND}_{\mathrm{S,V}}(i_s, i_c)$ pour le réseau de contraintes avec les domaines courants. Pour cela un graphe biparti, noté $B_V(i_s, i_c)$, est crée pour chaque $V \in \mathrm{V}$. Il est défini de la manière suivante :

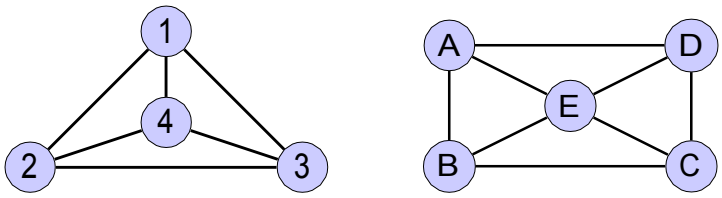
— L'ensemble des sommets de $B_V(i_s, i_c)$ est $V(i_s)$ t $V(i_c)$,

— L'ensemble des arêtes de $B_V(i_s, i_c)$ est :

$$\{(i_s^0, i_c^0) \in V(i_s) \times V(i_c) \mid i_c^0 \in dom(x_{i_s}) \wedge T(i_s, i^0) \leq T(i_c, i^0), \forall T \in S\}$$

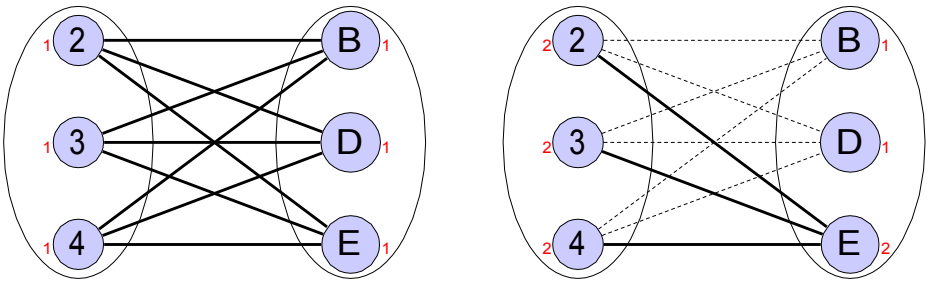
.

S'il n'existe pas de couplage du graphe biparti qui couvre $V(i_s)$, alors la partie droite de l'équation 3.1 (après le symbole $\Rightarrow$) est fausse et ainsi (X_{i_s}, i_c) n'a pas de support dans la contrainte. Donc i_c peut être supprimé de $dom(X_{i_s})$.



(a) Graphe motif G_s (b) Graphe cible G_c

Figure 3.2 : *Une instance du problème d'isomorphisme de sous-graphe.*



(a) $B_{adj}(1, A)$ pour S = {M¹} (b) $B_{adj}(1, A)$ pour S = {M²}

Figure 3.3 : *L'exemple (a) montre l'existence d'un couplage dans le graphe biparti lié à* $\text{SND}_{\{M^1\},\{adj\}}(1, A)$ *et l'exemple (b) montre qu'il n'y a pas de couplage faisable pour l'unique graphe biparti lié* $\text{SND}_{\{M^2\},\{adj\}}(1, A)$.

Exemple 3.20. La figure 3.2 montre une instance du problème d'isomorphisme de sous-graphe. En notant M_{G_s} et M_{G_c} les matrices d'adjacence respectivement de G_s et G_c, nous avons :

$$M_{G_s} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad M_{G_c} = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$$M^2_{G_s} = \begin{pmatrix} 3 & 2 & 2 & 2 \\ 2 & 3 & 2 & 2 \\ 2 & 2 & 3 & 2 \\ 2 & 2 & 2 & 3 \end{pmatrix} \quad M^2_{G_c} = \begin{pmatrix} 3 & 1 & 3 & 1 & 2 \\ 1 & 3 & 1 & 3 & 2 \\ 3 & 1 & 3 & 1 & 2 \\ 1 & 3 & 1 & 3 & 2 \\ 2 & 2 & 2 & 2 & 4 \end{pmatrix}$$

Considérons les contraintes $\text{SND}_{\{M^1\},\{adj\}}(1, A)$ et $\text{SND}_{\{M^2\},\{adj\}}(1, A)$. Pour simplifier, nous considérons ici une unique fonction de longueur et une unique fonction de voisinage.

Les graphes bipartis associés à ces contraintes sont schématisés dans la figure 3.3.

Nous avons adj(1) = {2, 3, 4} et adj(A) = {B, D, E} (en supposant que les domaines courant de X_2, X_3 et X_4 contiennent les valeurs initiales). Notons que les fonctions de score sont affichées (en rouge) à coté des sommets.

Il est clair qu'avec M^1, nous ne pouvons rien déduire sur (1, A) puisqu'il existe un couplage couvrant de $B_{\text{adj}}(1, A)$ en ne considérant que le score M^1.

Néanmoins, avec M^2, nous pouvons déduire que $X_1 = A$. En effet, avec le score basé sur M^2, toutes les arêtes en pointillés peuvent être supprimées (figure 3.3(b)). Par exemple, le score de (1, 2) est $M^2_{G_s}[1][2] = 2$ alors que le score de (A, B) est seulement $M^2_{G_c}[A][B] = 1$. Cela veut dire que 2 ne peut pas être relié à B et que cette arête peut être supprimée.

Finalement, en considérant les arêtes restantes (celles qui ne sont pas en pointillé), il est clair qu'il n'y a pas de couplage couvrant de $B_{\text{adj}}(1, A)$ en considérant le score M^2.

3.5.3 Simplification du graphe cible

Lorsque les domaines sont réduits durant le processus de propagation, il est quelquefois possible de raffiner les fonctions de score que nous utilisons. En effet, une arête peut être supprimée du graphe cible quand on a la garantie qu'aucun couple de sommets du graphe motif ne peut être couplé à cette arête. La définition suivante sera toujours donné dans le cadre et avec les notations précisées en début de chapitre.

Définition 3.20. Une arête $(i_c, i'_c) \in A_c$ sera dite **inaccessible** si

$$\forall (i_s, i'_s) \in A_s,\ (i_c, i'_c) \notin dom(X_{i_s}) \times dom(X_{i'_s}) \cup dom(X_{i'_s}) \times dom(X_{i_s}).$$

Ainsi, après le processus de filtrage, il est possible de simplifier le graphe cible en supprimant les arêtes inaccessibles. Les fonctions de score peuvent alors être mises à jour, comme par exemple les fonctions de longueur dans M puisque les matrices d'adjacence sont modifiées dès lors qu'une arête est supprimée. En conséquence, il est possible de recommencer le raisonnement de dominance pour les contraintes SND.

3.6 Étude qualitative

À partir de maintenant, SND_S sera utilisé comme raccourci de $\text{SND}_{S,\{\text{Id,adj,all}\}}$, et correspondra au filtrage établi par GAC sur le réseau initial augmenté de toutes les contraintes $\text{SND}_{S,\{\text{Id,adj,all}\}}(i_s, i_c)$. Dans notre étude, S pourra être M (l'ensemble des fonctions de longueur) ou une seule fonction M^k de M.

Nos premiers résultats montrent qu'il peut être utile de raisonner à la fois avec M^k et M^{k+1}.

Proposition 10. Il existe des entiers $k \geq 1$ tels que $\text{SND}_{\{M^k\}}$ n'est pas plus fort que $\text{SND}_{\{M^{k+1}\}}$.

Preuve. L'exemple 3.20 prouve que $\text{SND}_{\{M^1\}}$ ne peut pas être plus fort que $\text{SND}_{\{M^2\}}$, puisque $\text{SND}_{\{M^2\}}$ est capable d'inférer $X_1 = A$, contrairement à $\text{SND}_{\{M^1\}}$.

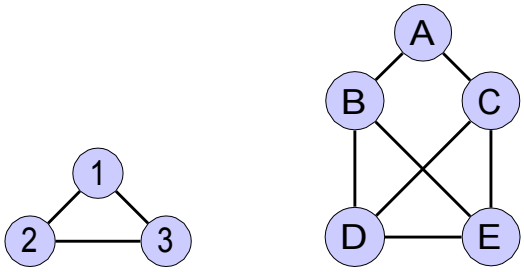
Proposition 11. Il existe des entiers $k \geq 1$ tels que $\text{SND}_{\{M^{k+1}\}}$ n'est pas plus fort que $\text{SND}_{\{M^k\}}$.

Preuve. Considérons le problème défini dans la figure 3.4. Les matrices M_G , M_{G_c} , M_{G_s} and M_{G_c} sont les suivantes :

$$M^5_{G_s} = \begin{bmatrix} 10 & 11 & 11 \\ 11 & 10 & 11 \\ 11 & 11 & 10 \end{bmatrix} \qquad M^5_{G_c} = \begin{bmatrix} 8 & 40 & 40 & 22 & 22 \\ 40 & 26 & 26 & 51 & 51 \\ 40 & 26 & 26 & 51 & 51 \\ 22 & 51 & 51 & 40 & 41 \\ 22 & 51 & 51 & 41 & 40 \end{bmatrix}$$

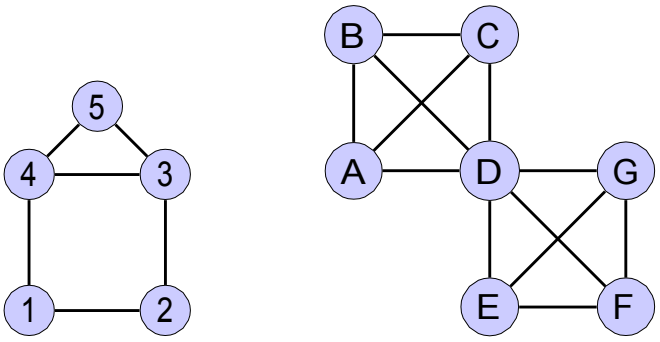
$$M^6_{G_s} = \begin{bmatrix} 22 & 21 & 21 \\ 21 & 22 & 21 \\ 21 & 21 & 22 \end{bmatrix} \qquad M^6_{G_c} = \begin{bmatrix} 80 & 52 & 52 & 102 & 102 \\ 52 & 142 & 142 & 103 & 103 \\ 52 & 142 & 142 & 103 & 103 \\ 102 & 103 & 103 & 143 & 142 \\ 102 & 103 & 103 & 142 & 143 \end{bmatrix}$$

Sur cet exemple, d'un côté, $SND_{\{M^6\}}$ ne permet d'effectuer aucune inférence (observez que toutes les valeurs dans $M^6_{G_c}$ sont systématiquement plus grandes que celles de $M^6_{G_s}$, ce qui garantit un couplage couvrant pour chaque fonction de voisinage). D'autre part, comme $M^5_{G_s}[1][1] = 10 > M^5_{G_c}[A][A] = 8$, avec Id, alors le sommet A peut être supprimé de $dom(X_1)$.



(a) G_p (b) G_t

Figure 3.4 : *Un exemple qui prouve que $SND_{\{M^6\}}$ n'est pas plus fort que $SND_{\{M^5\}}$.*



(a) G_s (b) G_c

Figure 3.5 : *Un exemple qui montre que SND_M n'est pas plus fort que SAC*

Proposition 12. SND_M est incomparable à SAC.

Preuve. Considérons tout d'abord l'exemple 3.20. Il montre que $\mathrm{SND}_{\mathrm{M}}$ peut filtrer plus de valeurs que SAC. En effet, $\mathrm{SND}_{\mathrm{M}}$ est capable d'inférer $X_1 = A$ alors qu'aucun test singleton réalisé par SAC ne permet de détecter une valeur incohérente sur cette instance.

Par exemple, établir GAC après l'affectation $X_1 = A$ réduit seulement les domaines de X_2, X_3 et X_4 à $\{B, E, D\}$, la valeur A est supprimée du fait de la contrainte $AllDiff$ et la valeur C est supprimée par les contraintes binaires. Comme aucun domaine ne devient vide, on ne peut pas en déduire que $X_1 = A$.

D'autre part, la figure 3.5 montre une instance où SAC filtre plus de valeurs que SND_M. En effet, considérons le test singleton pour (X_3, A). Avec la contrainte $AllDiff$, A est tout d'abord supprimé du domaine des autres variables. Avec les contraintes binaires impliquant X_3, les valeurs E, F et G sont supprimées des domaines de X_2, X_4 et X_5, ce qui donne à cet instant $dom(X_2) = dom(X_4) = dom(X_5) = \{B, C, D\}$. Nous avons ici 3 valeurs pour 3 variables, la contrainte $AllDiff$ va donc supprimer B, C et D du domaine de X_1, donnant $dom(X_1) = \{E, F, G\}$. Enfin, la contrainte binaire entre X_1 et X_2 supprime B et C de $dom(X_2)$. A ce stade, une incohérence est détectée et, donc, SAC infère que $X_3 = A$. Considérons maintenant M_{G_s}, M_{G_c}, $M^2_{G_s}$ et $M^2_{G_c}$. Nous avons :

$$M_{G_s} = \begin{pmatrix} 0&1&0&1&0\\ 1&0&1&0&0\\ 0&1&0&1&1\\ 1&0&1&0&1\\ 0&0&1&1&0 \end{pmatrix} \quad M_{G_c} = \begin{pmatrix} 0&1&1&1&0&0&0\\ 1&0&1&1&0&0&0\\ 1&1&0&1&0&0&0\\ 1&1&1&0&1&1&1\\ 0&0&0&1&0&1&1\\ 0&0&0&1&1&0&1\\ 0&0&0&1&1&1&0 \end{pmatrix}$$

$$M^2_{G_s} = \begin{pmatrix} 2&0&2&0&1\\ 0&2&0&2&1\\ 2&0&3&1&1\\ 0&2&1&3&1\\ 1&1&1&1&2 \end{pmatrix} \quad M^2_{G_c} = \begin{pmatrix} 3&2&2&2&1&1&1\\ 2&3&2&2&1&1&1\\ 2&2&3&2&1&1&1\\ ?&2&2&6&2&2&2\\ 1&1&1&2&3&2&2\\ 1&1&1&2&2&3&2\\ 1&1&1&2&2&2&3 \end{pmatrix}$$

Le raisonnement utilisant $SND_{\{M^1,M^2\}}$ ne permet pas d'identifier de valeurs incohérentes. Il y a toujours la possibilité de trouver un couplage couvrant sur chaque graphe biparti. Cela est également vrai pour M^β et à cause de la structure du graphe, la situation donne des scores non exploitables pour tout $k > 4$: la plus grande valeur de $M^4_{G_s}$ est plus petite que la plus petite valeur de $M^4_{G_c}$ (et cela sera toujours vrai pour $k \geq 5$). En d'autres termes, SND_M ne peut identifier aucune valeur incohérente.

Proposition 13. SND_M est strictement plus fort que LAD.

Preuve. LAD utilise seulement adj comme fonction de voisinage et aucune fonction de score. De plus, sur l'exemple 3.20, LAD se comporte comme $SND_{\{M^1\}}$, qui est strictement plus faible que SND_M.

Un résumé des relations entre les différents processus de filtrage existant dans la littérature pour le problème d'isomorphisme de sous-graphe et SND_M est donné par la figure 3.6.

3.7 Un algorithme SND incomplet

Cette section décrit un algorithme nommé SND^w qui met en application notre filtrage SND. En fait, cet algorithme réalise un niveau de filtrage plus faible que $SND_{M,\{Id,adj,all\}}$ complet. De plus amples détails sur ce point sont donnés à la fin de cette section.

Les étapes principales de SND^w sont décrites dans l'algorithme 8 ; les entrées sont le graphe motif G_s et le graphe cible G_c. Afin de réduire la quantité de mémoire nécessaire, cet algorithme évite de stocker de trop nombreuses matrices et exploite dans ce but les

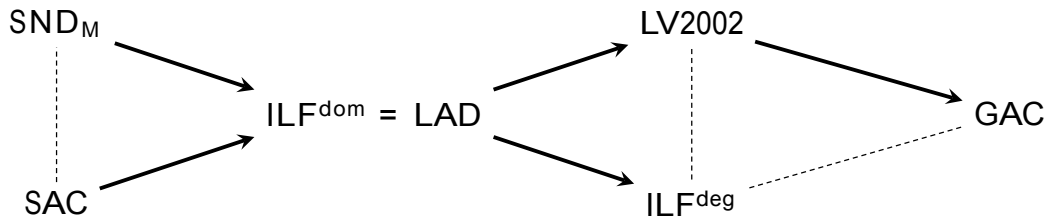


Figure 3.6 : *Résumé des relations entre les processus de filtrage. Une flèche de la forme $\varphi \rightarrow \psi$ signifie que φ est strictement plus fort que ψ. Une ligne en pointillé indique que les deux entités ne sont pas comparables.*

Algorithme 8 : SND^w(G_s : Source, G_c : Cible)

```
1  répéter
2      fini ← false ;
3      M^0_Gs ← I_np ; M^0_Gc ← I_nt
4      k ← 1 ;
5      répéter
6          modifie ← filtreUtilisantScore(k) ;
7          k ← k + 1 ;
8      jusqu'à k > MIN_ITERS or ¬modifie;
9      si ¬supprimeAreteCible() alors
10         fini ← true ;
11 jusqu'à fini;
```

Fonction filtreUtilisantScore(k : integer) : Boolean

```
1  M^k_Gs ← M^(k-1)_Gs × M_Gs ;
2  M^k_Gc ← M^(k-1)_Gc × M_Gc ;
3  modifie ← false ;
4  pour chaque variable X_is ∈vars(R) faire
5      pour chaque valeur i_c ∈dom(X_is) faire
6          si ¬estCoherent((i_s, i_c), ld)
7            ∨ ¬estCoherent((i_s, i_c), adj)
8            ∨ ¬estCoherent((i_s, i_c), all) alors
9              remove i_c from dom(X_is) ;
10             si dom(X_is) = ∅ alors
11               throw exception INCONSISTENT ;
12             modifie ← true ;
13 retourner modifie ;
```

scores M^k de manière indépendante. Le raisonnement commence donc avec M^1, puis avec M^2 et ainsi de suite, par le biais d'appels à la fonction filtreUtilisantScore (voir l'appel à la ligne 5, Alg. 8). La constante MIN_ITERS est fixée à une valeur faible (par exemple 4) afin d'exploiter au minimum les fonctions de score M^i avec $1 \leq i \leq$ MIN_ITERS. La boucle interne **tant que** (lignes 4-7) est stoppée quand le filtrage opéré avec M^k ne permet plus d'inférence (la fonction filtreUtilisantScore retourne alors `false`) et, comme indiqué ci-dessus, lorsqu'on a itéré suffisamment de fois. Après celà, nous essayons de supprimer des arêtes inaccessibles du graphe cible, comme indiquée dans la section 3.5.3, en utilisant une fonction nommée supprimeAreteCible (un parcours en largeur est lancé sur le graphe

Fonction estCoherent((i_s,i_c) : sommets, V : Voisinage) : Boolean

```
1 E ← ∅ ;
2 pour chaque i_s^0 ∈V(i_s) faire
3     pour chaque i_c^0 ∈V(i_c) ∩ dom(X_{i_s^0}) faire
4         si M^k_{G_s}[i_s][i_s^0] ≤ M^k_{G_c}[i_c][i_c^0] alors
5             E ← E ∪ {(i_s^0, i_c^0)} ;
6 retourner trouveCouple((V(i_s) t V(i_c), E)) ;
```

Séries			GAC		LAD		SAC		SND^w	
Name	#	|dom|	cpu	del	cpu	del	cpu	del	cpu	del
lv	793	5,877	0.5	188	0.01	103+754	24	1,237	0.6	**1,502**
si2	390	78K	0.6	3	0.29	25K99+24K41	168	**52K58**	1.1	51K48
si4	390	156K	0.75	8	1.05	54K34+52K98	177	**109K2**	1.4	107K5
si6	390	235K	0.8	12	2.19	76K12+84K09	193	165K3	2.1	**184K4**
sf	100	284K	1	0	0.32	141K3+135K7	567	234K3	2.5	**244K2**

Table 3.1: Pré-traitement. *Pour chaque technique de filtrage, cpu indique le temps moyen (en secondes) et del indique le nombre de valeurs supprimées.*

cible et supprime les arêtes inaccessibles au fur et à mesure). Si cette opération est effective, le processus entièrement recommencé en modifiant la matrice d'adjacence du graphe cible (retour à la ligne 1).

La fonction filtreUtilisantScore commence par calculer $M^k_{G_s}$ et $M^k_{G_c}$ à partir de $M^{k-1}_{G_s}$ et $M^{k-1}_{G_c}$. Nous devons donc stocker uniquement les matrices de deux niveaux différents ce qui nous permet de contrôler l'espace mémoire nécessaire par notre approche. Ensuite, pour chaque couple (i_s, i_c) tel que $i_c \in dom(X_{i_s})$, les conditions des lignes 6–8 correspondent à l'évaluation de la partie droite de l'équation 3.1, avec M comme unique fonction de longueur et {Id, adj, all} comme fonctions de voisinage. Quand cette condition est fausse (évaluée au moyen de la fonction estCoherent), la valeur i_c est supprimée de $dom(x_{i_s})$. Si c'était la dernière valeur possible du domaine, une exception est générée, indiquant une incohérence globale. Notons que la fonction filtreUtilisantScore retourne true quand au moins une valeur de domaine est supprimée. Finalement, la fonction estCoherent construit le graphe biparti associé au couple (i_s, i_c) et à la fonction de score M^k, et appelle ensuite un algorithme classique de couplage couvrant (fonction trouveCouple).

