UNIVERSITÉ D'ARTOIS

Intégration de techniques CSP pour la résolution du problème WCSP

THÈSE

présentée et soutenue publiquement le 6 novembre 2014

en vue de l'obtention du titre de

Docteur en Sciences

de l'Université d'Artois - Faculté des Sciences Jean Perrin

(Spécialité Informatique)

par

Nicolas Paris

Composition du jury

Rapporteurs :	De Givry Simon	INRA, Toulouse
	Jegou Philippe	Université d'Aix-Marseille
Examinateurs :	Deville Yves	Université Catholique de Louvain (Belgique)
	Lecoutre Christophe	Université d'Artois
	Roussel Olivier	Université d'Artois
	Tabary Sébastien	Université d'Artois

Centre de Recherche en Informatique de Lens – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.fr

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

Remerciements

Tout d'abord, je remercie les différents membres du jury et plus particulièrement Simon de Givry et Philippe Jégou qui ont accepté de rapporter cette thèse. Merci également à Yves Deville pour avoir accepté de participer au jury de cette thèse.

Naturellement, je tiens à remercier Christophe Lecoutre, mon directeur de thèse, ainsi qu'Olivier Roussel et Sébastien Tabary, mes co-encadrants. Au delà de m'avoir accordé leur confiance, ils m'ont fait découvrir durant ces trois années le monde de la recherche à travers leur rigueur scientifique et leur passion. Tout en me laissant l'autonomie nécessaire, ils ont su me guider durant toute cette expérience pour me permettre d'acquérir les qualités attendues d'un chercheur. Par mon travail, j'espère leur avoir rendu cette confiance et avoir atteint cet objectif.

Je remercie également Éric Grégoire, directeur du CRIL, de m'avoir accueilli au sein de son laboratoire. Je m'y suis réellement épanoui grâce aux outils mis à ma disposition, à l'organisation régulière de séminaires et à la participation à des conférences qui m'ont fait voyager à travers le monde et qui m'ont permis de développer ma culture scientifique. Ma reconnaissance va aussi aux chercheurs, aux enseignants-chercheurs et aux ingénieurs du CRIL qui ont toujours été disponibles pour discuter, partager leurs connaissances et leurs expériences, débattre sur des sujets divers et variés. Merci également aux personnels administratifs et techniques du CRIL qui m'ont permis de travailler dans des conditions idéales et de me concentrer principalement sur mon travail de recherche. Merci également à tous mes compagnons doctorants du CRIL, avec une pensée plus particulière pour Benoît Hoessen et Vincent Perradin, mes voisins de bureau, pour leur bonne humeur et leur complicité quotidienne. Je ne peux qu'être redevable à toutes ces personnes qui ont fait que cette thèse restera pour moi un très bon souvenir.

Je n'oublie pas non plus de remercier les différents partenaires du projet industriel PAJERO qui a permis de financer et de mener cette thèse à son terme : l'organisme BPI FRANCE (anciennement OSEO) pour son programme stratégique d'innovation des entreprises, ainsi que les laboratoires I3S, PRiSM et les entreprises Equitime, HSW, et CAPS.

Pour finir, un grand merci à Céline Gravier pour son soutien sans faille et sa patience durant ces trois années, et bien sûr à tous les membres de ma famille toujours compréhensifs et sans qui rien de tout ça n'aurait été possible. Je n'oublie pas non plus mes amis d'enfance et mes anciens collègues qui m'ont soutenu durant cette aventure et qui se reconnaîtront.

LA LIBERTÉ, C'EST LA FACULTÉ DE CHOISIR SES CONTRAINTES.

Jean-Louis Barrault - acteur, metteur en scène et directeur de théâtre - 1910-1994

Table des matières

Partie I État de l'art

Table des figures

Introduction générale

1 Le contexte de travail

Dans le domaine de l'intelligence artificielle, la programmation par contraintes (*CP* pour *Constraint Programming*) est un paradigme permettant de traiter entre autres les problèmes combinatoires de nature décisionnelle. Résoudre un problème de décision, c'est répondre "oui" ou "non" à la question qui est de savoir si une solution existe ou pas pour ce problème. Apparue dans les années 1970, l'approche CP consiste à modéliser certains de ces problèmes en réseaux de contraintes (*CN* pour *Constraint Network*) : ils sont définis par un ensemble de variables, représentant des inconnues, et un ensemble de contraintes représentant certaines limitations sur les inconnues. Le problème résultant est appelé *problème de satisfaction de contraintes* (*CSP* pour *Constraint Satisfaction Problem*) et une instance de ce problème, obtenue en spécifiant précisément les variables et les contraintes, est résolue par un programme appelé *solveur*. Une instance est satisfaisable si elle admet au moins une solution, c'est à dire une affectation de valeur à chaque variable permettant de satisfaire toutes les contraintes, sinon elle est insatisfaisable. Un solveur (complet) fournit une solution à toute instance CSP si elle existe et si on lui donne suffisamment de temps (potentiellement infini). Le problème de satisfaction de contraintes est un problème fortement combinatoire dans le sens où trouver une solution, ou prouver qu'il n'en existe aucune, nécessite généralement de considérer un nombre exponentiel de combinaisons pour y parvenir. La progression de la puissance des ordinateurs ne peut rien y changer. Pour réduire cette combinatoire, la programmation par contraintes propose un ensemble de techniques d'intelligence artificielle : des heuristiques, de l'inférence pour filtrer l'espace de recherche, des méthodes d'apprentissage, etc. Des instances CSP peuvent alors être résolues efficacement en pratique malgré leur forte complexité théorique. De plus, de nombreux problèmes décisionnels complexes réels de type "planification" (chaîne de montage en production, etc.), "ordonnancement" (séquençage de génome en biologie moléculaire, etc.), ou "allocation/affectation de ressources" (emploi du temps, etc.) s'intègrent naturellement au cadre de la programmation par contraintes.

Considérons par exemple le problème consistant à établir un emploi du temps, et plus précisément une instance d'emploi du temps scolaire introduite dans la figure 1. L'objectif est de proposer un emploi du temps permettant à chaque formation (F1, F2, etc.) de suivre les heures de cours prévues dans son programme pour chaque matière (mathématiques, informatique, etc.) et inculquées par différents enseignants (E1, E2, etc.) dans plusieurs salles à disposition dans l'école (salle 1, salle 2, etc.). La figure 1 illustre une solution partielle d'emploi du temps ciblée sur la formation F2 : on y voit par exemple que cette formation a cours d'informatique le lundi matin de 10h à 12h avec l'enseignant E1 dans la salle 3, puis un cours de mathématiques le mercredi après-midi de 14h à 16h avec l'enseignant E2 dans la salle 1, etc. À première vue, proposer un tel emploi du temps semble facile et on imagine que le nombre de solutions possibles est important. Cependant, certaines contraintes dites "*dures*" doivent être satisfaites par ces propositions pour que l'emploi du temps soit réalisable (acceptable) : un enseignant ne peut pas donner deux cours différents durant le même créneau horaire, des travaux pratiques d'informatique doivent se tenir dans une salle équipée d'ordinateurs, etc. À partir de la figure 2, on peut imaginer une

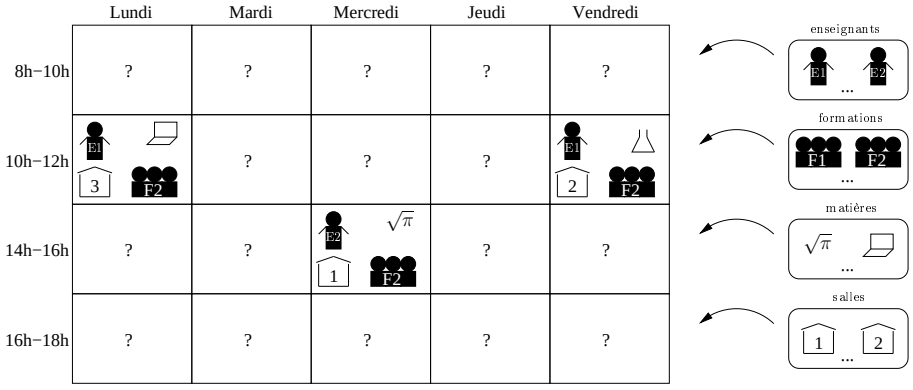


FIG. 1 – Solution partielle d'une instance d'emploi du temps scolaire.

modélisation possible (basique) du problème d'emploi du temps. Pour chaque couple "créneau horaire - salle", nous allons définir trois variables : une variable pour l'enseignant, une variable pour la matière et une variable pour la formation. Une fois toutes ces variables assignées, nous aurons ainsi un enseignant, une matière et une formation désignés pour chaque couple "créneau horaire - salle". Cependant, n'oublions pas les contraintes (rappelées par des flèches sur la figure 2) : nous définissons donc entre autres des contraintes portant sur les variables prévues pour les affectations de matières (une salle doit être adaptée à une matière), ainsi que sur les variables prévues pour les affectations d'enseignants (un seul cours à la fois). Résoudre cette instance CSP correspond donc à proposer un emploi du temps résultant de l'affectation dans chaque créneau horaire et pour chaque salle d'un enseignant, d'une matière et d'une formation de telle manière que toutes les contraintes soient satisfaites.

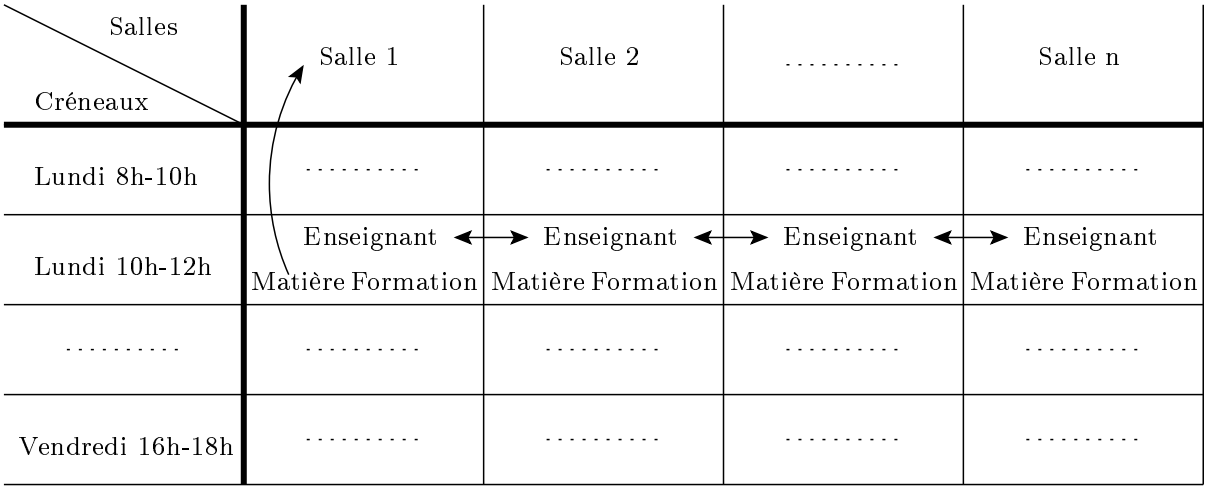


FIG. 2 – Modélisation CSP du problème d'affectation d'un emploi du temps scolaire.