Notre algorithme SND^w est clairement plus faible que SND$_{M,\{Id,adj,all\}}$ pour plusieurs raisons. Tout d'abord, comme indiqué précédemment, nous exploitons les différents scores de manière indépendante (par simplicité et pour contrôler l'espace mémoire nécessaire). Ensuite, dans nos expérimentations nous avons programmé une version incomplète du couplage couvrant d'un graphe biparti. Enfin, les domaines pouvant être modifiés de façon permanente, nous pouvions envisager de considérer à nouveau les fonctions de score utilisées précédemment (nous pouvions donc ajouter une boucle à l'algorithme 8). Nous pensons toutefois que le sur-coût aurait été trop important.

3.8 Résultats expérimentaux

Afin de montrer l'intérêt pratique de notre approche, nous avons mené diverses expérimentations en utilisant un cluster de machines Xeon 3.0GHz avec 13Go de RAM. Nous avons utilisé

les instances d'isomorphisme de sous-graphe provenant de [108] et classées de la manière

Séries		MAC		LAD		pSAC		pSND^w		MSND^w	
Name	#	cpu	sol	cpu	sol	cpu	sol	cpu	sol	cpu	sol
lv	793	24	683	8.3	726	89	695	7.6	724	9.6	**729**
si2	390	38	352	13	345	153	236	25	356	28	**357**
si4	390	42	357	19	358	142	212	24	359	22	**366**
si6	390	46	346	20	372	191	204	22	367	24	**375**
sf	100	120	58	2.6	**100**	557	62	4.3	99	18	99

Table 3.2: Chercher une solution. *Pour chaque technique de filtrage, cpu indique le temps moyen (en secondes) pour résoudre une instance et sol représente le nombre d'instances résolues (timeout à* 1200 *secondes).*

suivante :

— La série lv est composée de 793 instances, provenant de la base Stanford [63, 66].

— Les séries si2, si4, et si6 sont composées de 390 instances chacune et proviennent de la base VFlib [34, 104].

— La série sf est composée de 100 instances de réseaux *scale free* générées de manière aléatoire en utilisant la distribution d'une loi de puissance [121].

Nous comparons tout d'abord les temps cpu (exprimés en secondes) et le niveau de filtrage (donné par le nombre de valeurs supprimées (del)) de GAC, SAC[1], LAD et SND^w. Nous avons commencé à appliquer ces algorithmes isolément, en les considérant donc comme faisant partie d'une étape de pré-traitement. Pour GAC, SAC et SND^w, nous avons utilisé notre plateforme AbsCon (écrit en java), alors que pour LAD nous avons utilisé le programme C++ disponible sur la page de son auteur http://liris.cnrs.fr/christine.solnon/. Le tableau 3.1 montre les résultats en moyenne obtenus sur les cinq séries d'instances. Pour chaque série, # représente le nombre d'instance et D le nombre moyen de valeurs (D est la moyenne de $D_R = \sum_{v \in vars(R)} |dom(v)|$ pour tous les réseaux de contraintes R de la série considérée). Notons que la valeur del est donnée sous la forme $n_1 + n_2$ pour LAD car ce dernier exploite la structure des graphes pour initialiser les domaines (n_1 représente le nombre de valeurs supprimées à cette étape). Nous pouvons tout d'abord observer que SAC et SND^w sont très proches en terme de valeurs supprimées : sur certaines séries SAC est meilleur et sur d'autres, c'est SND^w qui l'est. Par contre au niveau du temps cpu, SAC apparaît clairement plus lent que SND^w (de plusieurs ordres de grandeur). Afin d'avoir une comparaison juste entre SAC et SND^w, nous n'avons gardé que les instances pour lesquelles SAC finit avant les 1200 secondes de timeout. Cela peut expliquer pourquoi LAD semble filtrer plus sur la série sf. En résumé, SND^w est significativement plus puissant que LAD (et proche de SAC), le sur-coût en temps est relativement faible (en prenant en compte aussi que les langages de programmation utilisés sont différents).

Pour notre deuxième expérimentation, nous avons cherché une solution par instance avec un timeout de 1200 secondes. Nous avons utilisé l'algorithme MAC qui maintient GAC durant le processus de recherche, LAD et aussi MSND^w, l'algorithme qui maintient SND^w durant la recherche (nous ne faisons qu'une boucle principale de l'algorithme 8, en supprimant l'appel à removeTargetEdges durant la recherche). Nous avons aussi utilisé MAC après pré-traitement de type SAC (pSAC) ou de type SND^w (pSND^w). Le tableau 3.2 montre les résultats obtenus. En terme du nombre d'instances résolues (sol), MSND^w est clairement la meilleure approche, sauf pour la série sf ou LAD est extrêmement efficace.

1. Nous avons testé SAC1 et SAC3 [12]. Les résultats sont assez proches ici (en moyenne). Seuls les résultats de SAC1 sont donc montrés.

3.9 Graphes additionnels

Cette partie abordera un travail de C. McCreesh et P. Prosser [79]. Il s'agit d'un algorithme qui est meilleur que l'implémentation donnée de SND sur certains aspects. Leur article a été publié après le notre et fait référence à SND. C'est pourquoi cette partie n'a pas été placée dans la section 3.4.

Cet algorithme fonctionne en plusieurs étapes. Tout d'abord un voisinage et un score particuliers sont utilisés pour faire du filtrage. Cet aspect est présenté sous une forme fonctorielle appelée **graphes additionnels**. Les auteurs remarquent que le calcul d'un couplage couvrant est efficace mais trop lourd en terme de temps. Une autre approche est alors envisagée avec une méthode appelée **counting-based all-different propagation**. Cette méthode filtre moins mais permet de gagner beaucoup de temps sur les problèmes présentés dans les expérimentations. Enfin les auteurs utilisent le parallélisme pour améliorer leurs performances.

Commençons par expliquer le filtrage effectué.

Pour un graphe $G = (S, A)$, nous noterons $G^{[n,l]}$ le graphe dont l'ensemble des sommets est S. De plus, deux sommets s et s^0 (éventuellement $s = s^0$) de S sont voisins dans $G^{[n,l]}$ s'il existe au moins n chemins de longueur l entre s et s^0 dans G.

Soit f un monomorphisme de G_s dans G_c. Pour tous les entiers $n, l \in \mathbb{N}^*$, un monomorphisme, noté $f^{[n,l]}$, est induit de $G_s^{[n,l]}$ dans $G_c^{[n,l]}$. L'idée est que si s et s^0 sont des sommets voisins dans $G_s^{[n,l]}$ alors $f(s)$ et $f(s^0)$ dont voisins dans $G_c^{[n,l]}$.

Une première fonction calcule des paires $(G_s^{[n,l]}, G_c^{[n,l]})$ pour

$$(n, l) \in \{(1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3, 3)\}.$$

Ce choix est arbitraire mais adapté aux problèmes traités. Les auteurs mentionnent que dans les problèmes testés, il n'est pas indispensable d'aller plus loin et que le coût pour $n > 3$ est important. Il serait sans doute intéressant d'estimer ce coût de manière théorique pour des problèmes quelconques. Il serait peut-être possible alors d'engendrer un nombre de paires qui serait fonction du problème.

Essentiellement, le filtrage utilisé est le suivant : lorsqu'une variable, par exemple X_{i_s}, est instanciée à $i_c \in S_c$, toutes les autres variables sont visitées. Pour chaque variable $X_{i_s^0} = X_{i_s}$, les éléments de $dom(X_{i_s^0})$ qui ne sont pas voisins de i_c dans $G_c^{[n,l]}$ sont supprimés. C'est-à-dire qu'une valeur $j_c^?$ est supprimée de $dom(X_{i_s^0})$ s'il y a au moins n chemins de longueur l entre i_s et i_s^0 et qu'il y a moins de n chemins de longueur l entre i_c et $i_c^?$. Comme nous allons le voir, ceci est légèrement différent du modèle SND implémenté mais reste un

cas particulier de SND.

Exemple 3.21. Prenons, par exemple, $n = 2$ et $l = 2$. Le filtrage précédent signifie que si entre i_s et i_s^0 il y a 100 chemins de longueurs 2 et qu'entre i_c et $i_c^?$ il y a 10 chemins de longueur 2, il n'y aura pas de filtrage alors que cette information devrait nous pousser à filtrer. Il faudra attendre au moins $n = 11$ et $l = 2$ pour supprimer i_c^0. Mais dans la pratique, la complexité du calcul ne nous permettra pas d'aller jusque là. En revanche, si entre i_c et i_c^0 il y avait un seul chemin de longueur 2 alors i_c serait supprimé de $dom(X_{i_s})$.

L'approche SND avec $S = \{M^k, k \in \mathbb{N}\}$ et $V = \{adj\}$ est très semblable. En revanche, la fonction de voisinage est adj alors que les auteurs considèrent tous les sommets. Il faut donc utiliser une fonction de voisinage all un peu modifiée.

Nous noterons all^+ la fonction de voisinage qui à un graphe $G = (S, A)$ associe l'application $all^+(G) : \begin{array}{ccc} S & \longrightarrow & \mathcal{P}(S) \\ i & \longrightarrow & S \end{array}$.

Considérons donc SND avec $S = \{M^k, k \in N\}$ et $V = \{all^+\}$. Nous nous rapprochons alors du modèle des graphes additionnels. Dans l'exemple 3.21, la fonction de score M^2 permettrait de filtrer i_c^{ζ}.

Ainsi, SND avec $S = \{M^k, k \in \{1, 2, 3\}\}$ et $V = \{allI^+\}$ filtre davantage que la méthode des graphes additionnels mais le calcul des puissances des matrices d'adjacences est sans doute plus coûteux.

Notons qu'au moment de l'initialisation les degrés des sommets voisins dans les graphes additionnels sont pris en comptes à la manière des multi-ensembles de C. Solnon. Cet aspect peut également être pris en compte avec le couplage couvrant et la fonction de voisinage adj est la fonction de score qui à un graphe G associe :

$$(s, s^0) \rightarrow deg_G(s^0).$$

Pour se rapprocher du modèle proposé, pour $n, l \in \mathbb{N}^*$, nous pouvons considérer la fonction de score

$$m^{n,l} : G \rightarrow ((s, s^0) \rightarrow m^{n,l}(G)(s, s^0)$$

où $m^{n,l}(G)(s, s^0) = 1$ s'il existe au moins n chemins de longueur l dans G entre s et s^0 et 0 sinon.

Ainsi SND avec $S = \{m^{n,l}, (n, l) \in \{(1, 1), (1, 2), (2, 2), (3, 2), (1, 3), (2, 3), (3, 3)\}\}$ et $V = \{allI^+\}$ se rapproche beaucoup du modèle proposé.

Dans l'article [79], les auteurs remarquent que le calcul du couplage couvrant est lourd. C'est pourquoi un algorithme nommé **counting-based all different propagation** est proposé. Il s'agit de passer chaque variable en revue dans l'ordre croissant des domaines et de supprimer du domaine d'une variable des sous-ensembles appelés **Hall set**. La logique est la suivante : s'il y a n possibilités pour n variables alors il est possible de supprimer ces valeurs des autres domaines (cf Algorithme 6 de [79]). Cette méthode permet de gagner beaucoup de temps même si le filtrage est moins fort.

Une implémentation de type parallélisme multi-cœur est proposée à partir de ces méthodes et permet de gagner encore du temps.

Ces méthodes sont intéressantes et efficaces en pratique. Nous avons vu que la partie théorique sur les graphes additionnels entre dans le cadre de SND, le propagateur utilisé et le parallélisme pourraient être utilisés sur SND avec d'autres fonctions de voisinage et de score. Par ailleurs, dans les tests effectués, le critère de temps a été retenu avec SND codé en java contre des solveurs codés en *C*. Il serait intéressant d'étudier théoriquement les différences et de comparer les méthodes sur un même langage. Ceci ouvre de nombreuses perspectives.

3.10 Conclusion

Nous avons proposé une approche générale pour résoudre le problème d'isomorphisme de sous-graphe. Cette approche, dite SND, est paramétrée par deux ensembles de fonctions de score et de voisinage. Nous avons montré l'intérêt théorique et pratique de SND sur une petite sélection de fonctions. Les perspectives immédiates de ce travail consistent d'une part à améliorer l'implantation courante et de l'autre, à mettre au point de nouvelles fonctions de score et de voisinage.

Chapitre 4

Contrôle statistique du processus de propagation de contraintes

Dans ce chapitre nous allons nous intéresser à la propagation de contraintes de l'algorithme MAC. Nous avons vu dans le chapitre 1 que cette propagation pouvait coûter très cher en temps et en espace. De plus, rien ne garantit que l'effort à faire pour obtenir le point fixe permette de supprimer beaucoup de valeurs dans les domaines. Dans un premier temps nous décomposerons le processus de propagation afin d'en explorer tous les aspects. Pour cela le filtrage ϕ_{AC} sera écrit comme une composition de filtrages élémentaires représentant chacun une étape dans la propagation. Ensuite, à partir d'une analyse statistique portant sur des propriétés précises du mécanisme de propagation, nous mettrons en lumière une corrélation entre la longueur de la queue de propagation et le nombre de valeur filtrées dans les domaines. Nous définirons une fonction de coût permettant d'évaluer l'effort à produire pour filtrer les valeurs restantes lors de la propagation. Nous montrerons alors qu'il est possible d'effectuer des prédictions fiables sur la capacité du processus de propagation à détecter l'incohérence. En utilisant cette observation dans le but de contrôler l'effort de propagation, nous montrons son intérêt pratique sur un jeu d'essai comportant de nombreuses séries d'instances CSP.

Enfin, nous montrerons également l'efficacité de cette technique sur les instances de coloration de graphes.

4.1 Passes arc cohérentes

Nous avons vu dans la partie 1 que le filtrage par arc cohérence maintenait l'arc cohérence du réseau après chaque assignation. Pour cela, des valeurs sont supprimées des domaines des variables jusqu'à l'obtention d'un point fixe arc cohérent. Le calcul de ce point fixe peut être long et coûteux.

Pour comprendre comment fonctionne la propagation, il nous faut décomposer MAC.

Définition 4.1. Soit R un réseau de contraintes. Nous appellerons R-**filtrage** une application qui à un sous-réseau réduit R^0 de R associe un sous-réseau réduit de R^0.

Nous allons maintenant définir un R-filtrage qui ne filtre que par cohérence d'arc sur une seule contrainte. Ce sera un élément de base avec lequel nous pourrons construire le filtrage ϕ_{AC}.

Définition 4.2. Soit R = (X, C) un réseau de contraintes et une contraintes $C \in$C. nous noterons ϕ^C_{AC} le R-filtrage qui à un sous-réseau $R_{|D}$ réduit de R associe le plus grand sous-réseau réduit $R_{|D'}$ de $R_{|D}$ pour l'ordre induit de D tel que $(D'_X)_{X\in X} = D' \subseteq D \subseteq$D et $C_{|\Pi_{X\in scp(C)} D'_X}$ est arc cohérente.

Cette propriété est stable par réunion et implique l'existence du majorant.

Remarque 4.1. Il ne s'agit pas d'un filtrage au sens général puisque l'application dépend de R. Les R-filtrages sont des filtrages pour un réseau R fixé.

Définition 4.3. Soit R = (X, C) un réseau de contraintes avec C = $\{C_1, \ldots, C_e\}$. Considérons la relation d'ordre strict $\prec$ tel que $C_1 \prec \ldots \prec C_e$. Nous noterons alors $\phi_{AC(1)}$ le R-filtrage qui à un sous-réseau R^0 de R associe $\phi_{AC} \circ \ldots \circ \phi_{AC}(R^0)$.

Ce R-filtrage sera appelé filtrage par **cohérence d'arc en une passe**.

Remarque 4.2. Comme le montre l'exemple suivant, le filtrage par cohérence d'arc en une passe dépend de l'ordre dans lequel sont prises les contraintes.

Exemple 4.1. Considérons le réseau de contraintes défini par :
- X = $\{X, Y\}$
- $dom(X) = dom(Y) = \{0, 1\}$
- C = $\{C1, C2\}$ avec $scp(C_1) = scp(C_2) = \{X, Y\}$.
- Les contraintes C_1 et C_2 sont données en extension par :
  - $C_1 = \{(1, 0)\}$
  - $C_2 = \{(0, 0), (0, 1), (1, 1)\}$.

Appliquons $\phi^{\prec}_{AC(1)}$ pour l'ordre $C_1 \prec C_2$:
- L'arc cohérence sur C_1 réduit $dom(X)$ à $\{1\}$ et $dom(Y)$ à $\{0\}$.
- L'arc cohérence sur C_2 réduit alors $dom(X)$ et $dom(Y)$ à l'ensemble vide.

Appliquons $\phi^{\prec}_{AC(1)}$ pour l'ordre $C_2 \prec C_1$:
- L'arc cohérence sur C_2 ne change pas les domaines car chaque valeur des domaines a un support.
- L'arc cohérence sur C_1 réduit $dom(X)$ à $\{1\}$ et $dom(Y)$ à $\{0\}$.

Définition 4.4. Nous appellerons **cohérence d'arc en *n* passes** et nous noterons $\phi^{\prec}_{AC(n)}$ la composition de *n* fois $\phi^{\prec}_{AC(1)}$.

La suite récurrente (R_n) définie par :

$$\forall n \geq 0,\ R_n = \phi^{\prec}_{AC(n)}(R)$$

est stationnaire. Pour voir cela, il existe de nombreuses méthodes. Nous pouvons, par exemple, considérer le cardinal des domaines des variables qui constituent des suites décroissantes et minorées. Cette limite ne dépend pas de l'ordre donné pour les contraintes et il s'agit de $\phi_{AC}(R)$. En revanche, comme nous l'avons vu, l'effet de $\phi^{\prec}_{AC(n)}$ est très dépendant de l'ordre. Ainsi, le temps mis pour obtenir $\phi_{AC}(R)$ peut varier de manière substantielle en fonction de l'ordre de filtrage des contraintes.

Bien que MAC soit considéré comme état de l'art, sur certains problèmes, FC peut parfois surclasser MAC. Par exemple, [28] montre que expérimentalement FC est meilleur que MAC sur des CSP difficiles dont le graphe primal est très dense et la dureté des contraintes est faible.

Plusieurs tentatives pour ajuster le bon niveau de filtrage ont été effectuées ces dernières années. Tout d'abord, dans [27], les auteurs proposent une vison paramétrée de la cohérence locale qui permet de nombreuses formes partielles de GAC d'être établies. Le paramètre utilisé pour cette approche est la distance entre la dernière variable assignée et les autres variables dans le graphe de contraintes. Ensuite, Mehta et van Dongen ont introduit une approche probabiliste [81] de manière à n'effectuer uniquement que les tests de contraintes utiles en ne recherchant pas de support pour une valeur lorsqu'il y a une forte probabilité qu'un tel support existe. Cette technique d'inférence de support probabiliste a été étudiée à la fois pour GAC et SAC (singleton arc consistency). Plus récemment, Stergiou [109] a

pro-posé d’adapter dynamiquement le niveau de cohérence locale devant être appliqué durant

la recherche. Il a montré que l'information concernant les domaines devenus vides (domain wipe-outs) et les suppressions de valeurs pendant la recherche peuvent être utilisées non seulement pour sélectionner les variables mais également pour ajuster automatiquement le niveau de filtrage

Dans la suite de ce chapitre, nous observons cette vitesse de convergence et nous proposons une méthode pour arrêter le filtrage AC avant son terme, estimant que l'essentiel des valeurs des domaines ont été filtrées et que l'effort à produire pour aller filtrer les valeurs restantes est trop important.

4.2 Propagation orientée variable

Dans cette section, nous allons détailler le mécanisme de queue de propagation dans la cadre de l'algorithme MAC.

Dans le *schéma orienté variable* [80, 117, 22], lorsqu'une valeur du domaine d'une variable est supprimée, la variable est ajoutée à un ensemble Q appelé *queue de propagation* (voir chapitre 2). Le *schéma orienté variable* peut être optimisé par l'utilisation de tampons temporels (timestamps). Ceux-ci permettent de dater certains événements et de suivre au cours du temps la progression d'un algorithme. Ils permettent notamment dans notre cas de déterminer si une révision est utile ou pas.

En introduisant une horloge (compteur) globale *time* et en associant un tampon *stamp*[X] à chaque variable X et un tampon *stamp*[C] à chaque contrainte C, il est possible de déterminer les révisions qui seront potentiellement effectives. La valeur *stamp*[X] indique à quel moment a eu lieu la dernière suppression d'une valeur dans *dom*(X) tandis que la valeur *stamp*[C] indique le moment le plus récent où la contrainte C a été rendue GAC-cohérente. Les variables *time*, *stamp*[X] pour chaque variable X et *stamp*[C] pour chaque contrainte C sont initialisées à 0. La valeur de *time* est incrémentée à chaque fois qu'une variable est ajoutée à Q et à chaque fois qu'une contrainte est rendue GAC-cohérente.

L'appel GAC^{var}(*vars*(R)) (voir l'algorithme 9) assure la cohérence d'arc généralisée pour un réseau de contraintes R donné ; dans un premier temps nous passons sous silence les instructions en gris clair des lignes 4,7 et 17-18. La valeur booléenne *false* est retournée si R est prouvé AC-incohérente, c'est-à-dire si $\phi_{AC}(R) = \bot$. Soit *past*(R) l'ensemble des variables passées de R, c'est-à-dire les variables de R qui ont été instanciées par un algorithme de recherche avec retour arrière comme forward checking ou MAC. Lors de l'initialisation, nous avons *past*(R) = $\emptyset$ car il n'y a aucune variable assignée et GAC^{arc}(*vars*(R)) calcule simplement la fermeture arc cohérente de R.

Pour établir GAC, les variables sont sélectionnées de manière itérative dans Q (ligne 6). Chaque contrainte C qui porte sur la variable sélectionnée X est prise en compte si *stamp*[X] > *stamp*[C]. Dans ce cas, chaque variable non assignée $Y = X$ dans *scp*[C] est révisée par rapport à la contrainte C. Chaque révision est effectuée par la fonction revise, de l'algorithme 11, qui retourne *true* si une valeur, au moins, a été supprimée. Si la révision est effective, Y est ajoutée à Q. Pour garantir que C est encore arc cohérente, nous devons déterminer si l'un des domaines des autres variables que X dans C a été modifié depuis le dernier établissement de GAC sur C. Ceci est pris en charge par la seconde partie de la condition de la ligne 10 de l'algorithme 9. Si la seconde partie de la condition est vérifiée alors X est une variable qui doit être révisée. Si un domaine devient vide, l'algorithme 9 retourne *false* à la ligne 13. Dans l'algorithme 11, pour chaque valeur a dans *dom*(X) la fonction seekSupport indique si il existe un support pour (X, a) sur C. De nombreuses implémentations de seekSupport ont été proposées telles que celles utilisées avec (G)AC3 [76], GAC2001 [18] et GAC3^{rm}[72].

Algorithme 9 : GAC^{var}(X_{evt} : ensemble de variables)

Résultat : *true* ssi $\phi_{AC}(R) = \bot$

```
1  Q ← ∅ ;
2  pour chaque variable X ∈X_evt faire
3      insert(Q, x) ;
4  cnt ← 0 ;
5  tant que Q = ∅ faire
6      choisir et supprimer X de Q ;
7      cnt ← cnt + 1 ;
8      pour chaque C ∈cont(R) | x ∈scp(C) ∧ stamp[X] > stamp[C] faire 9
   pour chaque variable Y ∈scp(C) | Y ∉past(R) faire
10             si Y = X or ∃Z∈scp(C) | Z = X ∧ stamp[Z] > stamp[C] alors 11
   si revise(C, Y) alors
12                 si dom(Y) = ∅ alors
13                     retourner false
14                 insert(Q, Y) ;
15         time ← time + 1 ;
16         stamp[C] ← time ;
17     si cnt ≥ threshold alors
18         break ;
19 retourner true
```

Algorithme 10 : insert(Q : ensemble de variables, X : variable)

```
1  Q ← Q ∪ {X} ;
2  time ← time + 1 ;
3  stamp[X] ← time ;
```

Algorithme 11 : revise(C : contrainte, X : variable)

Résultat : *true* ssi la révision de (C, X) est effective

```
1  nbElements ← |dom(X)| ;
2  pour chaque value a ∈dom(X) faire
3      si ¬seekSupport(C, X, a) alors
4          supprimer a de dom(X) ;
5  retourner nbElements = |dom(X)|
```

4.3 Contrôle de la propagation

Dans ce chapitre, nous considérons l'algorithme MAC. Il est considéré comme l'un des plus efficaces pour la résolution générique d'instances CSP. MAC [99] parcourt l'espace de recherche en profondeur d'abord avec retour arrière et établit la propriété d'arc cohérence (généralisée) après chaque décision prise en cours de recherche. Nous avons vu dans le chapitre 1 qu'il construit un arbre de recherche binaire tel que, à chaque noeud interne v, un couple (X, a) est sélectionné, X étant une variable non assignée, et a une valeur appar-tenant à $dom(X)$. On considère alors deux cas : l'assignation $X = a$ (décision positive) et la réfutation $X = a$ (décision négative).

Une variable *passée* est (explicitement) assignée, tandis qu'une variable *future* n'est pas

(explicitement) assignée. Pour maintenir GAC au cours de la recherche, on fait appel à $GAC^{var}(\{x\})$ après chaque décision (positive ou négative) prise sur une variable X. Dans le cas d'une décision positive, nous savons que X appartient désormais aux variables passées. En conséquence, la queue Q ne contient initialement que la variable X.

Par la suite, nous nous référons également à *forward checking* (FC), algorithme de recherche vu également au chapitre 1. Nous avons vu, qu'au niveau des réseaux de contraintes binaires, lorsqu'une variable X est assignée, seules les variables futures (non assignées) directement connectées à X
sont révisées. Pour les réseaux non binaires, différentes généralisations sont possibles. Le filtrage via FC n'est pas exécuté en phase de pré-traitement (i.e. avant
la recherche, lorsqu'aucune variable n'est assignée), ni après une décision négative.

Nous nous intéressons dans ce chapitre à deux propriétés particulières de la propaga-tion :

- la *longueur* de la propagation, qui correspond au nombre de variables qui ont été extraites de la queue via un appel à la fonction GAC^{var},
- et le *résultat* de la propagation qui correspond à la valeur, *true* ou *false*, renvoyée par GAC^{var}.

Plus précisément, nous avons étudié l'existence d'une corrélation entre la longueur de la propagation, et son résultat. On observe alors que pour de nombreuses séries d'instances [1], la longueur moyenne de la propagation est significativement plus petite lorsque la fonction GAC^{var} renvoie *false* que lorsqu'elle renvoie *true*.

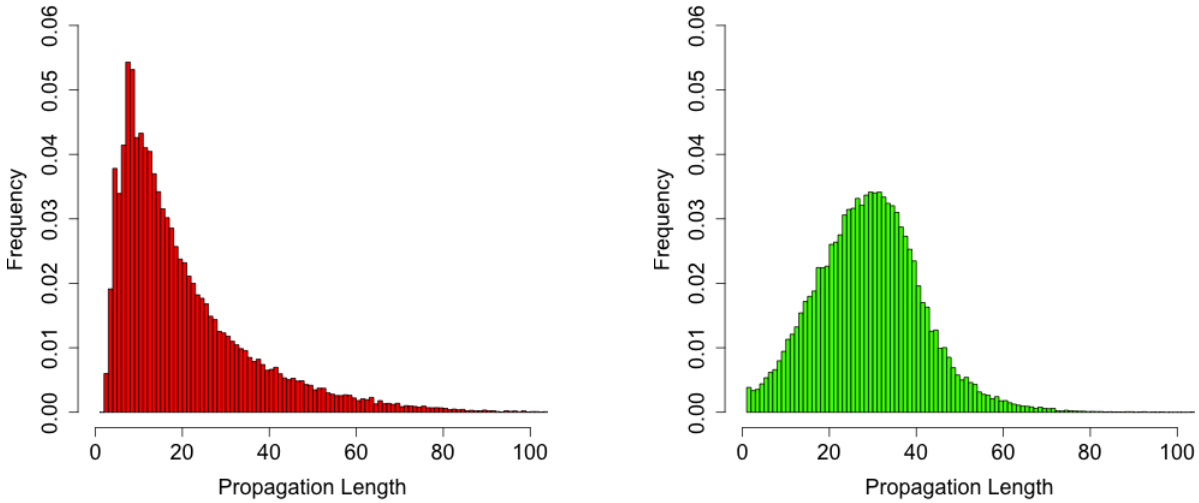
Par exemple, lors de la résolution de l'instance de mots croisés cw-words-vg5-9, MAC (avec l'heuristique *dom/wdeg*) appelle environ 91 000 fois GAC^{var}. 58 000 appels renvoient *false* et 33 000 renvoient *true*. La distribution précise (en pourcentages) est donnée par la figure 4.1(a). La longueur moyenne de le queue de propagation du résultat *false* (notée avg-false) est de 20.8, tandis qu'elle est de 29.7 pour les résultats *true* (avg-true). D'autres cas de figure présentent des écarts encore plus prononcés, comme le montrent très clairement les figures 4.1, 4.2 et 4.3. Les valeurs avg-false et avg-true suivantes sont obtenues :

- 4.4 et 24.7 pour l'instance graph-valiente -8-13
- 8.8 et 28.3 pour l'instance langford-3-12
- 9.8 et 12.8 pour l'instance qcp-15-120-5
- 4.2 et 12.6 pour l'instance rand-3-20-20-60-632-18
- 2.7 et 23.9 pour l'instance scens11-f4
- 2.0 et 45.3 pour l'instance lemma-50-9-mod

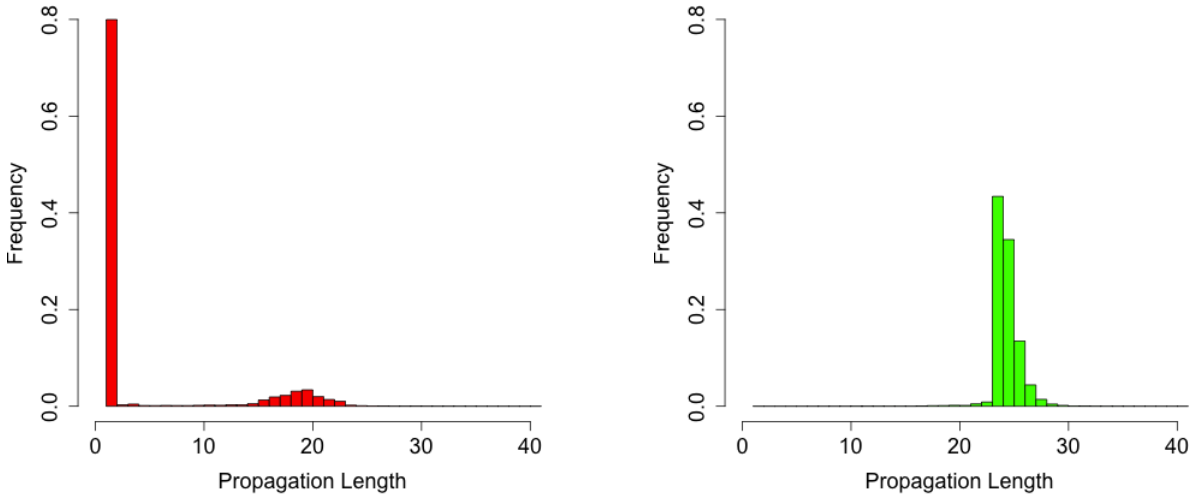
Précisons que chaque résultat présenté ci-dessus est représentatif du fonctionnement général de MAC sur l'ensemble de la série à laquelle appartient l'instance. La valeur de avg-false est en général bien plus petite que celle de avg-true, sauf dans quelques cas, assez rares, où l'inverse est constaté. Par exemple, au niveau de la série pseudo-ii, on obtient de manière assez surprenante 58.9 pour avg-false, et 1.1 pour avg-true sur l'instance pseudo-ii32c1 (voir figure 4.2(a)). En effet, sur ces instances montrer qu'une instanciation donne un réseau qui ne vérifie pas la cohérence d'arc demande beaucoup d'effort à GAC et donc une longueur de queue de propagation longue. En revanche, une instanciation permettant de produire un sous-réseau réduit arc cohérent filtre peu de valeurs et le point fixe est rapidement atteint. Il s'agit de réseaux qui sont toujours proches de la cohérence d'arc et sur lesquel GAC est peu efficace.

La mise en évidence d'une corrélation entre la longueur et le résultat de la propagation nous a amené à développer une nouvelle variante de MAC. Globalement, si nous pouvons déterminer une longueur de propagation à partir de laquelle la fonction GAC^{var} a de fortes probabilités de retourner *true*, pourquoi ne pas l'arrêter dès que cette longueur est atteinte ? Certaines valeurs qui auraient dû être supprimées seront conservées, mais en contrepartie, nous diminuons significativement l'effort nécessaire pour identifier les si-

1. Nos expérimentations ont porté sur un nombre conséquent de séries disponibles à http://www.cril. fr/~lecoutre/benchmarks.html.

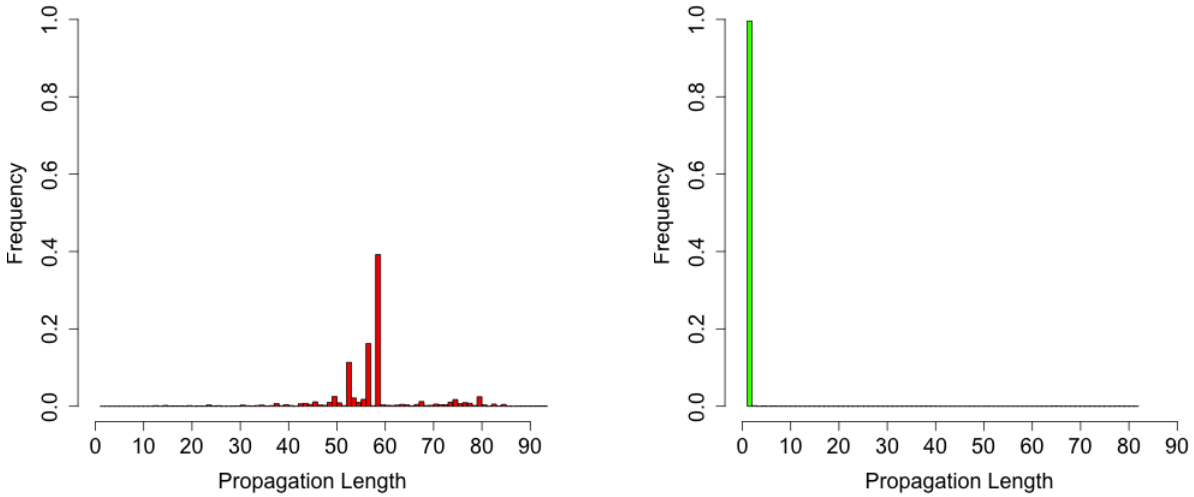


(a) crossword-m1c-words-vg5-9

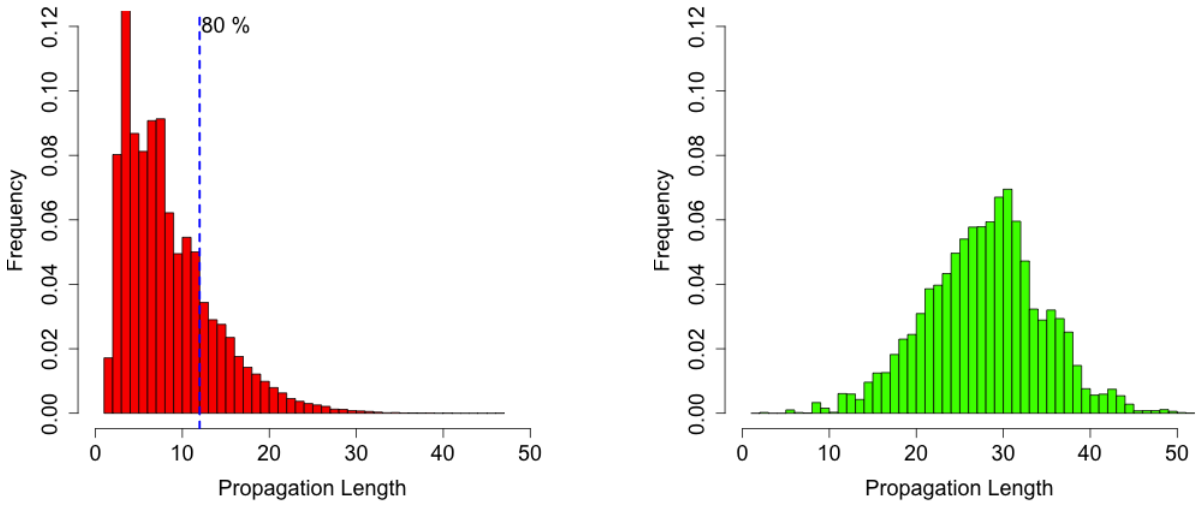


(b) graph-valiente

Figure 4.1 : *Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation* `false` *(à gauche) et* `true` *(à droite) pour différentes instances.*

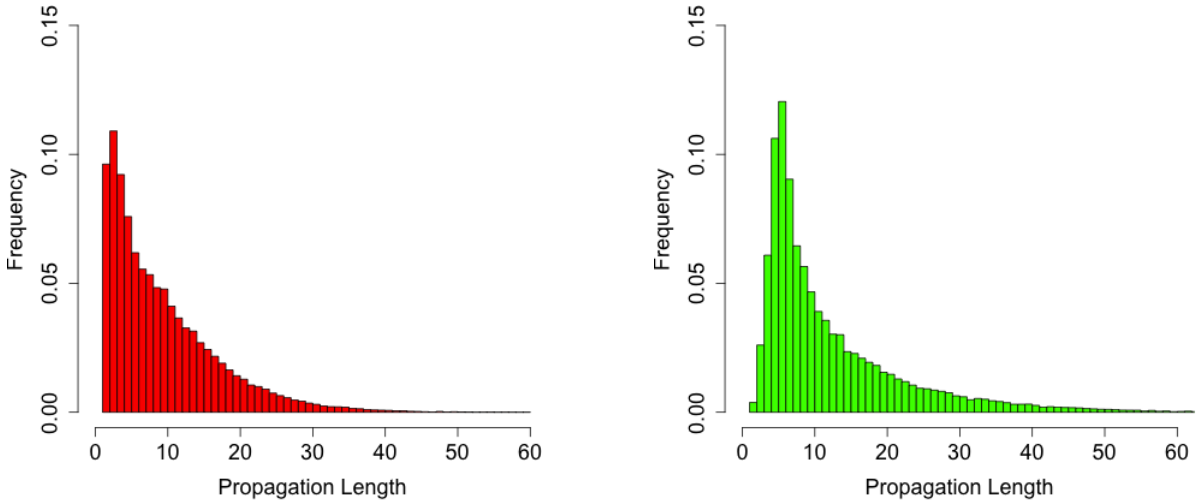


(a) pseudo-ii32c1

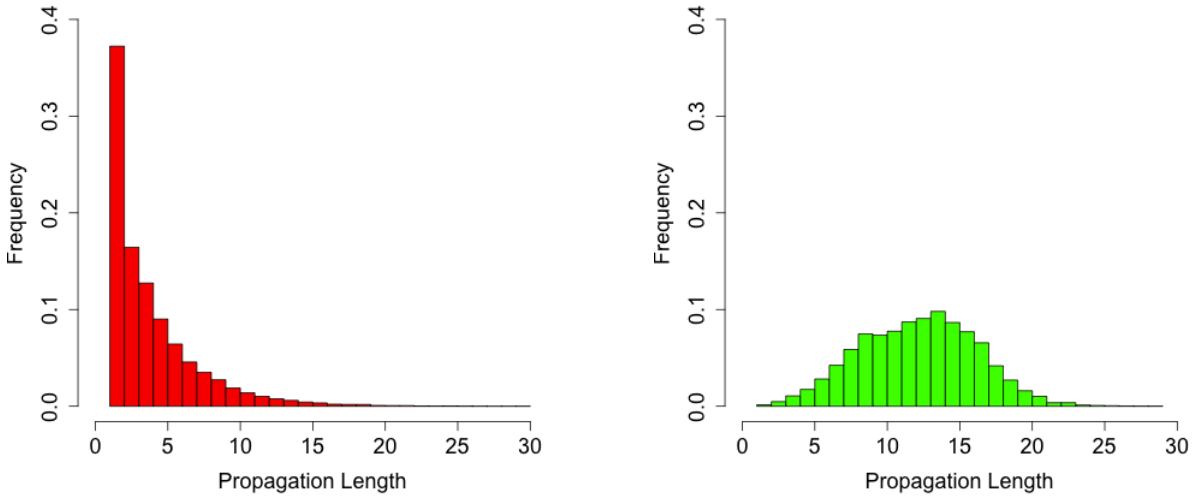


(b) langford-3-12

Figure 4.2 : *Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation false (à gauche) et true (à droite) pour différentes instances.*

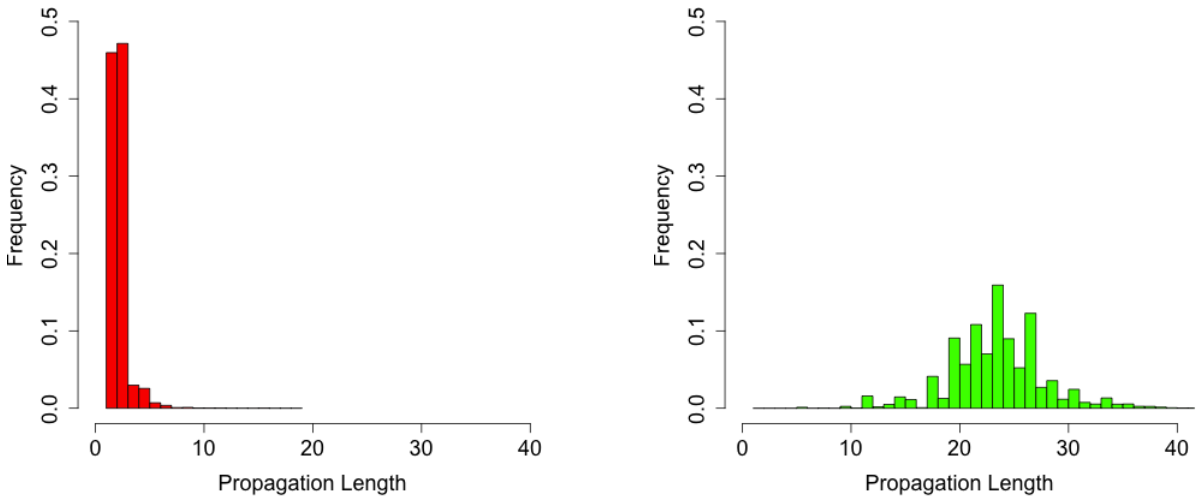


(a) qcp-15-120-5

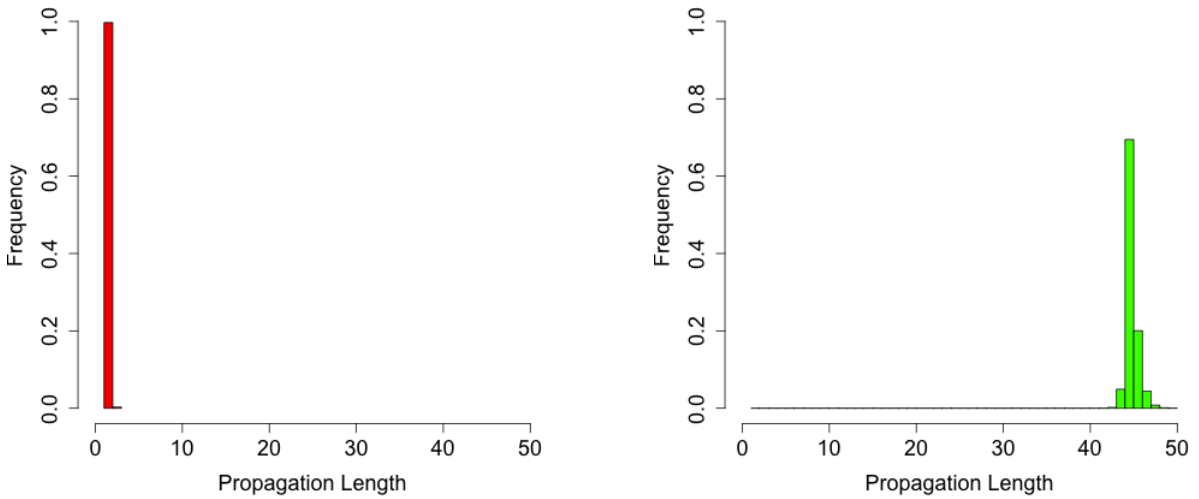


(b) rand-3-20-20-60-632-18

Figure 4.3 : *Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation false (à gauche) et true (à droite) pour différentes instances.*



(a) scens11-f4



(b) lemma-50-9-mod

Figure 4.4 : *Corrélation entre longueurs et résultats de propagation : distribution des résultats de propagation false (à gauche) et true (à droite) pour différentes instances.*

tuations d'échec engendrées par un domaine vide. Nous proposons pour cela MAC^c, qui correspond à l'algorithme 9 dont les lignes grises additionnelles, entraînent les modifications suivantes :

— un compteur *cnt* est utilisé : initialisé à zéro (ligne 4), il est incrémenté à chaque fois qu'une variable est prise dans Q (ligne 7) ;
— une variable entière *threshold* est ajoutée : elle correspond à la longueur à laquelle la propagation doit être stoppée ;
— un nouveau test apparaît en fin de boucle (lignes 17-18) pour stopper prématurément la propagation lorsque cela paraît utile.