Pour l'emploi du temps, nous avons également besoin de contraintes dites "*souples*". Ces contraintes sont souples car elles peuvent ne pas être satisfaites par une proposition d'emploi du temps et garantir malgré tout que l'emploi du temps proposé soit réalisable. Cependant, ces contraintes permettent d'exprimer des *préférences* et de leur satisfaction ou non dépendra alors la *préférence* d'une proposition d'emploi du temps par rapport à une autre. Dans notre emploi du temps scolaire, une contrainte souple correspond par exemple à éviter les créneaux horaires creux (un créneau horaire vide, c'est à dire dans lequel il n'y a pas de cours à donner, entouré de deux créneaux pleins) dans une même journée pour un

enseignant : une proposition d'emploi du temps pourra donc inclure ces créneaux creux et être réalisable, mais celle qui les minimisera sera préférée. Au carrefour de l'intelligence artificielle et de la recherche opérationnelle, la programmation par contraintes a introduit au début des années 2000 un cadre permettant de traiter des problèmes avec des contraintes dures et des contraintes souples : il s'agit du problème de satisfaction de contraintes pondérées (*WCSP* pour *Weighted Constraint Satisfaction Problem*). Par opposition aux contraintes dures présentées précédemment (qui autorisent ou interdisent des affectations de valeurs aux variables), on va donc autoriser la violation des contraintes souples. Cette souplesse se traduit au travers de *degrés de pénalités* ou *coûts* attribués aux différentes affectations (de valeurs) dans chacune des contraintes. Un *coût interdit* est défini pour les contraintes souples d'un réseau de contraintes pondérées (*WCN* pour *Weighted Constraint Network*) et permet malgré la souplesse introduite de continuer à refuser des affectations irréalisables (inacceptables). Une affectation est alors soit totalement autorisée (coût de 0), soit totalement interdite (coût interdit), soit autorisée avec le coût correspondant qui lui est attribué par la contrainte. Un exemple d'attribution de coûts au sein de la contrainte souple consistant à éviter les créneaux creux pour un enseignant dans une journée (dans le problème d'emploi du temps) est proposé dans la figure 3.

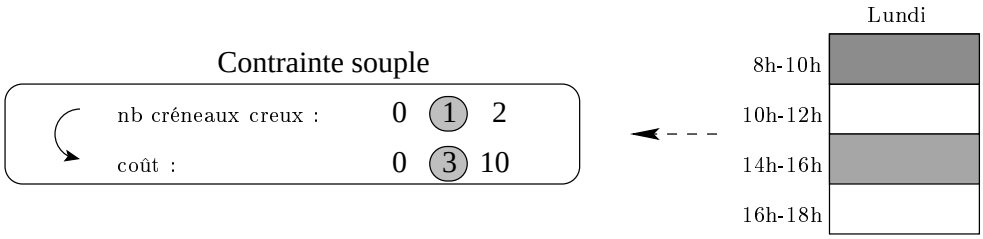


FIG. 3 – Coûts attribués par la contrainte souple.

Un coût de 0 est attribué s'il n'y a aucun créneau creux de prévu durant la journée, un coût de 3 est attribué si un créneau creux est prévu, un coût de 10 est attribué pour deux créneaux creux. On distingue clairement que plus le nombre de créneaux creux prévus durant une journée est conséquent pour un enseignant, plus le coût est important dans la contrainte souple et moins l'emploi du temps correspondant sera préféré. La solution d'un WCSP correspond alors à une affectation de valeur à chaque variable satisfaisant complètement les contraintes dures et minimisant les coûts dans les contraintes souples. Il ne s'agit plus simplement ici d'un problème de décision, il s'agit d'un problème d'optimisation : non seulement une solution doit être trouvée, mais en plus celle-ci doit être optimale, c'est à dire avec le coût le plus faible possible au niveau des contraintes souples. Le cadre WCSP et les différentes techniques de résolution qu'il propose permettent de résoudre ces problèmes.

2 Problématiques abordées et contributions apportées

L'efficacité des techniques proposées dans le cadre CSP est avérée et ne cesse de s'améliorer depuis des décennies. Les techniques dans le cadre WCSP sont plus complexes à cause des opérations de transferts de coût et d'un point de vue plus général parce qu'il s'agit de problèmes d'optimisation. La plupart d'entre elles ont été définies et/ou utilisées pour les réseaux de contraintes binaires ou ternaires pour lesquelles elles sont assez efficaces. Cependant, elles peuvent amener à une explosion combinatoire dans le cadre de contraintes de grande arité. L'idée générale soutenue dans cette thèse est que l'intégration de techniques proposées dans le cadre CSP peut permettre d'améliorer l'efficacité dans la résolution d'instances WCSP.

La première partie de ce mémoire trace un état de l'art de la programmation par contraintes, et plus précisément des cadres CSP et WCSP, qui représente réellement le contexte de nos différentes contributions. La seconde partie aborde nos deux contributions. Nous proposons dans notre première contribution un algorithme pour maintenir la cohérence d'arc souple généralisée GAC* et qui soit optimisé pour les contraintes tables souples de grande arité. Cette contribution se compose d'une première version de travail ayant été publiée dans [Lecoutre *et al.*, 2012] et étendue par une contribution en cours de soumission. Nous proposons ensuite dans notre deuxième contribution une approche de résolution pour les instances WCSP basée sur l'extraction de noyaux insatisfaisables minimaux, se composant d'une partie ayant été publiée dans [Lecoutre *et al.*, 2013] et étendue par des travaux n'ayant pas été soumis.

Première partie

État de l’art

Chapitre 1

Le problème de satisfaction de contraintes CSP

Sommaire

1.1 Réseaux de contraintes

1.1.1 Définitions et notations

Avant d'introduire le concept de réseau de contraintes et le problème de satisfaction de contraintes associé, il convient dans un premier temps de définir les composants au cœur de ce cadre, à savoir, variables et contraintes.

Définition 1 (Variable) *Une variable est une entité à laquelle il est possible d'affecter une valeur. À chaque variable est associé un domaine : il s'agit de l'ensemble des valeurs distinctes pouvant être affectées à cette variable. Une variable est discrète (resp. continue) si elle est associée à un domaine discret*[1] *(resp. continu). Une variable est soit assignée (une valeur lui a été affectée), soit libre (aucune valeur ne lui a été affectée). Une variable dont le domaine ne contient qu'une seule valeur est dite singleton.*

[1] Dans ce manuscrit, nous considérons uniquement les domaines finis

Nous nous sommes intéressés dans le cadre de notre travail aux variables discrètes. Nous notons $dom(x)$ le domaine fini de valeurs associé à une variable (discrète) x. Sans perte de généralité, pour les différents exemples présentés dans ce manuscrit, des lettres de l'alphabet seront utilisées (sauf indication contraire) pour désigner les valeurs contenues dans les domaines des variables : par exemple, la variable x telle que $dom(x) = \{a, b\}$ est une variable dont le domaine contient les valeurs a et b. Nous verrons également par la suite que les domaines des variables évoluent par la suppression de valeurs durant la résolution d'un réseau de contraintes. Nous notons alors $dom^{init}(x)$ le domaine initial de la variable x, c'est à dire le domaine constitué de toutes les valeurs pouvant être affectées initialement à x. La valeur a appartenant à $dom^{init}(x)$ est notée aussi (x, a). Le domaine courant de la variable x, c'est à dire le domaine constitué des valeurs pouvant être affectées à x à un instant donné de la résolution, est noté $dom(x)$ et il respecte $dom(x) \subseteq dom^{init}(x)$. Affecter une valeur $a \in dom(x)$ à x consiste à restreindre $dom(x)$ à $\{a\}$: la variable x est alors assignée (par la valeur a) et on note $x = a$. Une valeur est valide pour une variable si elle appartient au domaine courant de cette variable. La taille de $dom(x)$, représentée généralement par $|dom(x)|$ (cardinalité de l'ensemble $dom(x)$), correspond au nombre de valeurs appartenant à $dom(x)$.

Exemple 1 *Soient deux variables x et y avec $dom^{init}(x) = dom^{init}(y) = \{a, b, c\}$, $dom(x) = \{a, b\}$ et $dom(y) = \{c\}$. Les domaines initiaux des variables x et y contiennent les valeurs a, b et c. Le domaine courant de la variable x ne contient plus que les valeurs a et b. La variable y est singleton et son domaine courant contient uniquement la valeur c.*

Définition 2 (Instanciation) *Une instanciation I d'un ensemble $X = \{x_1, ..., x_q\}$ de q variables est un ensemble $\{(x_1, v_1), ..., (x_q, v_q)\}$ tel que $\forall 1 \leq i \leq q, v_i \in dom^{init}(x_i)$ et la variable x_i n'apparaît qu'une seule fois.*

Nous notons $vars(I)$ l'ensemble des variables dans une instanciation I. Soit $x \in vars(I)$: la valeur associée à la variable x dans I est notée $I[x]$. Soit X un ensemble de variables : une instanciation I est dite complète sur X si $\forall x \in X, x \in vars(I)$, sinon elle est dite partielle. Une instanciation I est valide si et seulement si $\forall x \in vars(I), I[x] \in dom(x)$.

Exemple 2 *Soit un ensemble de variables $X = \{x, y, z\}$ avec $dom^{init}(x) = dom^{init}(y) = dom^{init}(z) = \{a, b, c\}$.*

- *$I = \{(x, a)\}$ est une instanciation partielle sur X avec $vars(I) = \{x\}$.*
- *$I = \{(x, a), (y, c)\}$ est une instanciation partielle sur X avec $vars(I) = \{x, y\}$ et $I[y] = c$.*
- *$I = \{(x, a), (y, c), (z, b)\}$ est une instanciation complète sur X avec $vars(I) = \{x, y, z\}$.*

Au delà des variables, les contraintes représentent bien sûr l'autre partie fondamentale au cœur des réseaux de contraintes. Avant de définir ce concept, il est utile d'introduire avant tout les notions générales de produit cartésien et de relation, puis de les appliquer au cadre des domaines associés aux variables.