Remarquons que si *threshold* vaut 1, MAC^c est équivalent à FC, tandis que si *threshold* est positionné à l'infini, MAC^c devient équivalent à MAC. En d'autres termes, MAC^c va en général effectuer plus de propagation que FC, mais moins que MAC.

Il nous faut maintenant trouver une bonne valeur de seuil (*threshold*) lors de la résolution d'une instance. À partir de distributions telles que celles des figures 4.1, 4.2, 4.3 et 4.4, on peut en premier lieu calculer, pour chaque longueur n, le coût moyen $AC(n)$ de détection d'un domaine vide (résultat *false*) lorsque le seuil vaut n. Pour une longueur n, le coût moyen $AC(n)$ est calculé comme suit :

$$\frac{\sum_{i=1}^{n} i \times (F_i + T_i) + \sum_{i=n+1}^{\infty} n \times (F_i + T_i)}{\sum_{i=1}^{n} F_i}$$

où T_i et F_i correspondent respectivement aux nombres de résultats *true* et *false* lorsque la longueur est égale à i. Ce coût est exprimé en nombre d'opérations de base effectuées par GAC^{var}, c'est-à-dire en nombre de fois où une variable est extraite de Q. Cette formule signifie que, pour un entier i compris entre 1 et n, le coût est égal à i multiplié par le nombre de fois où la longueur a été égale à i, et pour tout entier i plus grand que n, le coût est égal à n (car la propagation a été stoppée à ce niveau) multiplié par le nombre de fois où la longueur a été égale à i. Ce coût global est alors divisé par le nombre de fois où GAC^{var} a renvoyé *false* lorsque la longueur était égale à i. Une illustration est donnée par les figures 4.5, 4.6 et 4.7. Le coût représente donc une estimation de l'effort à produire pour aller jusqu'à une certaine longueur de propagation.

Nous pouvons alors contrôler la propagation en positionnant le seuil à la valeur de i qui détermine le coût le plus bas. Dans l'hypothèse où nous connaîtrions à l'avance la distribution, nous obtiendrions, par exemple, des seuils égaux à 1 pour l'instance graph-valiente, à 12 pour langford-3-12, ou encore à 149 pour pseudo-ii32c1 (voir les figures 4.1(b), 4.2(b) et 4.2(a)). En d'autres termes, MAC^c pourrait identifier 80% des choix conduisant à un domaine vide pour les instances graph-valiente et langford-3-12, et 100% pour pseudo-ii32c1 (voir les figures 4.1(b), 4.2(b) et 4.2(a)). Cela signifie également que si de telles valeurs de seuil avaient été fixées dès le début de la résolution, MAC^c se serait comporté de manière similaire à FC sur l'instance graph-valiente, et de manière similaire à MAC sur l'instance pseudo-ii32c1. En pratique, il n'est pas possible de déterminer ces valeurs à l'avance, aussi nous proposons une procédure dynamique et adaptative de calcul du seuil au cours de la recherche, ce qui produit des variations des valeurs du seuil autour des valeurs théoriques présentées plus haut. Sur nos exemples, MAC^c se comporte alors de façon légèrement différente de FC ou de MAC.

Au niveau de notre implémentation, nous avons utilisé le coût moyen minimum comme valeur de seuil. En fait, le seuil est recalculé en permanence au cours de la recherche en fonction des résultats précédemment obtenus. Nous calculons le seuil de la façon suivante : nous initialisons la variable *threshold* à partir des s premiers appels à la fonction GAC^{var} (sans contrôle de la propagation). Autrement dit, au début de la recherche, *threshold* est initialisé à l'infini. La longueur et le résultat de ces appels sont enregistrés dans une struc-

ture de file (FIFO) de taille s. La valeur de seuil ainsi calculée est alors utilisée pour f appels à GAC^{var}. Le $f+1^{ème}$ appel à GAC^{var} est effectué sans contrôle (comme si *threshold*

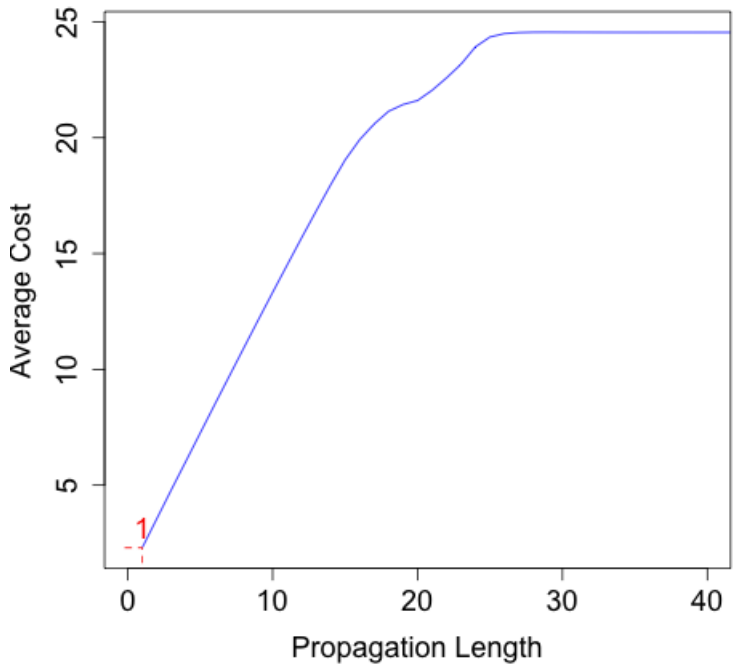


Figure 4.5 : *Coût moyen en fonction de la longueur de propagation pour l'instance graph-valiente.*

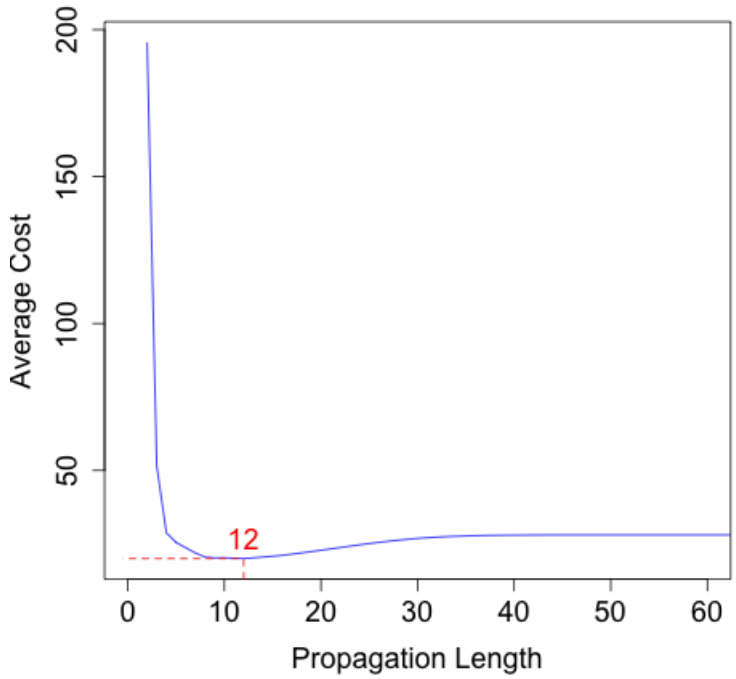


Figure 4.6 : *Coût moyen en fonction de la longueur de propagation pour l'instance langford-3-12.*

était positionné à l'infini – ceci n'est pas visible sur l'algorithme). La file va ensuite être mise à jour par simple suppression de la plus ancienne valeur et ajout de la plus récente, ce qui permet de faire évoluer la valeur du seuil. Ce processus est répété cycliquement jusqu'à

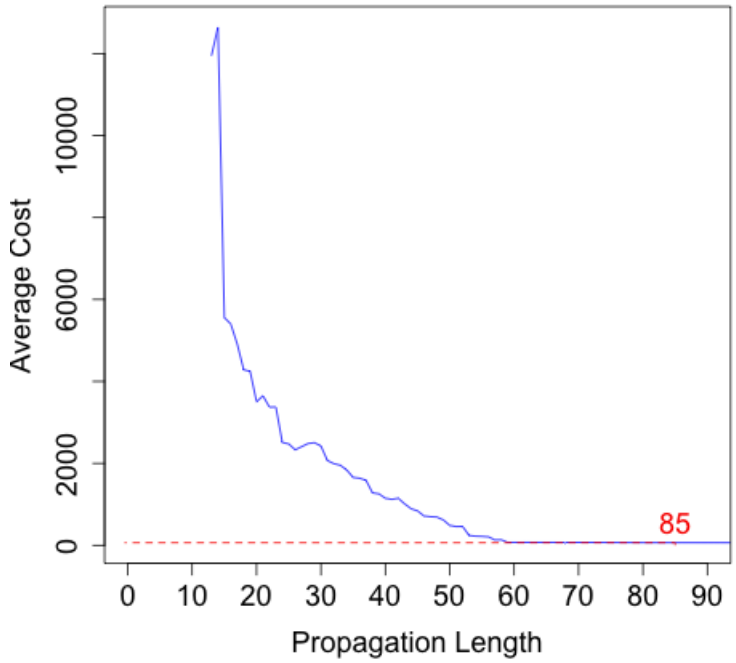


Figure 4.7 : *Coût moyen en fonction de la longueur de propagation pour l'instance pseudo-ii32c1.*

la fin de la recherche. Pour nos expérimentations, nous avons arbitrairement fixé les valeurs de s à 100, et de f à 10. Les résultats permettent donc d'envisager des prolongements intéressants et certaines pistes peuvent se révéler intéressantes comme, par exemple, l'utilisation des modèles de séries temporelles (par exemple, les modèles auto-régressifs) utilisés en économétrie [1]. Enfin, remarquons que le fait que *threshold* soit toujours supérieur ou égal à 1 nous assure de la correction de l'algorithme MAC^c : le niveau de filtrage reste au minimum celui de FC.

4.4 Résultats expérimentaux

Pour montrer l'intérêt pratique du contrôle de la propagation par contraintes à partir d'une analyse de nature statistique concernant la longueur de propagation, nous avons comparé l'efficacité de MAC, MAC^c et FC pour résoudre de nombreuses séries d'instances CSP. Avec l'utilisation de l'heuristique de choix de variable *dom/wdeg* [23] pour guider la recherche, 1338 instances parmi les 2053 (issues d'un nombre important de séries d'instances) ont été résolues par les trois algorithmes MAC, MAC^c and FC, en moins de 1200 secondes chacune. En moyenne, MAC^c est environ 10% plus rapide que MAC et deux fois plus rapide que FC. Ceci est illustré par le tableau 4.1 où, pour chaque série, le nombre nb_s d'instances résolues par les trois algorithmes dans le temps imparti (1200 secondes) ainsi que le nombre total nb d'instances sont donnés sous la forme (#Inst=nb_s/nb) en-dessous du nom de la série. Les temps CPU moyens affichés dans ce tableau sont calculés à partir des instances nb_s identifiées pour chaque série ; le nombre d'instances résolues par les trois algorithmes est indiqué à la ligne marquée par #résolus. MAC^c est particulièrement efficace sur les instances de colorations de graphes, et obtient d'assez bons résultats sur les séries pret, et qcp. Le tableau 4.2 fournit des détails sur la résolution de diverses instances

de la série. Pour chaque instance, le temps CPU total pour la résoudre est donné ainsi que le nombre de noeuds explorés (nds) et le nombre (ccks) de tests de contraintes (sauf pour

les contraintes non-binaires en extension où STR2 [70] est utilisé pour appliquer GAC). En général, lorsque MAC et MAC^c explorent à peu près le même arbre de recherche, MAC^c évite de nombreux tests de contraintes, comme on peut le voir pour david-10, qwh-20-166-0, lemma-50-9-mod. Une autre vision des résultats est donné par la figure 4.8 qui représente des nuages de points permettant une comparaison par paires pour MAC^c opposé à FC et à MAC.

4.5 Conclusion

Dans ce chapitre, nous avons montré qu'il existe une corrélation intéressante entre la *longueur* et le *résultat* de la propagation de contraintes. Cette observation nous a permis de faire des prévisions raisonnables quant à la capacité de la propagation de contraintes à détecter une incohérence et, par conséquent, nous a conduit à proposer une variante de l'algorithme de recherche MAC lorsque la propagation est stoppée prématurément (i.e. lorsque la probabilité de retour-arrière à partir du noeud courant est faible). Nous croyons que cette corrélation, observée pour la première fois dans la littérature, ouvre des perspectives pour améliorer l'efficacité des solveurs.

Comme perspective, nous projetons de développer des alternatives à notre mode actuel

Series		FC	MAC	MAC^c
crosswords-Vg	cpu	92.2	**20.5**	20.7
(#Inst = 200/258)	#résolus	200	214	**217**
graph-coloring	cpu	55.5	58.3	**41.4**
(#Inst = 198/459)	#résolus	210	203	**213**
graph-valiente	cpu	19.9	14.7	**14.2**
(#Inst = 681/793)	#résolus	681	**693**	690
pseudo-ii	cpu	**22.9**	39.1	45.5
(#Inst = 14/41)	#résolus	**31**	16	16
langford	cpu	75.9	**1.06**	1.46
(#Inst = 38/72)	#résolus	38	**47**	**47**
pret	cpu	703	368	**327**
(#Inst = 4/8)	#résolus	4	4	4
qcp	cpu	57.0	48.1	**31.8**
(#Inst = 35/60)	#résolus	**38**	35	**38**
qwh	cpu	45.8	**24.6**	25.3
(#Inst = 30/40)	#résolus	30	30	**31**
rand-3	cpu	190	140	**133**
(#Inst = 120/300)	#résolus	127	**132**	131
scens11	cpu	181	63.6	**59.3**
(#Inst = 9/12)	#résolus	9	**10**	**10**
schurrLemma	cpu	**28.8**	116	50.4
(#Inst = 9/10)	#résolus	9	9	9
All Series	cpu	62.8	36.3	**32.0**
(#Inst = 1338/2053)	#résolus	1387	1393	**1406**

Table 4.1: *Temps moyen (en secondes) pour résoudre les instances des différentes séries (avec un time-out de* 1200*s pour chaque instance) avec MAC-wdeg/dom.*

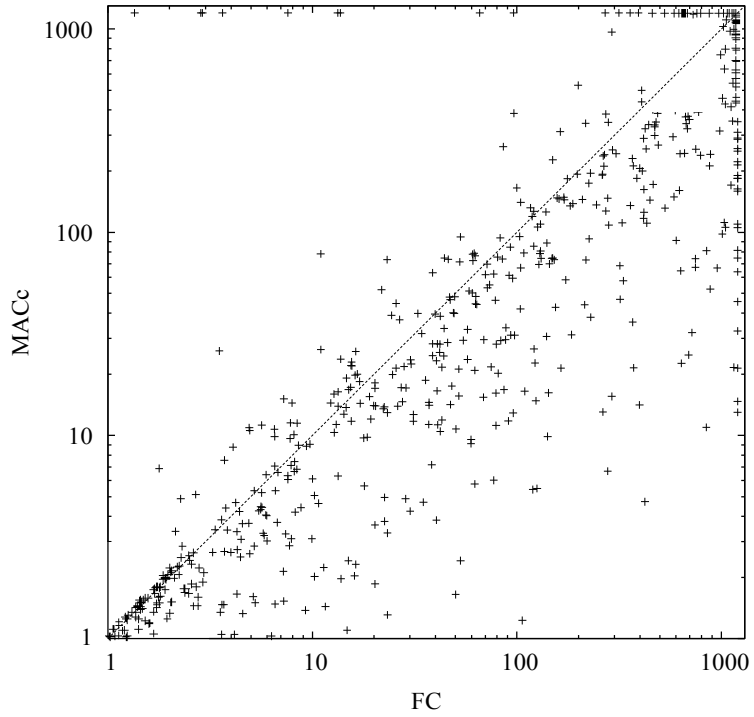
Instances		FC	MAC	MAC^c
cw-words-vg5-9	cpu	308	56.0	**47.0**
	nds	6,065*K*	127*K*	143*K*
2-insertions-5-3	cpu	17.5	**9.7**	**9.7**
	nds	447*K*	151*K*	163*K*
	ccks	2,149*K*	1,705*K*	1,495*K*
david-10	cpu	366	276	**225**
	nds	9,998*K*	4,439*K*	4,440*K*
	ccks	158*M*	163*M*	148*M*
graph-val-8-13	cpu	29.7	27.3	**23.6**
	nds	826*K*	405*K*	433*K*
	ccks	16*M*	14*M*	11*M*
pseudo-ii-32c1	cpu	**7.6**	12.9	14.3
	nds	120*K*	49,348	60,824
	ccks	1,107*K*	5,999*K*	6,506*K*
langford-3-12	cpu	1,059	**13.2**	21.7
	nds	31*M*	179*K*	345*K*
	ccks	701*M*	4,616*K*	12*M*
pret-60-25	cpu	662	354	**323**
	nds	22*M*	12*M*	10*M*
qcp-15-120-5	cpu	62.7	29.9	**21.1**
	nds	1,742*K*	601*K*	420*K*
	ccks	6,316*K*	7,498*K*	4,617*K*
qwh-20-166-0	cpu	157	76.4	**72.6**
	nds	4,213*K*	1,347*K*	1,348*K*
	ccks	14*M*	17*M*	15*M*
rand-3-20-18	cpu	48.6	31.1	**28.6**
	nds	315*K*	43,952	56,085
scen11-f4	cpu	1,191	358	**338**
	nds	20*M*	3,381*K*	3,718*K*
	ccks	757*M*	261*M*	277*M*
lemma-50-9-mod	cpu	**54.0**	202	89.9
	nds	660*K*	204*K*	204*K*
	ccks	86*M*	706*M*	280*M*

Table 4.2: *Résultats détaillés sur diverses instances (time-out de 1200 se-condes par instance) avec MAC-dom/wdeg.*

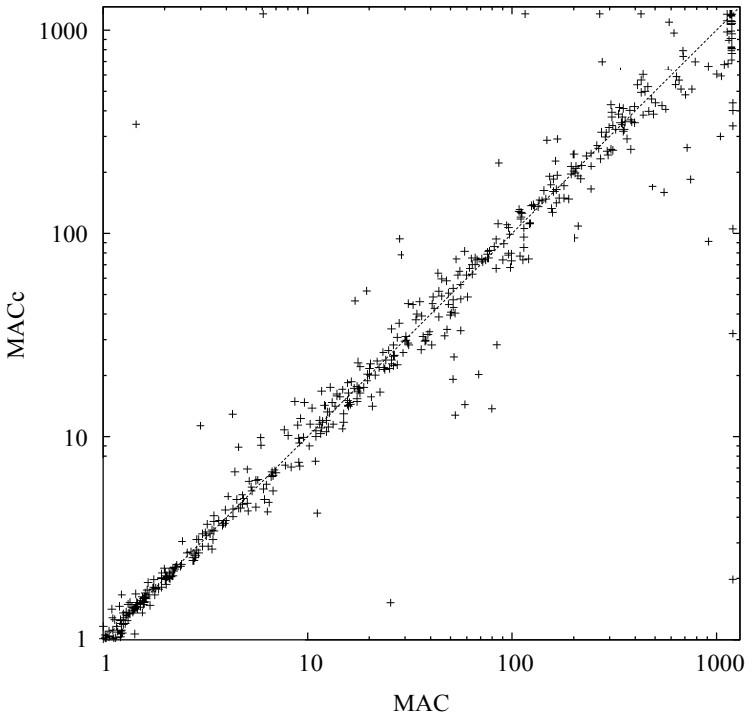
de calcul des valeurs de seuil par exemple, une procédure qui garantisse que x% d'échecs puisse être identifié (où x est l'objectif fixé par l'utilisateur). Nous aimerions également déterminer si certaines caractéristiques des graphes de contraintes telles que la densité, le diamètre, la longueur caractéristique des chemins et/ou le coefficient de regroupement (clustering) [118], peuvent être directement liées à l'efficacité du contrôle de propagation. D'autre part, il peut être intéressant de combiner l'approche probabiliste d'inférence [81] avec notre approche probabiliste de détection d'incohérence, car elles semblent être orthogonales. Enfin, parmi les perspectives que nous avons identifiées, une dernière piste intéressante est l'étude des techniques de shaving s'appuyant sur des tests singletons à coût d'obtention réduit. Cela est prometteur puisque seul le résultat de la propagation est utile pour le shaving. De manière générale, cela pourrait être lié également à la cohérence d'arc paresseuse [105].