Définition 3 (Produit cartésien) *Soient $D_1, D_2, ..., D_q$ des ensembles finis d'entiers. Le produit cartésien $D_1 \times D_2 \times ... \times D_q$, noté aussi $\prod_{i=1}^{q} D_i$, correspond à l'ensemble $\{(v_1, \ldots, v_q) \mid \forall 1 \leq i \leq q, v_i \in D_i\}$.*

Définition 4 (Relation) *Une relation définie sur une séquence $D_1, D_2, ..., D_q$ est un sous-ensemble du produit cartésien $\prod_{i=1}^{q} D_i$.*

Dans le cadre d'un réseau de contraintes, nous allons parler de produit cartésien des domaines de variables. Soient $dom(x_1), dom(x_2), ..., dom(x_q)$ les domaines respectifs de q variables $x_1, x_2, ..., x_q$:

le produit cartésien $dom(x_1) \times dom(x_2) \times ... \times dom(x_q)$, noté aussi $\prod_{i=1}^{q} dom(x_i)$, correspond à l'ensemble $\{(v_1, \ldots, v_q) \mid \forall 1 \leq i \leq q, v_i \in dom(x_i)\}$. Une relation définie sur $\prod_{i=1}^{q} dom(x_i)$ est donc un sous-ensemble de $\prod_{i=1}^{q} dom(x_i)$.

Exemple 3 *Soient trois variables x, y et z avec $dom(x) = \{a, b\}$, $dom(y) = \{a, c\}$ et $dom(z) = \{b, c\}$. Le produit cartésien des domaines de ces trois variables correspond à :*

$$dom(x) \times dom(y) \times dom(z) = \left\{ \begin{array}{l} (a,a,b) \\ (a,a,c) \\ (a,c,b) \\ (a,c,c) \\ (b,a,b) \\ (b,a,c) \\ (b,c,b) \\ (b,c,c) \end{array} \right\}$$

Un exemple de relation sur le produit cartésien des domaines de ces trois variables est donné par :

$$\left\{ \begin{array}{l} (a,a,b) \\ (a,c,b) \\ (a,c,c) \\ (b,a,b) \end{array} \right\}$$

Nous abordons maintenant le concept de contrainte. À noter que dans le cadre des réseaux de contraintes, il s'agit de contraintes dures contrairement aux contraintes souples introduites dans le chapitre 2.

Définition 5 (Contrainte) *Une contrainte (dure) c porte sur un ensemble de variables (totalement ordonné) appelé portée (ou scope) et noté $scp(c)$. Toute variable de $scp(c)$ est dite impliquée dans c. Elle est définie par une relation notée $rel(c)$ qui autorise (et indirectement interdit) les valeurs que peuvent prendre simultanément (dans leurs domaines) les variables appartenant à sa portée et qui vérifie $rel(c) \subseteq \prod_{x \in scp(c)} dom^{init}(x)$.*

L'arité d'une contrainte c correspond au nombre de variables contenues dans sa portée, c'est à dire $|scp(c)|$. Principalement, l'arité permet de qualifier et de catégoriser les contraintes : une contrainte avec une arité égale à 1 est appelée contrainte unaire et porte sur une seule variable, pour une arité égale à 2 c'est une contrainte binaire portant sur deux variables, et pour une arité égale à 3 (ternaire) et au delà, nous parlons de contraintes n-aires. Pour les contraintes unaires, binaires et ternaires, nous plaçons généralement le nom des variables en indice de la lettre c. Par exemple, pour une contrainte unaire impliquant x, nous noterons c_x, pour une contrainte binaire impliquant x et y, nous noterons c_{xy}, et pour une contrainte ternaire impliquant x, y, et z, nous noterons c_{xyz}. Plus généralement, pour une contrainte d'arité quelconque, nous noterons c_S la contrainte dont $scp(c_S) = S$. Deux variables impliquées dans au moins une même contrainte sont dites voisines ou connectées. Le degré (initial) d'une variable est le nombre de contraintes dans lesquelles elle est impliquée. Une contrainte est couverte si toutes les variables appartenant à sa portée sont assignées.

Définition 6 (Tuple) *Un tuple est représenté par une séquence de valeurs entourées de parenthèses et séparées à l'aide de virgules. Au niveau d'une contrainte c d'arité r, un tuple $\tau = (v_1, \ldots, v_r)$ correspond à une instanciation de l'ensemble des variables de la portée de c telle que $\forall 1 \leq i \leq r, v_i \in dom^{init}(x_i), x_i \in scp(c)$. Un tuple contenant r valeurs est appelé un r-tuple.*

La valeur affectée à une variable x dans un tuple τ est notée $\tau[x]$. Chaque élément du produit cartésien $\prod_{i=1}^{r} dom^{init}(x_i)$ des domaines des variables de la portée de c est représenté par un r-tuple.

Exemple 4 *Soit c_{xyz} une contrainte dure avec $dom(x) = dom(y) = dom(z) = \{a, b, c\}$ et soit une instanciation $I = \{(x, a), (y, c), (z, b)\}$ dans c_{xyz}. Le tuple τ correspondant à I sur c_{xyz} est noté $\tau = (a, c, b)$, avec $\tau[x] = a, \tau[y] = c$ et $\tau[z] = b$.*

Une contrainte peut être représentée en intention, en extension ou de manière implicite. Nous parlons alors de contraintes en intention, de contraintes en extension et de contraintes globales.

Définition 7 (Contrainte en intention) *Une contrainte en intention est une contrainte dans laquelle les instanciations autorisées sont définies par un prédicat, c'est à dire une fonction qui associe vrai ou faux à toute instanciation. Ce prédicat est défini à partir d'opérateurs logiques, relationnels et arithmétiques.*

Définition 8 (Contrainte en extension) *Une contrainte en extension est une contrainte dont la relation est représentée sous la forme d'une liste de tuples. On parle de relation de type support (resp. conflit) quand il s'agit de la liste des tuples autorisés (resp. interdits) par la contrainte.*

Une contrainte en extension est aussi appelée contrainte table. Nous noterons $table[c_{xyz}]$ la liste des tuples autorisés dans c_{xyz} dans le cas d'une représentation dite "positive" et $\overline{table}[c_{xyz}]$ la liste des tuples interdits dans le cas d'une représentation dite "négative" : nous parlons de contraintes tables positives ou négatives. Notons que $table[c_{xyz}] = rel(c_{xyz})$ et que $\overline{table}[c_{xyz}] = \prod_{x \in scp(c)} dom^{init}(x) \backslash rel(c_{xyz})$. La table $table[c_{xyz}]$ est représentée par un tableau de tuples indicé de 0 à $(t-1)$ avec t qui représente la taille de la table, ce qui correspond pour $table[c_{xyz}]$ au nombre de tuples autorisés dans c_{xyz}. Le i$^{\text{ème}}$ tuple dans $table[c_{xyz}]$ est donné par $table[c_{xyz}][i]$. Les tuples peuvent être stockés par ordre lexicographique : dans ce cas, pour tout tuple τ_i avec $0 \leq i < t$, $\tau_i \prec \tau_{i+1}$ avec $\tau_i = table[c_{xyz}][i]$. Le nombre de tuples pouvant être conséquent, il est important de choisir la sémantique de la représentation (support ou conflit) qui va permettre de minimiser le nombre de tuples de la relation.

Exemple 5 *Soient deux contraintes c_x et c_{xy}, avec $dom(x) = dom(y) = \{a, b, c\}$, représentées en intention et en extension :*

- *$rel(c_x) : x \neq a$, est la représentation de c_x en intention, tandis que $table[c_x] = \{(b), (c)\}$ (ou $\overline{table}[c_x] = \{(a)\}$) est la représentation de c_x en extension.*
- *$rel(c_{xy}) : x \neq y$, est la représentation de c_{xy} en intention, tandis que $table[c_{xy}] = \{(a, b), (a, c), (b, a), (b, c), (c, a), (c, b)\}$ (ou $\overline{table}[c_{xy}] = \{(a, a), (b, b), (c, c)\}$) est la représentation de c_{xy} en extension.*

Définition 9 (Tuple valide) *Un tuple $\tau = \{(x_1, v_1), ..., (x_r, v_r)\}$ est valide si et seulement si $\forall 1 \leq i \leq r$ $v_i \in dom(x_i)$, sinon il est non valide.*

Exemple 6 *Dans la contrainte c_{xy} définie ci-dessus, le tuple (c, b) est valide car $c \in dom(x)$ et $b \in dom(y)$, par contre le tuple (e, b) n'est pas valide car $e \notin dom(x)$.*

Définition 10 (Tuple autorisé (resp. interdit)) *Un tuple τ est autorisé (resp. interdit) dans une contrainte c si et seulement si $\tau \in rel(c)$ (resp. $\tau \notin rel(c)$).*

Définition 11 (Tuple support (resp. conflit)) *Un tuple τ est un support (resp. conflit) dans une contrainte c si et seulement si τ est un tuple valide et autorisé (resp. interdit) dans c.*

Exemple 7 *Dans la contrainte c_{xy} définie ci-dessus, le tuple valide (c, b) est autorisé car il appartient à $rel(c_{xy})$: $\{(a, b), (a, c), (b, a), (b, c), (c, a), (c, b)\}$: il s'agit donc d'un tuple support dans c_{xy}. Par contre, le tuple valide (b, b) est interdit car il n'appartient pas à $rel(c_{xy})$: il s'agit donc d'un tuple conflit.*

Définition 12 (Projection/restriction d'une instanciation) *Soit un ensemble de variables X et une instanciation I. La projection/restriction de I sur X, notée $I[X]$ ou $I_{\downarrow X}$, est définie par $I[X] = \{(x, a) \mid (x, a) \in I \wedge x \in X\}$.*

Définition 13 (Instanciation satisfaisante/insatisfaisante) *Une instanciation I couvre une contrainte c ssi $scp(c) \subseteq vars(I)$. Une instanciation I satisfait une contrainte c si et seulement si elle couvre c et que le tuple correspondant à la restriction de I sur $scp(c)$ est un support dans c ; on dit également que la contrainte c est satisfaite par I. A contrario, l'instanciation I ne satisfait pas la contrainte c si elle couvre cette contrainte et que le tuple correspondant à la restriction de I sur $scp(c)$ est un conflit dans c ; la contrainte c est alors insatisfaite (ou violée) par I.*

Définition 14 (Contrainte globale) *Une contrainte globale est une contrainte avec une sémantique implicite et dont la portée peut contenir un nombre quelconque de variables.*

Ce type de contraintes permet généralement de capturer des problèmes récurrents (ré-utilisabilité) et d'intégrer des techniques spécifiques dédiées (c'est à dire spécifiquement adaptées à la sémantique de la contrainte).