Instances		FC	MAC	MAC^c
1-fullins-5-5	cpu	164	62.1	**59.4**
	nds	2, 123*K*	307*K*	346*K*
abb313GPIA-7	cpu	**16.2**	42.4	26.7
	nds	23, 643	9, 525	9, 615
anna-9	cpu	83.6	61.3	**56.2**
	nds	1, 006*K*	447*K*	447*K*
games120-7	cpu	4.26	3.51	**2.87**
	nds	43, 107	19, 355	19, 377
huck-8	cpu	8.91	**5.78**	6.12
	nds	112*K*	49, 197	50, 434
lei450-15a-08	cpu	**23.0**	65.3	35.1
	nds	183*K*	80, 112	80, 183
miles500-10	cpu	816	738	**565**
	nds	10*M*	4, 540*K*	4, 594*K*
myciel6-5	cpu	898	216	**186**
	nds	12*M*	2, 086*K*	1, 853*K*

Table 4.3: *Temps CPU en secondes (et nombre de noeuds visités) pour résoudre des instances représentatives du problème de coloration de graphes. L'heuristique est dom/wdeg.*



(a) MAC^c est représenté contre FC.



(b) MAC^c est représenté contre MAC.

Figure 4.8 : *Comparaison par paires (temps CPU) sur 2053 instances de diverses séries. Le time-out pour résoudre une instance est 1200 secondes.*

Chapitre 5

Graphe de contraintes

Pour pouvoir résoudre efficacement tout type de problème un solver de contraintes doit être capable de ne pas toujours utiliser la même stratégie et doit s'adapter au problème. Dans ce chapitre nous allons décrire une méthode permettant de créer de manière statistique de nouvelles liaisons entre les variables. Ces informations pourront être exploitées de diverses manières. Une heuristique de choix de variables peut en découler. Il est également possible de créer un graphe amenant à la constitution de sous-problèmes. En particulier dans le cas de problèmes insatisfiables, il sera parfois possible d'identifier des sous-problèmes insatisfiables.

5.1 Graphe de contraintes

Définition 5.1. Soit $R = (X, C)$ un réseau de contraintes. Nous appellerons **graphe primal** et nous noterons G_R le graphe non orienté dont :

— l'ensemble des sommets est X,

— l'ensemble des arêtes est donné par

$$A = \{(X, Y) \in X \times X \mid \exists C \in C \text{ tel que } \{X, Y\} \subseteq scp(C)\}.$$

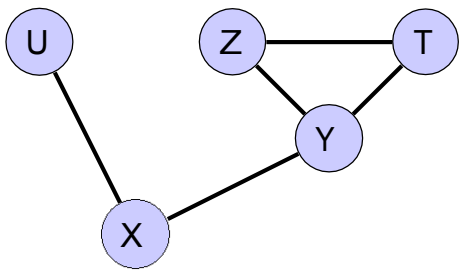
Exemple 5.1. Considérons le réseau de contraintes (X, C) comportant cinq variables X, Y, Z, T et U de domaines égaux à $[\![1, 5]\!]$. Les contraintes sont données par :

— $C_1 : X + Y = 3$,

— $C_2 : Y + Z + T = 1$,

— $C_3 : X + U > 2$.

Le graphe primal du réseau est alors le suivant :



Remarque 5.1. Dans la littérature, il existe d'autres (hyper)-graphes associés à un réseau de contraintes : hypergraphe de contraintes (macrostructure), hypergraphe de compatibilité (microstructure) ou encore graphe dual [69]. L'hypergraphe de contraintes, par exemple, est un hypergraphe dans lequel les hyperarêtes sont données par le scope de chaque contrainte. La figure 5.1 montre l'hypergraphe de contraintes du réseau de contraintes de l'exemple 5.1.

La difficulté de ce type de représentation réside dans l'équilibre entre la simplicité du graphe et la quantité d'informations retranscrites.

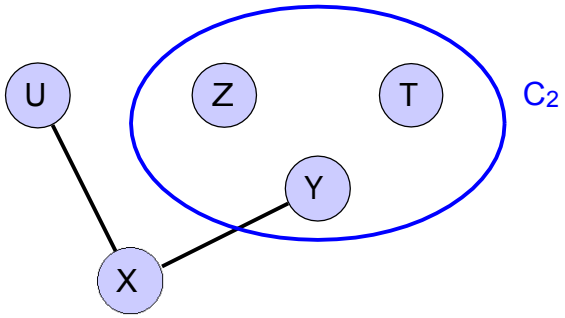


Figure 5.1 : *Hypergraphe de contraintes du réseau de l'exemple 5.1.*

La représentation d'un problème sous la forme d'un graphe primal est très basique. Notre objectif ici, est de faire apparaître des liaisons cachées entre les variables. Comme nous le verrons dans la section suivante, nous avons choisi de tenter de les obtenir statistiquement.

5.2 Graphe statistique

Dans cette section, nous décrirons la construction d'un graphe obtenu de manière statistique et qui peut évoluer au cours de la recherche. L'objectif est d'apprendre rapidement quelles sont les variables liées dans le problème. Plus précisément, un début de recherche ou l'utilisation d'un algorithme MAC (branches gloutonnes maintenues AC) permettront d'évaluer l'impact d'une décision d'une variable sur une autre. Un nouveau graphe de type graphe primal et dont les sommets sont les variables pourra alors être construit. Les informations collectées pourront être exploitées de diverses manières. Dans les sections suivantes, nous monterons comment les utiliser afin de détecter rapidement l'incohérence de certains problèmes.

5.2.1 Collecte de données statistiques

Dans cette étape, une collecte de données est lancée sur une partie de la recherche. L'objectif est d'obtenir une série statistique évaluant l'impact des décisions sur les domaines des variables.

Notre collecte de données dépend d'un entier n qu'il faut fixer avant de commencer. Il s'agit du nombre d'instanciations attendues pour arrêter la collecte.

Pour obtenir les données, il est possible de lancer un algorithme tel que MAC pour maintenir l'arc cohérence de branches gloutonnes. Lorsque n instanciations ont été effectuées, la collecte s'arrête. Pour chaque décision prise lors de cette opération, l'impact sur le domaine des autres variables est enregistré. Plus précisément, lorsqu'une décision est prise sur une variable X_0, nous nous intéressons aux variables dont le domaine a été modifié. Si le domaine d'une variable X_1 est modifié, alors deux cas de figure se présentent. Soit le domaine de X_1 n'a jamais été modifié par une décision sur X_0. Dans ce cas, un compteur pour X_1 est initialisé à 1 dans un tableau indexé par X_0. Soit le domaine de X_1 a déjà été modifié par une décision sur X_0. Dans ce cas, le compteur de X_1 dans le tableau indexé par X_0 est incrémenté de 1. Dans tous les cas, nous enregistrons également le nombre de décisions sur X_0.

Exemple 5.2. Une collecte de données peut donner le début de fichier suivant :

```
21 variables
X0 #decs= 52
X1:1 X2:4 X3:6 X4:5 X5:6 X6:5 X7:4 X8:3 X9:4 X10:3 X11:4 X12:1
```

```
X1 #decs= 54
X0:4 X2:5 X3:4 X4:4 X5:1 X6:5 X7:5 X8:5 X9:3 X10:4
```

Dans ce cas n a été fixé à 100. Le problème comporte 21 variables. Nous voyons que 52 décisions ont été prises pour la variable X_0 et lors de ces décisions le domaine de la variable X_1 a été modifié 1 fois, le domaine de la variable X_2 a été modifié 4 fois, etc.

Cette idée a besoin d'une formulation mathématique, ce qui est donné par la définition suivante.

Définition 5.2. Soit (X, C) un réseau de contraintes contenant n variables. Une **matrice de poids** du réseau R est une matrice $(a_{ij})_{i,j}$ de taille $n \times n$ à coefficients dans $[0, 1]$.

Pour une heuristique de variables fixée $X_1 < \ < X_n$ et $i, j \in$J1, nK, le coefficient a_{ij} correspond au poids qu'on souhaite donner à (X_i, X_j).

Ainsi, la collecte des données permet d'obtenir une matrice de poids en considérant le quotient du nombre de changement du domaine sur le nombre de décisions. Il est à noter qu'une variable qui n'est pas touchée apparaît dans la matrice de poids avec le coefficient 0.

Exemple 5.3. Considérons un réseau de contraintes contenant 4 variables X_1, X_2, X_3 et X_4 et dont les domaines des variables sont égaux à J0, 100K. Supposons qu'une collecte de données sur ce réseau donne :

```
4 variables
X1 #decs= 27
X2:14 X3:7 X4:15
X2 #decs= 15
X1:4 X3:4 X4:12
X3 #decs= 15
X1:4 X2:5 X4:9
X4 #decs= 24
X1:4 X2:13 X3:5
```

La matrice de poids associée pour l'heuristique $X_1 < X_2 < X_3 < X_4$ est :

$$\begin{pmatrix} 0 & \frac{14}{27} & \frac{7}{27} & \frac{15}{27} \\ \frac{4}{15} & 0 & \frac{4}{15} & \frac{12}{15} \\ \frac{5}{15} & \frac{5}{15} & 0 & \frac{9}{15} \\ \frac{4}{24} & \frac{13}{24} & \frac{5}{24} & 0 \end{pmatrix}$$

5.2.2 Construction d'un graphe statistique

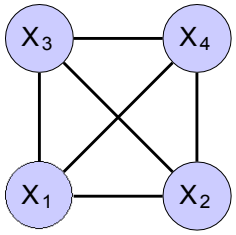
À partir des données collectées nous allons construire un graphe dont chaque arête devra représenter un lien entre les variables. Pour simplifier, nous allons construire un graphe non pondéré. Théoriquement un graphe pondéré serait plus intéressant, cependant dans nos expérimentations cela n'apporte pas grand chose.

Définition 5.3. Soit (X, C) un réseau de contraintes avec n variables et $M = (a_{ij})_{i,j}$ une matrice de poids pour le réseau R et pour l'heuristique $X_1 < \ < X_n$. Nous noterons G_S le graphe non orienté dont les sommets sont les variables de R et pour lequel deux sommets X_i et X_j sont liés si $a_{ij} > 0$. Lorsque le contexte sera clair, nous noterons simplement et par abus G_S.

Exemple 5.4. Reprenons l'exemple 5.3. Nous avions,

$$M = \begin{pmatrix} 0 & \frac{14}{27} & \frac{7}{27} & \frac{15}{27} \\ \frac{4}{15} & 0 & \frac{4}{15} & \frac{12}{15} \\ \frac{5}{15} & \frac{5}{15} & 0 & \frac{9}{15} \\ \frac{4}{24} & \frac{13}{24} & \frac{5}{24} & 0 \end{pmatrix}$$

Le graphe G_S^M est donné par :



Avec cette construction nous avons trop de liaisons et cela ne met pas en avant les liaisons intéressantes. Ceci justifie l'introduction de l'application de la définition 5.4. Par ailleurs, cette construction permet de retrouver le graphe primal. En effet, si nous reprenons l'exemple 5.1 et considérons la matrice

$$Adj = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

avec l'ordre $T < U < X < Y < Z$, le graphe primal est donné par G_S^{Adj}.

Définition 5.4. Soit n un entier naturel et $q \in [0, 1]$. Nous noterons s_q l'application de $M_n([0, 1])$ dans $M_n([0, 1])$ qui à une matrice $(a_{ij})_{i,j} \in M_n([0, 1])$ associe la matrice $(b_{ij})_{i,j}$ tel que

$$b^{ij} = \begin{cases} a_{ij} & \text{s'il y a au moins } nq \text{ éléments de } \{a_{i1}, \ldots, a_{i,n}\} \text{ plus petits ou égaux à } a_{ij}. \\ 0 & \text{sinon.} \end{cases}$$

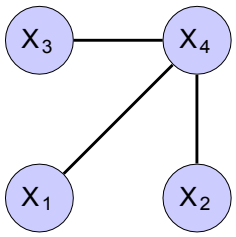
Cette application va nous permettre de ne laisser que les meilleures liaisons. En effet, nous allons pouvoir ne garder que les arêtes du graphe correspondant à un pourcentage donné des poids les plus forts.

Définition 5.5. Soit $q \in [0, 1]$ et M une matrice de poids. Nous noterons $G_S^M(q)$ le graphe $G_S^{s_q(M)}$. Ce graphe sera noté simplement $G_S(q)$ si le contexte est clair.

Exemple 5.5. Reprenons l'exemple 5.3. Nous avons

$$s_{0,75}(M) = \begin{pmatrix} 0 & 0 & 0 & \frac{15}{27} \\ 0 & 0 & 0 & \frac{12}{15} \\ 0 & 0 & 0 & \frac{9}{15} \\ 0 & \frac{13}{24} & 0 & 0 \end{pmatrix}$$

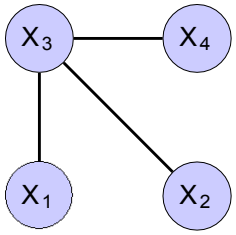
Le graphe $G_S(0, 75)$ est alors :



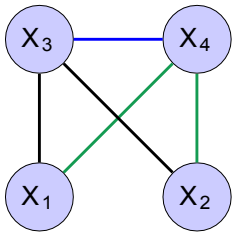
Nous supposerons, ici que les contraintes du réseau initial sont :

$$C_1 = X_1 + X_3$$
$$C_2 = X_3 + X_4$$
$$C_2 = X_3 + X_2$$

Le graphe primal est alors :



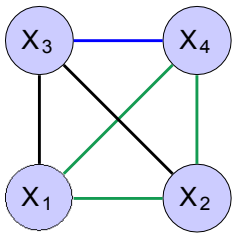
Nous pouvons alors représenter les deux graphes sur un seul en indiquant en vert les arêtes du graphe statistique qui ne sont pas dans le graphe primal. Nous avons alors :



Nous faisons alors apparaître les nouvelles liaisons. Nous avons :

$$s_{0,25}(M) = \begin{pmatrix} 0 & \frac{14}{27} & 0 & \frac{15}{27} \\ \frac{4}{15} & 0 & \frac{4}{15} & \frac{12}{15} \\ & \frac{5}{15} & 0 & \frac{9}{15} \\ 0 & \frac{13}{24} & \frac{5}{24} & 0 \end{pmatrix}$$

Nous avons donc le graphe :



Cette construction permet de construire d'une multitude de manières un graphe à partir de données collectées. Il suffit de modifier la matrice M et l'application s_q.

La collecte de données ne fait, à priori, pas apparaître de symétries. Il est donc plus naturel de définir un graphe orienté. Le plus proche de la réalité serait un graphe orienté et pondéré. Cependant, les résultats obtenus sont meilleurs avec un graphe non orienté. Définissons tout de même cet objet.

Définition 5.6. Soit (X, C) un réseau de contraintes avec n variables et $M = (a_{ij})_{i,j}$ une matrice de poids pour le réseau R et pour l'heuristique $X_1 < \ < X_n$. Nous noterons G_{SD}^M le graphe orienté dont les sommets sont les variables de R et qui contient l'arc (X_i, X_j) si $a_{ij} > 0$. Lorsque le contexte sera clair, nous noterons simplement et par abus G_{SD}.

Tout comme dans le cas non orienté, nous noterons $G_{SD}^M(q)$ le graphe $G_{SD}^{s_q(M)}$. Ce graphe sera noté simplement $G_{SD}(q)$ si le contexte est clair.

La figure 5.3 montre le problème Scen11-f3. Il représente en particulier le graphe $G_S(0,75)$ et le graphe primal. Comme précédemment, les sommets sont les variables du problème. Le graphe $G_S(0,75)$ est en bleu en dehors des arêtes du graphe qui sont de nouvelles liaisons (i.e : ne sont pas dans le graphe primal). Les sommets et les arêtes qui sont dans le graphe primal mais pas dans $G_S(0,75)$ sont en gris.

Nous voyons sur les figures 5.2, 5.3, 5.4, 5.5, 5.6 et 5.7 que le réglage du quantile influence fortement les informations obtenues.

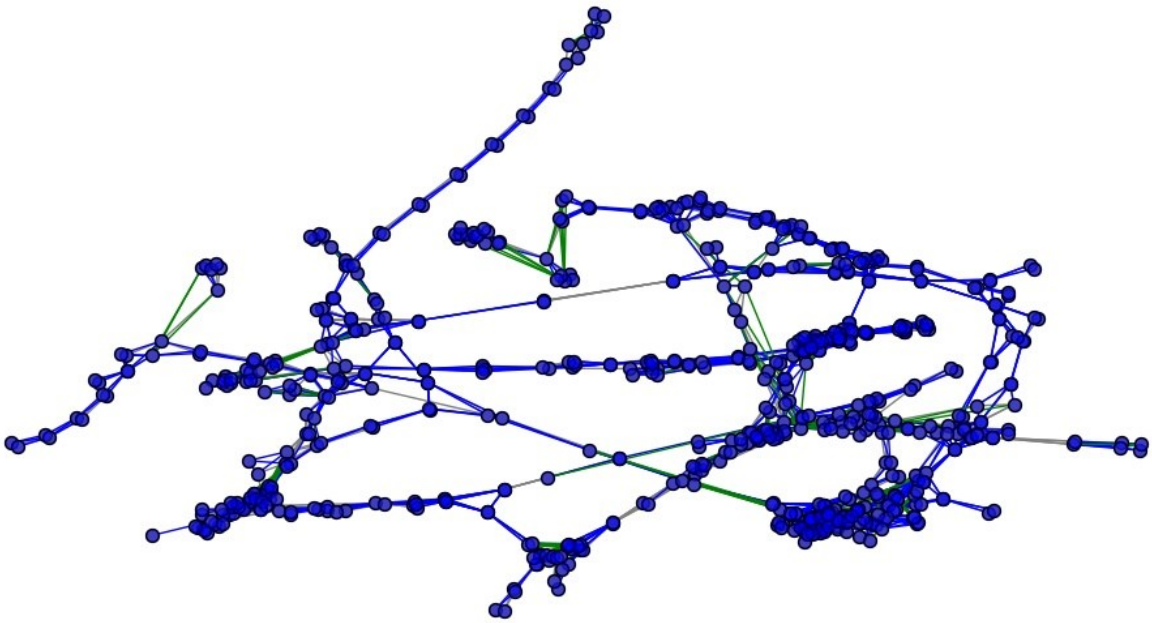


Figure 5.2 : G_R - *Scen11-f3 quantile* = 0, 5.

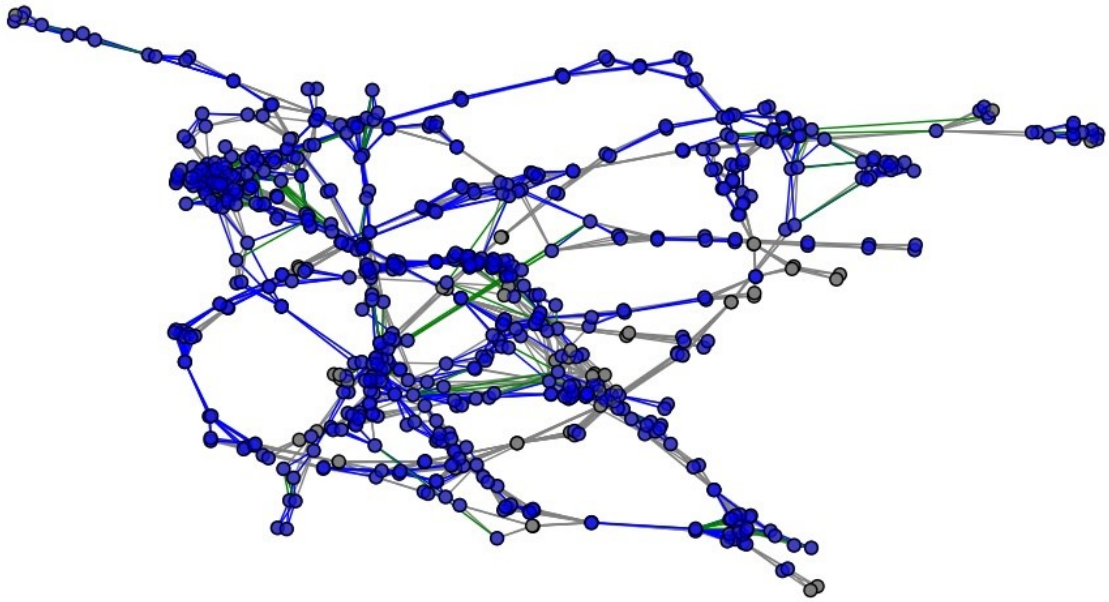


Figure 5.3 : G_R - *Scen11-f3 quantile* = 0, 75.

5.3 Sous-problème inconsistant

Le graphe obtenu de manière statistique a pour but de faire apparaître les liaisons fortes (cachées ou implicites) entre les variables. Pour une majorité de paramètres q, ce graphe est non connexe et sur les problèmes inconsistants, les noyaux sont souvent contenus dans des composantes connexes. Cela peut permettre donc de réduire un problème et

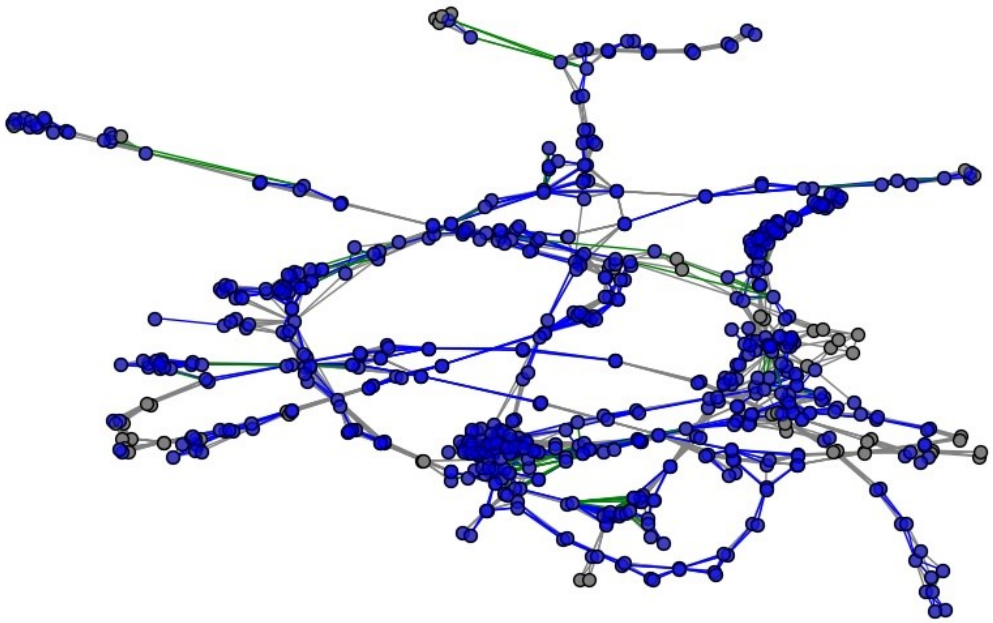


Figure 5.4 : G_R - *Scen11-f3 quantile* = 0, 8.

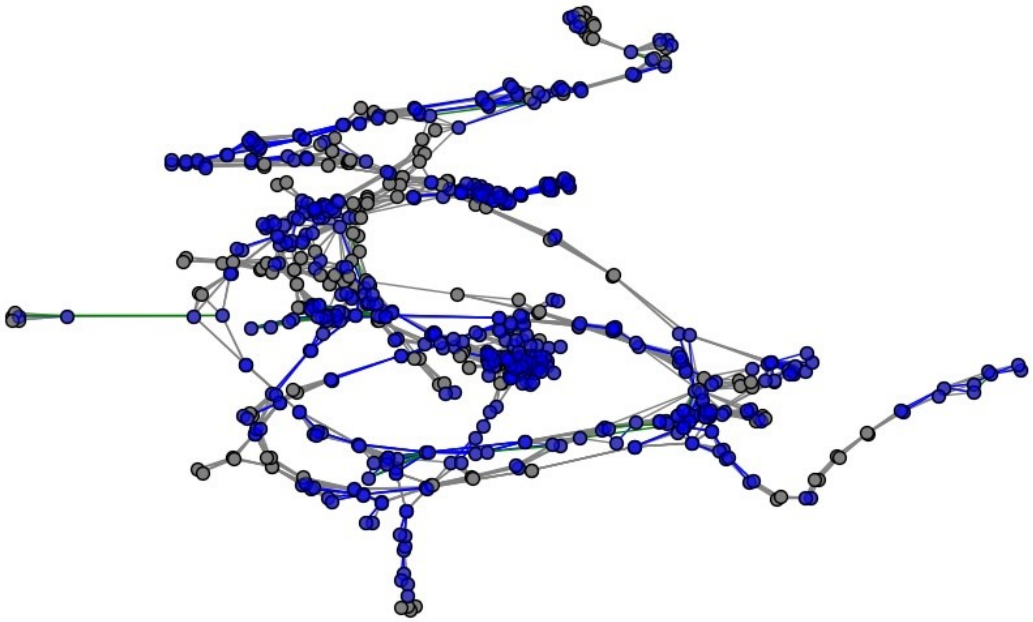


Figure 5.5 : G_R - *Scen11-f3 quantile* = 0, 9.

d'obtenir rapidement une incohérence. Dans les figures suivantes les sommets d'un noyau seront représenté en rouge. La figure 5.8 donne une représentation des graphes pour le problème Scen11-f9 pour $q = 0, 5$. La figure 5.9 montre une composante connexe du graphe statistique pour le problème Scen11-

f9. Le problème initial comporte 680 sommets. La composante connexe comporte 575 sommets et contient un noyau inconsistant. Il s'agit

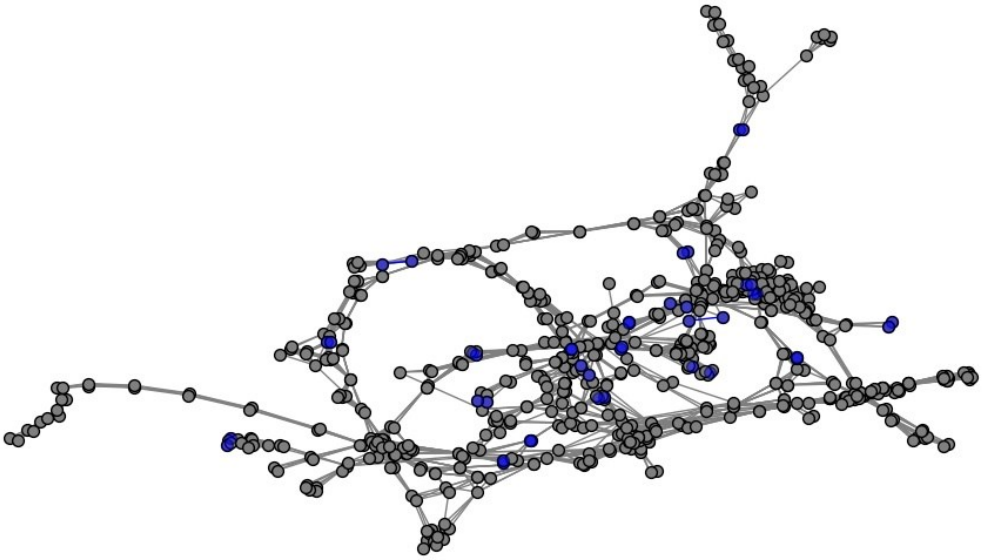


Figure 5.6 : G_R *- Scen11-f3 quantile* = 0, 95.

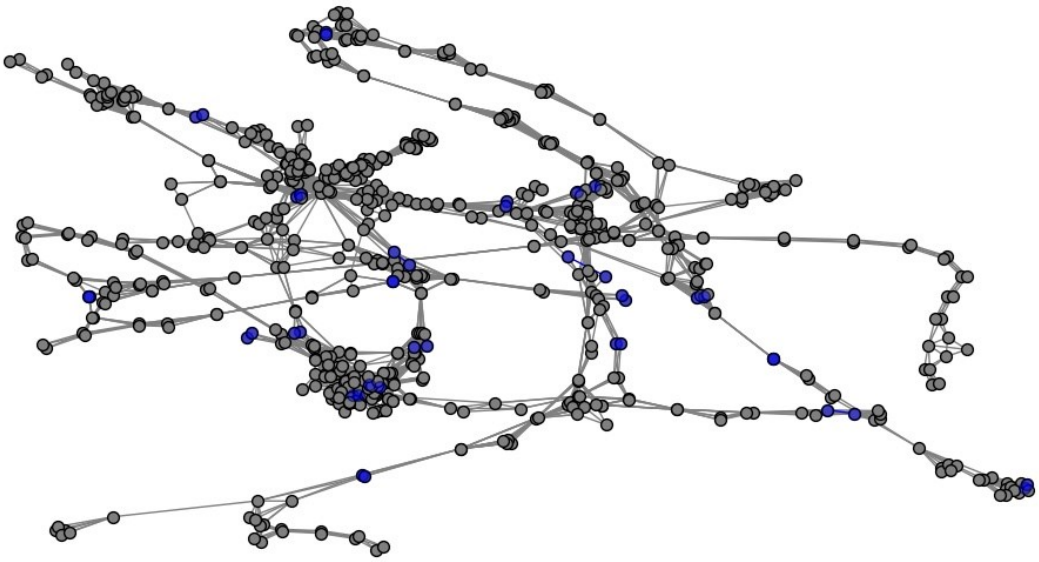


Figure 5.7 : G_R *- Scen11-f3 quantile* = 0, 99.

donc d'un sous-problème inconsistant.

Le réglage du quantile influence le nombre de variables des composantes connexes du graphe obtenu de manière statistique. Nous pouvons le voir en considérant le problème QueenKnight-25. La figure 5.10 nous montre une représentation des graphes pour q = 0, 75. Un composante connexe est donnée avec la figure 5.11. Il s'agit bien d'un sous-problème inconsistent mais la réduction est légère. Les mêmes représentations sont données pour q = 0, 90 avec les figures 5.12 et 5.13. Nous voyons que le noyau inconsistant est trouvé immédiatement.

Cette technique peut donc permettre de trouver très rapidement une incohérence pour

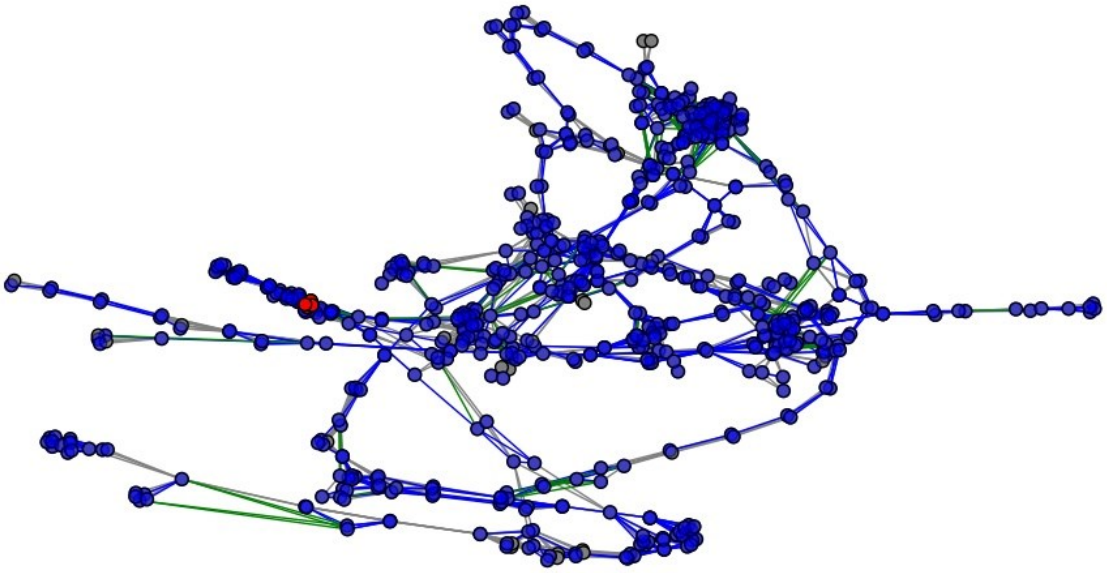


Figure 5.8 : G_R - *Scen11-f9 quantile*= 0,5 - 680 *sommets.*

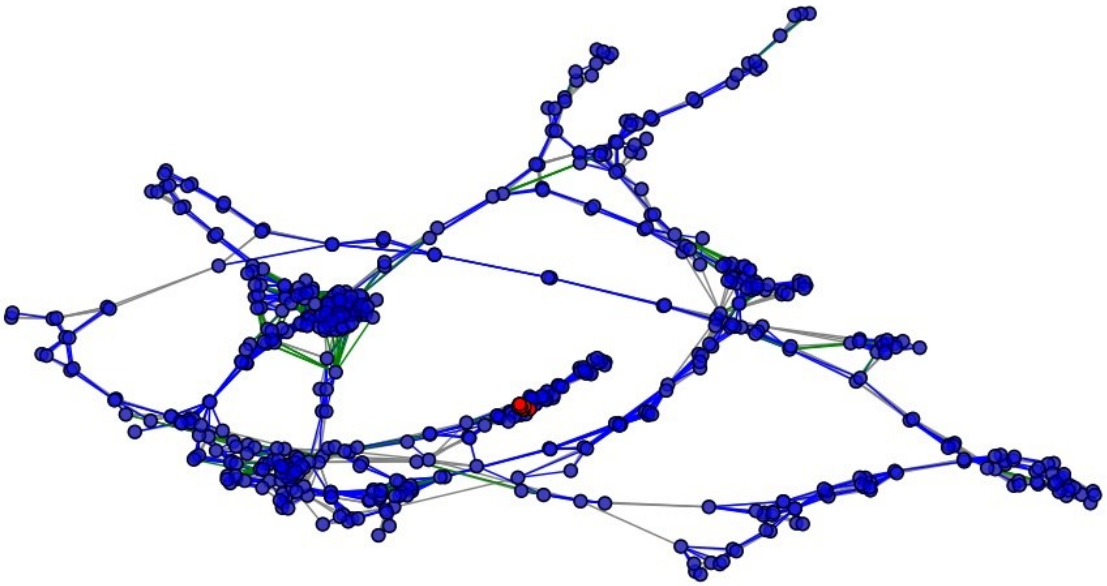


Figure 5.9 : *Composante connexe - Scen11-f9 - quantile*= 0,5 - 575 *sommets.*

certains problèmes.

Le réglage du quantile reste toutefois déterminant. Il est possible de réaliser plusieurs tests pour différentes valeurs ou d'utiliser des techniques de coût similaires à celles utilisées dans le chapitre 4. Des travaux sont en cours sur ce sujet.

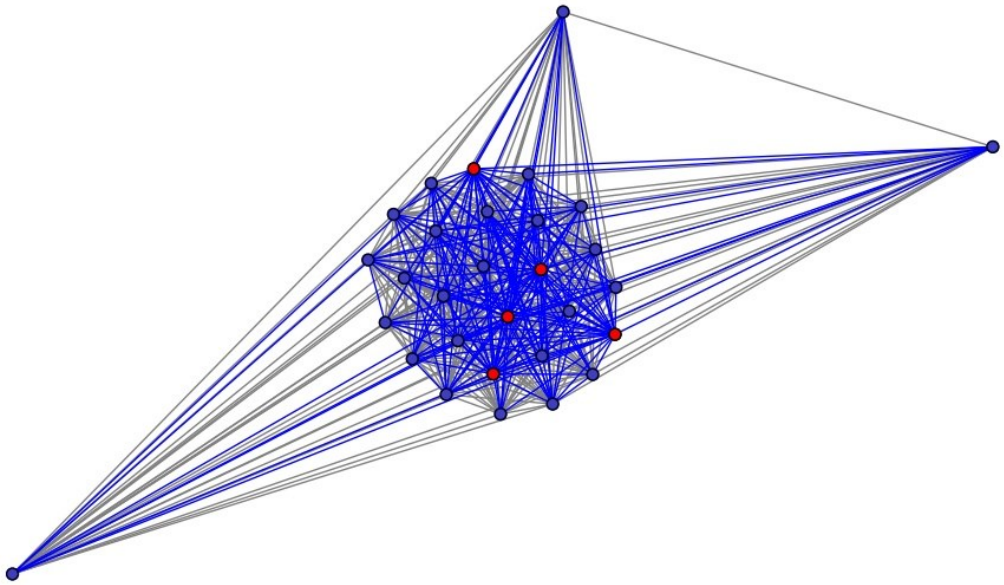


Figure 5.10 : G_R - *QueenKnight-25 - quantile*= 0,75.

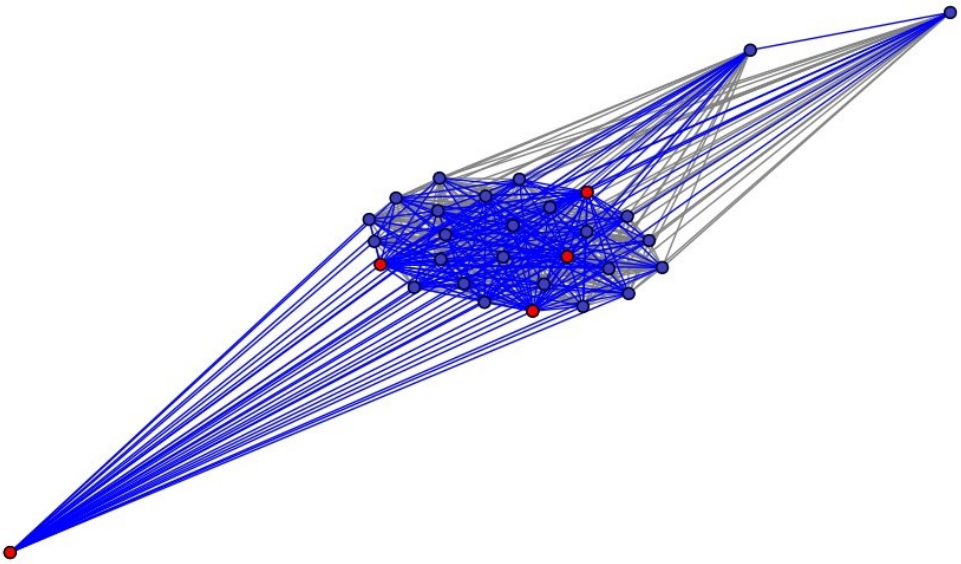


Figure 5.11 : *Composante connexe - QueenKnight-25 - quantile*= 0,75.

5.4 Ajout de contraintes tables

5.4.1 Principe

En faisant apparaître de nouvelles liaisons, nous pouvons espérer faire émerger des sous-problèmes intéressants même s'ils ne sont pas toujours inconsistants. L'idée est ici d'utiliser la matrice de poids issue de la collecte des données afin de créer le sous-problème le plus avantageux possible. Résoudre ce problème permettra alors d'ajouter des contraintes tables.

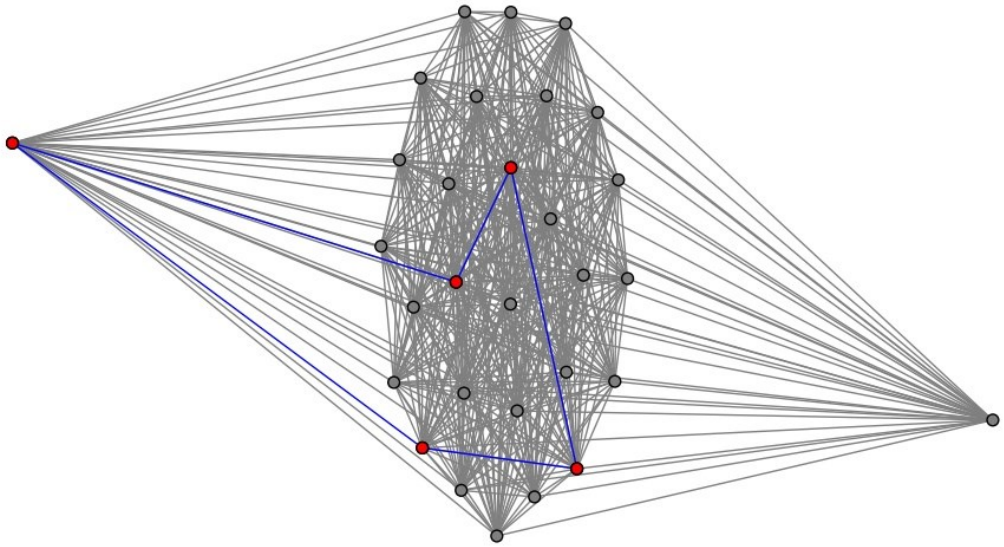


Figure 5.12 : G_R *- QueenKnight-25 - quantile*= 0, 9.