Exemple 8 *Une des contraintes globales les plus connues est sans doute la contrainte $AllDifferent$ dont la sémantique impose que les valeurs affectées aux variables de sa portée soient toutes différentes. Elle peut être définie pour deux variables, $AllDifferent(x, y)$ (similaire à la contrainte en intention c_{xy} avec $rel(c_{xy}) : x \neq y$), tout comme elle peut être définie pour n variables, $AllDifferent(x_1, \ldots, x_n)$. La sémantique est la même dans ces deux utilisations : que ce soient pour deux ou n variables, les valeurs choisies doivent être toutes différentes. Même si la contrainte globale $AllDifferent$ est équivalente à un ensemble de contraintes binaires, son intérêt réside dans sa technique de filtrage dédiée : pour plus d'informations, voir [Régin, 1994, Puget, 1998].*

Définition 15 (Réseau de contraintes) *Un réseau de contraintes se compose d'un ensemble fini de variables et d'un ensemble fini de contraintes portant sur ces variables.*

Nous utiliserons la notation $P = (\mathcal{X}, \mathcal{C})$ pour tout réseau de contraintes (CN pour Constraint Network) P composé de l'ensemble des variables $\mathcal{X}$ et de l'ensemble des contraintes $\mathcal{C}$. L'ensemble des variables du problème P associé à ce réseau de contraintes sera noté $vars(P)$ et l'ensemble des contraintes sera noté $cons(P)$. L'ensemble des variables déjà assignées (variables dites *passées*) dans P sera généralement noté $assigned(P)$ tandis que l'ensemble des variables non encore assignées (variables dites *futures*) sera noté $unassigned(P)$. Pour toute contrainte $c \in cons(P)$, nous avons $scp(c) \subseteq vars(P)$. Nous notons n le nombre de variables $|vars(P)|$, d la taille du plus grand domaine des variables de P, e le nombre de contraintes $|cons(P)|$ et r l'arité maximale des contraintes dans P. L'arité maximale r dans P permet de qualifier et de catégoriser le réseau de contraintes. Un réseau tel que $r = 1$, $r = 2$ et $r = 3$ est appelé respectivement réseau unaire, binaire et ternaire. Pour $r > 3$, nous parlons de réseau n-aire.

Introduisons à présent plusieurs définitions qui seront illustrées dans la section 1.1.2.

Définition 16 (Réseau de contraintes normalisé) *Un réseau de contraintes est normalisé si et seulement si quelles que soient les contraintes c_1 et $c_2 \in cons(P)$, $c_1 \neq c_2 \Rightarrow scp(c_1) \neq scp(c_2)$.*

Pour normaliser un réseau (qui ne l'est pas), toutes les contraintes de même portée doivent être fusionnées.

Définition 17 (Instanciation localement cohérente) *Une instanciation I est localement cohérente sur un réseau de contraintes P si et seulement si I est valide et que toutes les contraintes couvertes par I sont satisfaites. Si $vars(I) = vars(P)$, I correspond alors une instanciation localement cohérente complète, sinon elle est partielle.*

Définition 18 (Solution) *Une solution d'un réseau de contraintes P est une instanciation localement cohérente complète sur P.*

L'ensemble des solutions d'un réseau de contraintes P est noté $sols(P)$.

Définition 19 (Instanciation globalement incohérente) *Une instanciation est globalement incohérente si elle ne peut pas être étendue à une solution, sinon elle est globalement cohérente. Une instanciation globalement incohérente est appelée aussi nogood.*

Une instanciation localement incohérente est nécessairement globalement incohérente, cependant la réciproque n'est pas vrai.

Définition 20 (Réseau de contraintes satisfaisable) *Un réseau de contraintes est satisfaisable si et seulement si il admet au moins une solution, sinon il est insatisfaisable.*

Définition 21 (Sous-réseau de contraintes) *Un sous-réseau de contraintes P' d'un réseau de contraintes P est un réseau de contraintes obtenu en supprimant des contraintes de P et/ou des variables de P.*

Définition 22 (Réseaux de contraintes équivalents) *Deux réseaux de contraintes P et P' sont équivalents si et seulement si $sols(P) = sols(P')$.*

Dans certains cas, il peut s'avérer utile de connaître les raisons de l'insatisfaisabilité d'un réseau de contraintes, c'est à dire les contraintes (et donc les variables) non satisfaites qui participent à la non satisfaction globale du réseau. En effet, ces informations peuvent permettre de comprendre et d'apporter les réparations ou modifications nécessaires à ce réseau pour le rendre satisfaisable quand c'est nécessaire. Ces ensembles de contraintes (et variables) sont appelés des noyaux insatisfaisables et peuvent être minimaux :

Définition 23 (Noyau insatisfaisable) *Un noyau insatisfaisable d'un réseau de contraintes P est un sous-réseau de contraintes de P qui est insatisfaisable.*

Définition 24 (Noyau insatisfaisable minimal) *Un noyau insatisfaisable est minimal (MUC pour Minimal Unsatisfiable Core) si et seulement si tout sous-ensemble strict de ce noyau est satisfaisable. Éliminer une contrainte de ce noyau le rend donc satisfaisable.*

Ces noyaux, minimaux ou pas, constituent une *preuve d'insatisfaisabilité* d'un réseau insatisfaisable. Diverses méthodes, comme [de Siqueira et Puget, 1988], [Junker, 2004], [Hemery *et al.*, 2006] et [Wieringa, 2012], ont été proposées pour déterminer et extraire ces noyaux. Nous décrirons plus en détail dans le chapitre 4 la méthode proposée dans [Hemery *et al.*, 2006], méthode d'extraction que nous avons mise en place et exploitée dans le cadre de notre deuxième contribution.