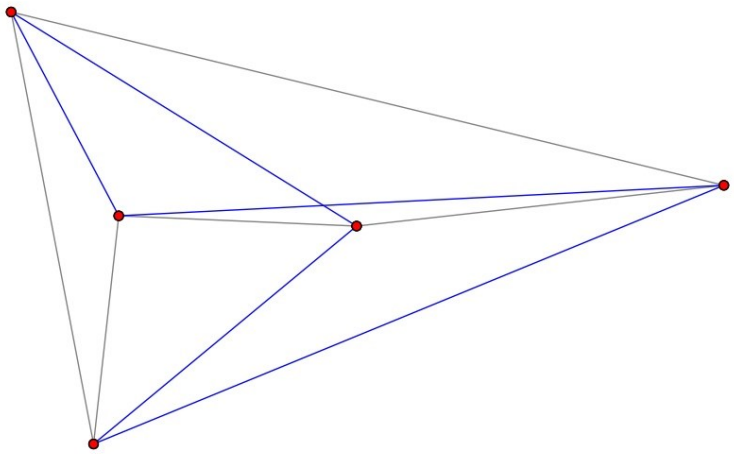


Figure 5.13 : *Composante connexe - QueenKnight-25 - quantile*= 0, 9.

Définition 5.7. Une contrainte **table** est une contrainte définie en extension et listant des tuples autorisés ou interdits.

Définition 5.8. Soit R = (X, C) un réseau de contraintes contenant n variables $(X_1, \ldots, X_n)$. Soit $M = (a_{ij})_{i,j}$ une matrice de poids associée à l'heuristique $X_1 < \ < X_n$. Pour un ensemble $S \subseteq$X et une variable $X_{i_0} \in$X $\setminus S$, nous noterons

$$p_S(X_{i_0}) = \sum_{X_i \in S} a_{ii_0} + a_{i_0 i}.$$

Nous noterons $\prec_M$ la relation d'ordre sur $S \subseteq$X définie par

$$X_i \prec_S X_j \quad ssi \quad p_S(X_i) < p_S(X_j)$$
$$ou\ p_S(X_i) = p_S(X_j) \text{ et } i \leq j.$$

La relation binaire $\prec_S$ est une relation d'ordre total, ce qui permet de définir la suite $(u_k(M))_{1 \leq k \leq n}$ de la manière suivante :

— Les deux premiers éléments de la suite $u_1(M) = X_{i_0}$ et $u_2(M) = X_{j_0}$ sont tels que :
  — $i_0 < j_0$,
  — $a_{i_0 j_0} + a_{j_0 i_0} = \max\{a_{ij} + a_{ji} \mid i, j \in \llbracket 1, n \rrbracket, i = j\}$,
  — $\forall i, j \in \llbracket 1, n \rrbracket, a_{i_0 j_0} + a_{j_0 i_0} = a_{ij} + a_{ji} \Rightarrow i_0 < i$ ou ($i_0 = i$ et $j_0 \leq j$).
— $\forall k \in \llbracket 3, n \rrbracket, u_k(M)$ est l'élément maximum dans $X \setminus S_{k-1}$ pour $\prec_{S_k}$ avec $S_{k-1} = \{u_1(M), \ldots, u_{k-1}(M)\}$.

À chaque étape, une variable est ajoutée de manière à avoir la meilleure liaison avec l'ensemble des variables déjà sélectionnées. Pour obtenir un sous-problème avec cette méthode, il faut se fixer un entier $k \in \llbracket 1, n \rrbracket$. Nous considérons alors le sous-réseau de contraintes R_k engendré par l'ensemble des variables $\{u_1(M), \ldots, u_k(M)\}$. Le réseau R_k est alors résolu de manière à lister les tuples autorisés (ou interdits) et créer ainsi une contrainte table pour le réseau R.

Exemple 5.6. Soit (X, C) un réseau de contraintes avec 4 variables et $M = (a_{ij})_{i,j}$ la matrice de poids pour le réseau R et pour l'heuristique $X_1 < < X_4$ définie par

$$M = \begin{pmatrix} 0 & 0,3 & 0,4 & 0,1 \\ 0,5 & 0 & 0,6 & 0,2 \\ 0,7 & 0,1 & 0 & 0,1 \\ 0,1 & 0,8 & 0,6 & 0 \end{pmatrix}$$

La meilleure liaison est (X_1, X_3) avec le poids $a_{1,3} + a_{3,1} = 0,4 + 0,7 = 1,1$. Nous avons $S_2 = \{X_1, X_3\}$. Le poids relatif à X_2 pour cet ensemble de variables est $0,3 + 0,5 + 0,6 + 0,1 = 1,5$. Le poids relatif à X_4 pour cet ensemble de variables est $0,1 + 0,1 + 0,1 + 0,6 = 0,9$. C'est donc la variable X_4 qui est l'élément maximum pour $\prec_{S_2}$. Nous avons donc

$$\begin{aligned} u_1(M) &= X_1 \\ u_2(M) &= X_3 \\ u_3(M) &= X_4 \\ u_4(M) &= X_2. \end{aligned}$$

5.4.2 Résultats expérimentaux

Dans la pratique, il semble difficile de prendre des sous-problèmes trop volumineux. Nous avons donc fixé à 5 le nombre de variables des sous-problèmes permettant de créer les contraintes tables. Une synthèse des résultats obtenus est donnée dans les tableaux 5.1 et 5.2. Dans la colonne notée MAC$^+$ les instances sont augmentées des contraintes tables apprises statistiquement et résolues avec l'algorithme MAC. Sur ces expériences, la méthode permet de gagner un peu de temps sur certaines catégories d'instances. Le gain n'est pas encore assez significatif et la méthode doit être améliorée. Cependant, l'approche est assez souple pour permettre d'inclure des idées nouvelles et d'apporter les ajustements nécessaires.

Instances		MAC	MAC+
1-fullins-5-5	cpu	23.8	**22.5**
	mem	553.4*K*	553.4*K*
3-insertions-3-3	cpu	28.8	**21.9**
	mem	1571.7*K*	**1026.5K**
aim-200-2-0-sat-4_ext	cpu	8.5	**5.0**
	mem	481.4*K*	**413.3K**
anna-10	cpu	502.2	**225.7**
	mem	1085.9*K*	**290.5K**
anna-9	cpu	56.9	**19.2**
	mem	1027.1*K*	**754.4K**

Table 5.1: *Temps CPU en secondes pour résoudre les instances.*

5.5 Conclusion

Les résultats obtenus avec la collecte d'informations statistiques sont encourageant. Des sous-problèmes inconsistants peuvent parfois être identifiés et l'ajout de contraintes tables peut accélérer la résolution.

Cette méthode est assez générique et permet toutes sortes de variations. Les informa-tions peuvent être également utilisées de différentes manières.

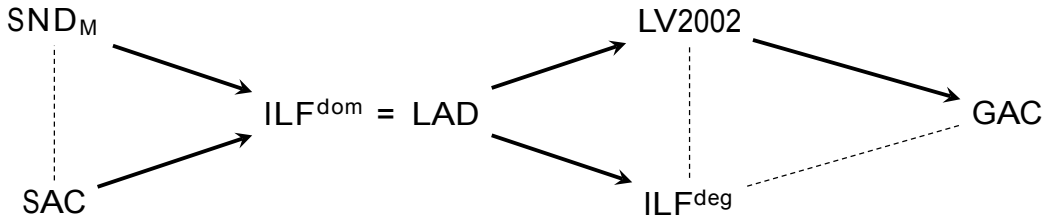
Cette technique peut permettre d'engendrer de nouvelles contraintes au cours de la recherche ou de créer de nouveau sous-problèmes. De nouvelles données peuvent compléter les données recueillies tout au long de la résolution. Les nouvelles liaisons créées peuvent donner une heuristique de variables en considérant, par exemple, le degré des sommets obtenus de manière statistique. Il serait également possibles de pondérer les liaisons. Les variables instanciées seraient alors liées. Au contraire, nous pourrions également créer des heuristiques afin d'instancier des variables ayant le moins de liaisons possibles afin de trouver au plus vite une incohérence.

Instances		MAC	MAC+
cril-6_ext	cpu	3.5	**2.2**
	mem	413.1*K*	413.1*K*
cril-7.ext	cpu	6.2	**1.4**
	mem	481.2*K*	**344.9K**
david-10	cpu	458.3	**146.0**
	mem	**616.4K**	2867.2*K*
david-8	cpu	5.3	**2.5**
	mem	**277.2K**	481.6*K*
dubois-20_ext	cpu	49.2	**15.7**
	mem	2662.0*K*	**889.9K**
dubois-23_ext	cpu	482.6	**115.4**
	mem	**1844.1K**	2128.7*K*
fapp07-0600-0	cpu	**2.9**	3.0
	mem	311.1*K*	311.1*K*
fapp10-0900-6	cpu	19.1	**0.1**
	mem	323.5*K*	**0.1K**
homer-13	cpu	**1.2**	4.7
	mem	**279.1K**	2461.1*K*
huck-10	cpu	228.0	**25.6**
	mem	**929.0K**	1367.5*K*
rand-2-40-8-753-100-50_ext	cpu	3.7	**1.3**
	mem	**277.6K**	482.1*K*
rand-2-40-8-753-100-93_ext	cpu	3.3	**1.5**
	mem	277.6*K*	**482.1K**
rand-2-40-8-753-100-96_ext	cpu	5.1	**2.6**
	mem	**345.7K**	482.1*K*
fapp40-3000-4	cpu	**26.4**	85.1
	mem	**486K**	737*K*
fapp40-3000-8	cpu	**31.4**	96.7
	mem	**486K**	2381*K*
fpsol2-i-1-05	cpu	**2.79**	4.85
	mem	291*K*	291*K*
scen11-f6	cpu	10.1	**9.7**
	mem	352.8*K*	**215.1K**
schur	cpu	12.4	**3.7**
	mem	**284.4K**	352.5*K*

Table 5.2: *Temps CPU en secondes pour résoudre des instances.*

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés à diverses relations entre la théorie des graphes et les réseaux de contraintes. Nous avons pu constater que ces relations sont fortes. En nous inspirant de travaux antérieurs d'isomorphismes de sous-graphes (comme LAD), nous avons construit un filtrage générique pour le problème d'isomorphisme de sous-graphe. Ceci nous a permis de généraliser ainsi un certain nombre de résultats existants. Nous avons montré également que notre méthode englobait de plus la partie graphes additionnels dans les travaux plus récents de [79]. Nous avons défini les fonctions de voisinage, ce qui donne un cadre général pour construire des voisinages de sommets. Elles serviront ainsi à sélectionner des sommets du graphe motif et du graphe cible dans le but d'effectuer des comparaisons. Nous avons également défini les fonctions de score, ce qui donne un cadre général pour les fonctions de comparaison entre sommets du graphe motif et sommets du graphe cible (degré, coefficient clustering, nombre de chemins, etc). Nous avons utilisé notre modèle avec des fonctions de voisinage et de score particulières : Id, adj et all pour le voisinage et $(M^k)_k$ pour le score. Nous avons défini ainsi des fonctions de score exploitant les propriétés des puissances des matrices d'adjacence. Nous avons montré que notre algorithme était plus fort en terme de filtrage que ILF, LAD, LV2002 et GAC, puis qu'il était équivalent à SAC.



C. McCreesh et P. Prosser ont constaté dans [79] que sur certains problèmes leur algorithme résolvait plus d'instances que celui présenté dans cette thèse dans un temps limité. Pour améliorer notre méthode, nous pourrions utiliser d'autres fonctions de score et de voisinage. Une possibilité serait de créer des suites exactes sur des graphes (cliques, modulo, etc.) et d'utiliser les groupes d'homologie [1] pour définir des fonctions de voisinage. Nous pourrions aussi utiliser l'article de C. McCreesh et P. Prosser. La partie graphe additionnel étant un cas particulier de notre méthode, nous pourrons exploiter les autres aspects de cet algorithme. Les auteurs ont remarqué que la contrainte *AllDiff* était particulièrement difficile et ont mis en place un propagateur spécial. Ceci nous permet de penser que nous pourrions améliorer notre algorithme en faisant progresser également cette partie.

Nous avons aussi développé dans cette thèse un algorithme de propagation basé sur l'algorithme MAC particulièrement efficace sur les problèmes de théorie des graphes. Nous pourrions adapter cette méthode au problème d'isomorphisme de sous-graphe. Ainsi, nous avons montré que dans l'algorithme MAC la longueur de la queue de propagation était très souvent plus courte lorsqu'une incohérence était détectée à l'issue du processus. De plus, nous avons utilisé une fonction de coût générique et nous pourrions la modifier de manière à prendre en compte des paramètres du graphe. En effet, les variables correspondent à des

1. L'homologie est une technique classique en mathématiques, voir [53]

sommets du graphe motif et leur apparition répétée dans la queue de propagation peut être due à une propriété de ce sommet dans le graphe.

Le temps de recherche d'un point fixe dans l'algorithme MAC est très variable d'un problème à l'autre. Les raisons qui expliquent cela sont encore assez méconnues. Une abstraction du problème peut permettre une analyse ce mécanisme et une comparaison avec les autres filtrages. Pour cela, il est possible de reformuler le problème un peu à la manière de ce qui a été fait au chapitre 1 et dans [76, 11, 3]. Un réseau de contraintes est la donnée de domaines $D_1, \ldots, D_n$ et de contraintes $C_1, \ldots, C_m$. Une contrainte peut être considérée (en extension) comme un sous-ensemble de $\mathrm{D} = D_1 \times \times D_n$ (voir [76, 11, 3]). Ainsi un problème de satisfaction de contraintes peut s'énoncer de la manière suivante :

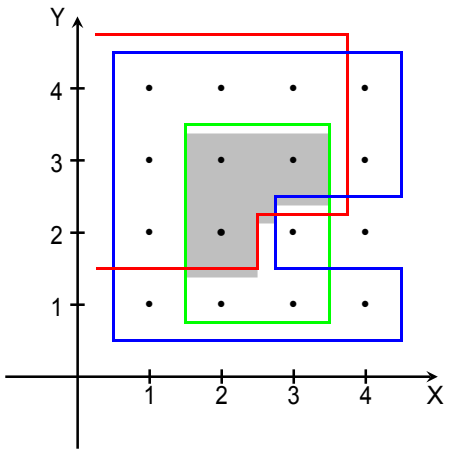
Considérons n sous-ensembles finis $D_1, \ldots, D_n$ de N et m sous-ensembles finis $C_1, \ldots, C_m$ de N^n, déterminer un élément (ou tous les éléments) de

$$\mathrm{D} \cap \mathrm{C}$$

où $\mathrm{D} = D_1 \times \times D_n$ et $\mathrm{C} = \cap_{i=1} C_i$.

Un solveur de contraintes détermine donc une intersection d'ensembles. Bien sûr, cela ne permet en aucun cas de simplifier le problème. Ces ensembles peuvent être tellement grands que cette intersection est impossible à déterminer à l'échelle humaine en testant tous les éléments de D. Un réseau de contraintes considère en permanence des domaines, donc l'ensemble courant est un produit cartésien de domaines donc un pavé. La solution sera rarement un pavé. Mais c'est cette forme qui permet aux solveurs de contraintes de résoudre des problèmes dans un temps raisonnable. En effet, cette forme permet de travailler en permanence sur les projections, ce qui réduit la taille des éléments à considérer. Le filtrage permet de supprimer des valeurs des domaines, ce qui correspond ici à des hyperplans. Dans l'exemple suivant $D_X = D_Y = \mathrm{J1,4K}$, ainsi $\mathrm{D} = \mathrm{J1,4K}^2$. Nous avons 3 contraintes représentées par les ensembles

$$\begin{aligned} C_1 &= \mathrm{D} \setminus \{(3,2),(4,2)\} \\ C_2 &= \mathrm{J2,4K} \times \mathrm{J2,4K} \setminus \{(3,2)\} \\ C_3 &= \mathrm{J2,3K} \times \mathrm{J1,3K}. \end{aligned}$$



L'ensemble des solutions est $\{(2,2),(2,3),(3,3)\}$.

Nous pouvons alors reformuler la propriété de cohérence d'arc dans ce contexte. Notons p_i la projection sur la $i^{\text{ème}}$ composante, c'est -à-dire, pour un réseau de contraintes de n variables,

$$\forall i \in \mathrm{J1},n\mathrm{K},\ p_i : \begin{pmatrix} \mathrm{N}^n & \longrightarrow & \mathrm{N} \\ (x_1, \ldots, x_n) & \longrightarrow & x_i \end{pmatrix}.$$

En reprenant les notations précédentes, un réseau est arc cohérent si $\forall i \in J1, nK,\ p_i(D) = \cap_j p_i(C_j \cap D)$. Avec les mêmes notations, le plus grand sous-réseau arc cohérent de D est le

point fixe de la suite récurrente $(D^{(n)})$ sur N^n définie par $D^{(0)} = \mathrm{D}$ et $\forall n \geq 0$, $D^{(n+1)} = f(D^{(n)})$ où $f : D \to \Pi_i \cap_j p_i(D \cap C_j)$.

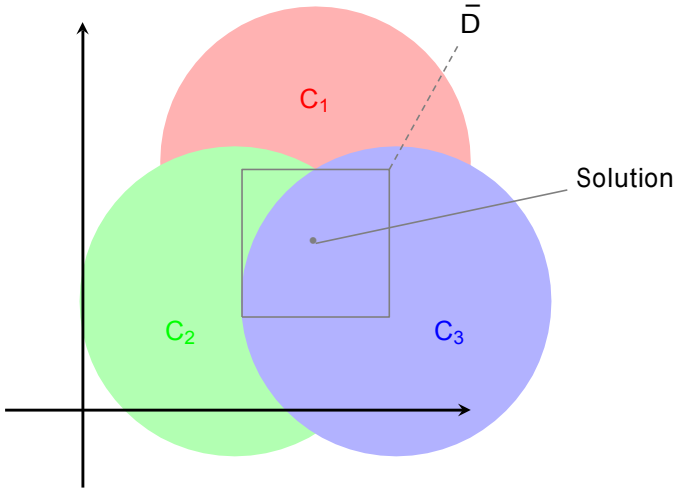
L'algorithme MAC est construit de manière un peu différente. Il est donné par une autre suite récurrente. Notons, $\forall i \in [\![1, n]\!]$ et $\forall j \in [\![1, m]\!]$, g_{ij} la fonction définie par

$$g_{ij} : \begin{cases} \mathrm{P}(\mathrm{N}^n) & \longrightarrow & \mathrm{P}(\mathrm{N}^n) \\ D & \longrightarrow & p_1(D) \times \times p_i(D \cap C_j) \times \times p_n(D) \end{cases}$$

Notons également $g = \Pi_{i=1}^{n} \Pi_{j=1}^{m} g_{ij}$. La suite récurrente définie par $E^{(0)} = \mathrm{D}$ et $\forall n \geq 0$, $E^{(n+1)} = g(E^{(n)})$ est convergente (lorsque $E^{(0)}$ est un produit cartésien) et sa limite correspond à l'état du réseau de contraintes après l'application de l'algorithme MAC. La limite est indépendante de l'ordre de la composition des fonctions g_{ij}. Une preuve est donnée dans [3], page 274. Notons D^M ce point fixe (qui est le même que celui de f).

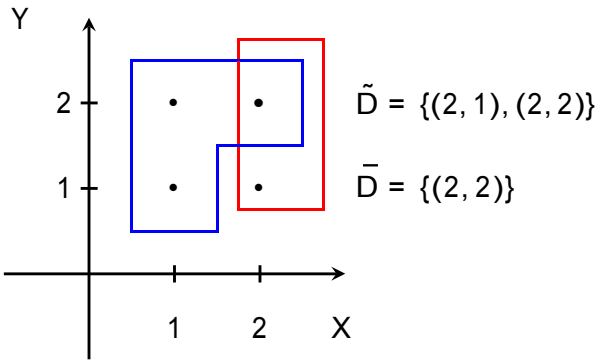
L'étude de D^M et de la fonction g peut être intéressante pour améliorer les solveurs. Si le réseau est généré aléatoirement, quelle est la probabilité d'obtenir un point fixe vide ? Nous avons déjà vu que le nombre d'étapes pour obtenir D^M dépendait de l'ordre des g_{ij}, peut-on trouver une heuristique qui minimise ce temps ? Aujourd'hui les nouvelles méthodes sont testées sur des solveurs contenant une méthode de filtrage par défaut qui peut influencer les résultats obtenus. Il serait intéressant d'avoir un solveur générique basé sur l'algorithme MAC dans lequel nous pourrions implémenter nos méthodes en comprenant l'influence de l'algorithme MAC et ainsi plus facilement comparer les approches. Il est également possible de réaliser des encadrements de ce point fixe.

L'ensemble $\Pi_{i=1}^{n} p_i(\mathrm{C} \cap \mathrm{D})$ est le réseau le plus proche de la solution. Sur ce réseau il n'est plus possible d'effectuer de filtrage. Notons $\bar{\mathrm{D}}$ l'ensemble associé.

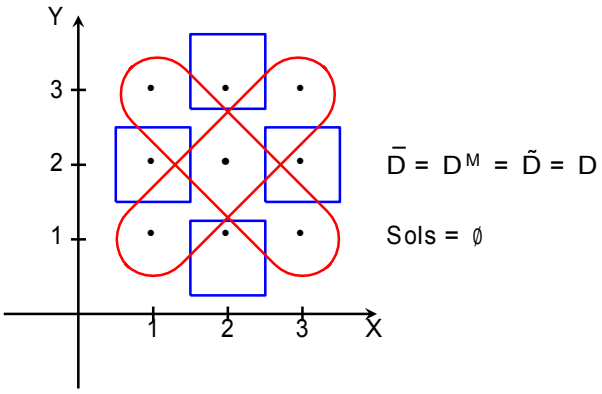


Nous avons clairement $\bar{\mathrm{D}} \subseteq \mathrm{D}^M$.

Notons $\tilde{\mathrm{D}}$ l'ensemble $\Pi_{i=1}^{n} \cap_{j=1}^{m} p_i(C_j) \cap p_i(\mathrm{D})$. Cet ensemble contient clairement D^M.



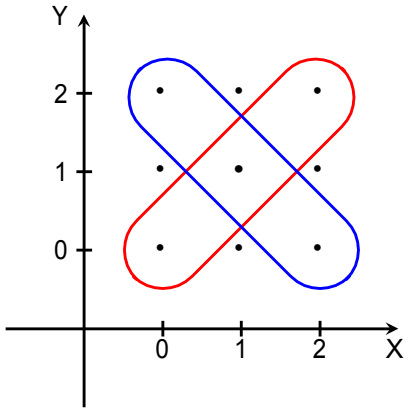
De même, le problème peut être sans solution mais avec un ensemble D^M non trivial.



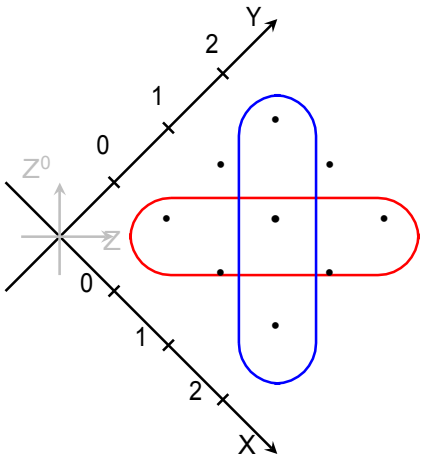
Nous voyons donc que pafois l'algorithme MAC ne permet plus de filtrer de valeur alors que géométriquement il semble encore possible de le faire moyennant une rotation. À titre d'exemple, prenons le réseau ayant pour variables X et Y, dont les domaines sont $D_X = D_Y = \{0, 2\}$ et dont les contraintes sont

$$\begin{aligned} Y &= X \\ Y &= -X + 2 \end{aligned}$$

Nous pouvons représenter cela par



Aucun filtrage n'est possible à cause de la forme de pavé que doivent avoir les réseaux de contraintes. Cependant, une rotation peut changer cela :



Le changement de variables $Z = X + Y$ et $Z^0 = Y - X$ permet de déduire le réseau de contraintes défini par : $D_Z = \{0, 4\}$ et $D_{Z^0} = \{-2, 2\}$, les contraintes deviennent

$$\begin{aligned} Z^0 &= 0 \\ Z &= 2. \end{aligned}$$

Le nouveau réseau a alors une solution évidente. Cela ne résout pas forcément le problème puisqu'il faudrait ensuite résoudre un système, mais cette idée se rapproche des reformulations de problèmes. Il peut donc être intéressant d'étudier ces changements de variables et ces transformations géométriques. En fonction du type des contraintes cela peut être rentable.

De manière générale, un réseau de contraintes est une structure qu'il serait intéressant de transporter en définissant des morphismes de réseaux de contraintes. Ces morphismes pourraient conserver toutes les solutions ou une partie.

Nous avons défini une méthode statistique pour déterminer des liens cachés entre les variables. Cette méthode comprend une collecte de données qui mesure l'impact d'une décision sur les domaines des autres variables. Elle comprend également une construction permettant d'obtenir un graphe de type graphe primal à partir de ces données. Nous avons mis en évidence dans certains problèmes insatisfiables, la présence de noyaux insatisfiables dans les composantes connexes du graphe obtenu après filtrage d'un certain nombre de liaisons. Les composantes connexes représentent ainsi des sous-problèmes insatisfiables. Nous avons également utilisé la collecte de ces données afin d'ajouter des contraintes tables au problème. Cette méthode se révèle efficace sur certains problèmes mais pas encore sur une majorité de problèmes. Beaucoup d'améliorations sont possibles. Par exemple, la construction des clusters peut être effectuée grâce à des algorithmes génériques de clustering.

Les réseaux de contraintes permettent de créer différents graphes représentant une projection du problème. Nous avons donc choisi une méthode statistique simple afin d'essayer d'entrevoir rapidement la structure du problème. D'autres méthodes qui utilisent la construction exposée sont à l'étude. Il est possible de pondérer le graphe ou de modifier les conditions permettant de relier deux sommets. Les heuristiques issues de ce graphe obtenu statistiquement semblent représenter également une piste intéressante. Par ailleurs, les différentes méthodes peuvent être combinées.

Enfin beaucoup de problèmes mathématiques se ramènent à la détermination d'un graphe sous différentes contraintes. Notamment les problèmes de pavage [120]. La génération de graphes peut se faire avec des réseaux de contraintes SAT ou CSP. Cela peut permettre de trouver une solution à un problème ou de renforcer une intuition comme cela a été le cas dans [60].

Bibliographie

[1] *Handbook of Econometrics*. Elsevier, 1983–2007.

[2] M.S. Affane and H. Bennaceur. A weighted arc-consistency technique for Max-CSP. In *Proceedings of ECAI'98*, pages 209–213, 1998.

[3] K.R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[4] James Bailey and Peter J. Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005, Proceedings*, pages 174–186, 2005.

[5] Nicolas Barnier and Pascal Brisset. Graph coloring for air traffic flow management. In *CPAIOR'02 : Fourth Internation Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems*, pages 133–147, Le Croisic, France, March 2002.

[6] R. Bartak and R. Erben. A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04*, pages 257–262, 2004.

[7] N. Beldiceanu, M. Carlsson, and J. Rampon. Global constraint catalog. Technical report, Swedish Institute of Computer Science, http://www.emn.fr/x-info/sdemasse/gccat/, 2005-2008.

[8] N. Beldiceanu, M. Carlsson, J. Rampon, and C. Truchet. Graph invariants as necessary conditions for global constraints. In *Proceedings of CP'05*, pages 92–106, 2005.

[9] Nicolas Beldiceanu and Evelyne Contejean. Introducing Global Constraints in CHIP. *Mathl. Comput. Modelling*, 20(12) :97–123, 1994.

[10] Claude Berge. *Graphes et hypergraphes*. Dunod université : mathématiques. Dunod, 1973 (Paris, Paris, Bruxelles, Montréal.

[11] C. Bessiere. Arc consistency and arc consistency again. *Artificial Intelligence*, 65 :179–190, 1994.

[12] C. Bessiere, S. Cardon, R. Debruyne, and C. Lecoutre. Efficient algorithms for singleton arc consistency. *Constraints*, 16(1) :25–53, 2011.

[13] C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of CP'04 workshop on constraint propagation and implementation*, pages 17–27, 2004.

[14] C. Bessiere and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59, 2005.

[15] C. Bessiere, E.C. Freuder, and J. Régin. Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107 :125–148, 1999.

[16] C. Bessiere, G. Katsirelos, N. Narodytska, CG. Quimper, and T Walsh. Decompositions of all different, global cardinality and related constraints. *IJCAI*, 2016.

[17] C. Bessiere, P. Meseguer, E.C. Freuder, and J. Larrosa. On Forward Checking for non-binary constraint satisfaction. *Artificial Intelligence*, 141 :205–224, 2002.

[18] C. Bessiere and J. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, 2001.

[19] Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, pages 29–83. 2006.

[20] Christian Bessiere. Constraint propagation. In *Handbook of Constraint Programming*, pages 29–83. 2006.

[21] Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems : structure and behaviour. *J. Exp. Theor. Artif. Intell.*, 15(1) :25–46, 2003.

[22] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *Proceedings of CPAI'04 workshop held with CP'04*, pages 29–43, 2004.

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

[24] P. Briggs and L. Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4) :59–69, 1993.

[25] H. Bunke. Recent developments in graph matching. In *Proceeding of ICPR'00*, volume 2, pages 117–124, 2000.

[26] A. Chmeiss and P. Jégou. Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2) :121–142, 1998.

[27] A. Chmeiss and L. Sais. Constraint satisfaction problems : Backtrack search revisited. In *Proceedings of ICTAI'04*, pages 252–257, 2004.

[28] Assef Chmeiss and Lakhdar Sais. De FC à MAC : un algorithme paramétrable pour la résolution des CSP. In Christine Solnon, editor, *Premières Journées Francophones de Programmation par Contraintes*, Premières Journées Francophones de Program-mation par Contraintes, pages 267–276, Lens, June 2005. CRIL - CNRS FRE 2499, Université d'Artois. http ://www710.univ-lyon1.fr/ csolnon.

[29] Olivier Cogis and Claudine Robert. *Théorie des graphes : au-delà des ponts de Königsberg : problèmes, théorèmes, algorithmes*. Vuibert, Paris, 2003. Autres tirages : 2e tirage revu et corrigé en octobre 2003, 2005.

[30] Hélène Collavizza, François Delobel, and Michel Rueher. A note on partial consistencies over continuous domains. In *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, pages 147–161, 1998.

[31] D. Conte, P. Foggia, C. Sansone, and M. Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3) :265–298, 2004.

[32] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted CSP. In *Proceedings of AAAI'08*, pages 253–258, 2008.

[33] Martin C. Cooper. An optimal k-consistency algorithm. *Artif. Intell.*, 41(1) :89–95, 1989.

[34] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the VF graph matching algorithm. In *Proceedings of ICIAP'99*, pages 1172–1177, 1999.

[35] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 26(10) :1367–1372, 2004.

[36] Thomas H. Cormen, Charles Eric Leiserson, Ronald L. Rivest, and Xavier Cazin. *Introduction à l'algorithmique*. Science informatique. Dunod, Paris, 1994. Trad. de : Introduction to algorithms.

[37] D.G. Corneil and C.C. Gotlieb. An efficient algorithm for graph isomorphism. *Journal of the ACM*, 17(1) :51–64, 1970.

[38] G.B. Dantzig and Stanford University. Systems Optimization Laboratory. *Origins of the Simplex Method*. Technical report (Stanford University. Systems Optimization Laboratory). Defense Technical Information Center, 1987.

[39] M. Davis and H. Putnam. Computational methods in the propo- sitional calculus. Rensselaer Polytechnic In- stitute, 1958.

[40] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7) :394–397, 1962.

[41] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

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

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

[44] Reinhard Diestel. *Graph theory*. Graduate texts in mathematics. Springer, Berlin, 2005. cop. 2006 pour l'édition brochée.

[45] R. Faure, B. Lemaire, and C. Picouleau. *Précis de recherche opérationnelle - 6e éd. : Méthodes et exercices d'application*. Mathématiques. Dunod, 2009.

[46] J-C. Fournier. *Théorie des graphes et applications*. Collection Informatique. Lavoisier, 2011.

[47] Eugene C. Freuder. Synthesizing constraint expressions. *Commun. ACM*, 21(11) :958–966, 1978.

[48] Eugene C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1) :24–32, 1982.

[49] M. Gardner. *Logic machines and diagrams*. McGraw-Hill, 1958.

[50] J. Gaschnig. A constraint satisfaction method for inference making. In *Proceedings of the 12th Annual Allerton Conference on Circuit and System Theory*, pages 866–874, 1974.

[51] I.P. Gent, C. Jefferson, and I. Miguel. Minion : A fast, scalable constraint solver. In *Proceedings of ECAI'06*, pages 98–102, 2006.

[52] Alan M. Gibbons. *Algorithmic graph theory*. Cambridge University Press, Cambridge, New York, Oakleigh, 1985. Autres tirages : 1987, 1988, 1989, 1991, 1994.

[53] R. Godement. *Topologie algébrique et théorie des faisceaux*. Number vol. 1 in Ac-tualités scientifiques et industrielles. Hermann, 1958.

[54] Eugene Goldberg and Yakov Novikov. Berkmin : A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12) :1549 – 1561, 2007. {SAT} 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[55] C. Gomes and D. Shmoys. Completing quasigroups or latin squares : a structured graph coloring problem. In *Proceedings of Computational Symposium on Graph Coloring and Generalization*, 2002.

[56] Carla P. Gomes, Bart Selman, and Henry A. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, pages 431–437, 1998.

[57] Mattias Grönkvist. A constraint programming model for tail assignment. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, First International Conference, CPAIOR 2004, Nice, France, April 20-22, 2004, Proceedings*, pages 142–156, 2004.

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

[59] Emmanuel Hebrard, Eoin O'Mahony, and Barry O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, pages 181–185, 2010.

[60] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, pages 228–245, 2016.

[61] Université Monpellier II, editor. *Développement d'outils algorithmiques pour l'intel-ligence artificielle. Application à la chimie organique.* PhD thesis, 1995.

[62] M. Kaufmann. *Des points des flèches*. Science-poche.

[63] D.E. Knuth. *The Stanford GraphBase : A Platform for Combinatorial Computing*. ACM Press, 1993.

[64] Martin Kutz, Khaled M. Elbassioni, Irit Katriel, and Meena Mahajan. Simultaneous matchings : Hardness and approximation. *J. Comput. Syst. Sci.*, 74(5) :884–897, 2008.

[65] J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.

[66] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical Structures in Computer Science*, 12(4) :403–422, 2002.

[67] Vianney Le clément de saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, sep 2013.

[68] C. Lecoutre. Benchmarks of constraint networks in XCSP format. Technical report, http ://www.cril.univ-artois.fr/ lecoutre/research/benchmarks/benchmarks.html 2007.

[69] C. Lecoutre. *Constraint networks : techniques and algorithms*. ISTE/Wiley, 2009.

[70] C. Lecoutre. STR2 : Optimized simple tabular reduction for table constraints. *Constraints*, 16(4) :341–371, 2011.

[71] C. Lecoutre and S. Cardon. A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204, 2005.

[72] C. Lecoutre and F. Hemery. A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130, 2007.

[73] C. Lecoutre and J. Vion. Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2 :21–35, 2008.

[74] Matthew D. T. Lewis, Tobias Schubert, and Bernd W. Becker. *Speedup Techniques Utilized in Modern SAT Solvers*, pages 437–443. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[75] Inês Lynce and João P. Marques Silva. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

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

[77] A. Malapert and C. Lecoutre. A propos de la bibliotheque de modeles xcsp. *JFPC'14*, pages 337–340, 2014.

[78] João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On computing minimal correction subsets. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 615–622, 2013.

[79] Ciaran McCreesh and Patrick Prosser. *A Parallel, Backjumping Subgraph Isomorphism Algorithm Using Supplemental Graphs*, pages 295–312. Springer International Publishing, Cham, 2015.

[80] J.J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, 19 :229–250, 1979.

[81] D. Mehta and M.R.C. van Dongen. Probabilistic consistency boosts MAC and SAC. In *Proceedings of IJCAI'07*, pages 143–148, 2007.

[82] Deepak Mehta and Marc R. C. van Dongen. Probabilistic consistency boosts MAC and SAC. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 143–148, 2007.

[83] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[84] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535, 2001.

[85] B.A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. *Search in Artificial Intelligence*, pages 287–342, 1988.

[86] S. Ndojh Ndiaye and C. Solnon. CP models for maximum common subgraph problems. In *Proceedings of CP'11*, pages 637–644, 2011.

[87] Huy Xuong Nguyen. *Mathématiques discrètes et informatique*. Logique mathématiques informatique. Masson, 1992 (Impr. en Belgique, 1991), Paris, Milan, Barcelone.

[88] W.J. Older, G.M. Swinkels, and M.H. van Emden. Getting to the real problem : experience with bnr prolog in or. In *Proceedings of the Third Conference on Practical Applications of Prolog*, 1995.

[89] D. C. Porumbel. Isomorphism testing via polynomial-time graph extensions. *Journal of Mathematical Modelling and Algorithms*, 10(2) :119–143, 2011.

[90] J.F. Puget. The next challenge for CP : Ease of use. Invited Talk at CP-2004, 2004.

[91] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *aaai94p*, pages 362–367, aaai94l, 1994.

[92] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367, 1994.

[93] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1.*, pages 209–215, 1996.

[94] Jean-Charles Régin. Using constraint programming to solve the maximum clique problem. In *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, pages 634–648, 2003.

[95] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[96] O. Roussel and C. Lecoutre. XML representation of constraint networks : Format XCSP 2.1. Technical Report arXiv :0902.2362, CoRR, 2009.

[97] Jean de Rumeur. *Communications dans les réseaux de processeurs*. Études et recherches en informatique. Masson, cop. 1994 (Impr. en Belgique), Paris, Milan, Barcelone.

[98] Lawrence Ryan. Efficient algorithms for clause-learning sat solvers, 2004.

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

[100] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI*, pages 125–129, 1994.

[101] Daniel Sabin and Eugene C. Freuder. Understanding and improving the MAC algorithm. In *Principles and Practice of Constraint Programming - CP97, Third International Conference, Linz, Austria, October 29 - November 1, 1997, Proceedings*, pages 167–181, 1997.

[102] A. Sache. *La théorie des graphes*. Que sais-je ? Presses Universitaires de France, 1974.

[103] M. Samy Modeliar. *Mathématiques du DUT informatique*. Références sciences. Ellipses Marketing, 2015.

[104] M. De Santo, P. Foggia, C. Sansone, and M. Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recognition Letters*, 24(8) :1067–1079, 2003.

[105] T. Schiex, J.C. Régin, C. Gaspin, and G. Verfaillie. Lazy arc consistency. In *Proceedings of AAAI'96*, pages 216–221, 1996.

[106] Christian Schulte and Peter J. Stuckey. When do bounds and domain propagation lead to the same search space ? In *Proceedings of the 3rd international ACM SIGPLAN conference on Principles and practice of declarative programming, September 5-7, 2001, Florence, Italy*, pages 115–126, 2001.

[107] João P. Marques Silva and Karem A. Sakallah. Grasp&mdash ;a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[108] C. Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13) :850–864, 2010.

[109] K. Stergiou. Heuristics for dynamically adapting propagation. In *Proceedings of ECAI'08*, pages 485–489, 2008.

[110] Edward P. K. Tsang, John A. Ford, Patrick Mills, Richard Bradwell, Richard Williams, and Paul Scott. Towards a practical engineering tool for rostering. *Annals OR*, 155(1) :257–277, 2007.

[111] J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1) :31–42, 1976.

[112] M.R.C. van Dongen. Lightweight arc-consistency algorithms. Technical Report TR-01-2003, University college Cork, 2003.

[113] P. van Hentenryck, Y. Deville, and C.M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321, 1992.

[114] P. van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3) :139–164, 1998.

[115] W.J. van Hoeve. The alldifferent constraint : a survey. In *Proceedings of the Sixth Annual Workshop of the ERCIM Working Group on Constraints*, 2001.

[116] W.J. van Hoeve and I. Katriel. Global constraints. In *Handbook of Constraint Programming*, chapter 6, pages 169–208. Elsevier, 2006.

[117] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of AI/GI/VI'92*, pages 163–169, 1992.

[118] D.J. Watts and S.H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393 :440–442, 1998.

[119] B. Weisfeiler and A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, ser. 2(9), 1968.

[120] Martin Saralegi y J. I. Royo Prieto. *Euler y un balón de fútbol*, volume 36. SIGMA, Servicio Central de Publicaciones del Gobierno Vasco, 2010.

[121] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15(3) :327–353, 2010.

[122] M. Zytnicki, C. Gaspin, and T. Schiex. A new local consistency for weighted CSP dedicated to long domains. In *Proceedings of SAC'06*, pages 394–398, 2006.

Index

A

all different, 5
apprentissage, 9
arc cohérente, 12
arité, 4

B

branchement, 8

C

carré magique, 1
clique, 45
cohérence d'arc en *n* passes, 66
contrainte, 4
  binaire, 4
  non-binaire, 4
  ternaire, 4
  unaire, 4
  violée, 8
contrainte en extension, 4
contrainte en intention, 4
couplage, 51
  couvrant, 51
  maximum, 51

D

D-cohérence aux bornes, 15, 37
delayedConstruction, 32
diagramme de classe, 29
digraphe, 41

F

FC Edges, 48
FilterFrom, 32, 34
filtrage, 10
fonction
  de voisinage, xvi
fonction
  de score, xvi

G

GAC Edges, 48
graphe, 41
  biparti, 51
  cible, 44
  dual, 81
  non orienté, 41
  primal, 81
  simple, 41
  source, 44

H

heuristique de choix de valeurs, 10
heuristique de choix de variables, 10
hypergraphe
  de compatibilité, 81
  de contraintes, 81

I

ILF, 49
instanciation, 8
  complète, 8

L

LAD, 52
limite d'un sparse set, 24
LV2002, 48

M

matrice
  de poids, 83
  d'adjacence, 53
morphisme de graphe, 44
  injectif, 44
multi-ensemble, 49

N

niveau, 8

P

portée, 4
problème
  d'isomorphisme de sous-graphe, 41
propagation, 8

R

R-filtrage, 65
réseaux de contraintes
  D-cohérent, 16
  Z-cohérent, 17
réseaux de contraintes, 5
  binaires, 5
retour arrière, 9

S

scope, 4
SND, 52
solution d'un réseaux de contraintes, 6
sommet
  voisin, 48
sous-réseau, 6
  induit, 7
sous-réseau réduit, 7 sparse set, 23

T

tuple autorisé, 4

V

variable, 3

Z

Z-cohérence aux bornes, 17, 37