1.1.2 Un exemple concret : le problème de coloriage de carte

Afin de mettre en pratique les différents concepts présentés dans la partie précédente, nous allons nous intéresser au problème de coloriage de carte. Ce problème consiste, pour une carte plane découpée en régions adjacentes, à colorier chacune de ces régions tout en s'assurant que deux régions adjacentes ne soient pas de la même couleur. Depuis le théorème des quatre couleurs [Wilson et Nash, 2003], nous savons qu'il est possible de colorier n'importe quelle carte avec seulement quatre couleurs. Nous allons donc modéliser sous la forme d'un réseau de contraintes une instance particulière du problème de coloriage de carte basée sur quatre couleurs.

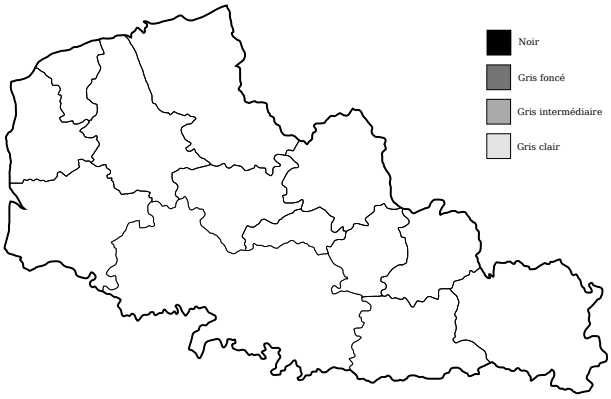


FIG. 1.1 – Carte découpée en régions à colorier avec quatre couleurs : noir, gris foncé, gris intermédiaire et gris clair.

L'objectif est donc de colorier la carte illustrée de la figure 1.1, c'est à dire affecter une couleur parmi les couleurs *noir - gris foncé - gris intermédiaire - gris clair* à chaque région, tout en respectant la contrainte imposant que deux régions voisines ne soient pas de la même couleur. Dans ces termes, la modélisation de ce problème en un réseau de contraintes devient naturelle. En effet, les variables représentent chacune une région à colorier et les domaines associés sont constitués des différentes couleurs disponibles. Nous choisissons de faire correspondre la valeur n à la couleur noire, la valeur gf au gris foncé, la valeur gi au gris intermédiaire et la valeur gc au gris clair. La figure 1.2 illustre l'ensemble des variables de notre réseau de contraintes. Plus formellement, soit $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ le réseau que nous modélisons, nous avons $\mathcal{X} = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}\}$ avec $\forall 0 \leq i \leq 12, dom^{init}(x_i) = \{n, gf, gi, gc\}$.

Les contraintes interdisent une couleur identique pour deux régions voisines. Nous pouvons représenter cela par une contrainte binaire en intention, définie par le prédicat d'inégalité $\neq$, pour chaque couple de régions voisines. Ainsi, $\mathcal{C}$ définit l'ensemble des contraintes binaires représentées en intention de notre réseau :

$$\mathcal{C} = \left\{ \begin{array}{lll} c_{x_0x_1} : x_0 \neq x_1 & c_{x_0x_3} : x_0 \neq x_3 & c_{x_1x_2} : x_1 \neq x_2 \\ c_{x_1x_3} : x_1 \neq x_3 & c_{x_2x_3} : x_2 \neq x_3 & c_{x_3x_4} : x_3 \neq x_4 \\ c_{x_3x_5} : x_3 \neq x_5 & c_{x_3x_6} : x_3 \neq x_6 & c_{x_4x_5} : x_4 \neq x_5 \\ c_{x_4x_8} : x_4 \neq x_8 & c_{x_5x_6} : x_5 \neq x_6 & c_{x_5x_7} : x_5 \neq x_7 \\ c_{x_5x_8} : x_5 \neq x_8 & c_{x_6x_7} : x_6 \neq x_7 & c_{x_6x_9} : x_6 \neq x_9 \\ c_{x_6x_{11}} : x_6 \neq x_{11} & c_{x_7x_8} : x_7 \neq x_8 & c_{x_7x_9} : x_7 \neq x_9 \\ c_{x_8x_9} : x_8 \neq x_9 & c_{x_6x_7} : x_6 \neq x_7 & c_{x_9x_{10}} : x_9 \neq x_{10} \\ c_{x_9x_{11}} : x_9 \neq x_{11} & c_{x_{10}x_{11}} : x_{10} \neq x_{11} & c_{x_{10}x_{12}} : x_{10} \neq x_{12} \\ c_{x_{11}x_{12}} : x_{11} \neq x_{12} & & \end{array} \right\}$$

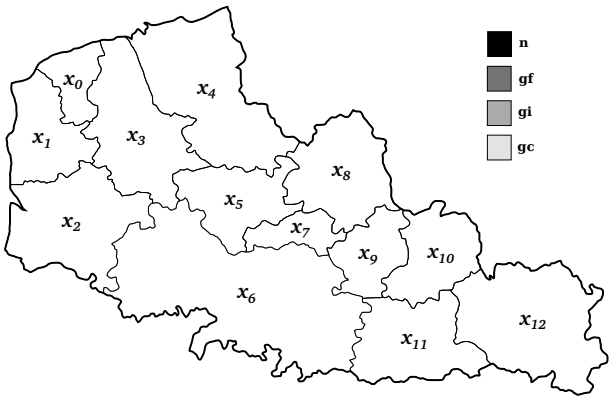


FIG. 1.2 – Chaque région est représentée par une variable. Pour chaque variable, quatre valeurs (de couleurs) possibles : n, gf, gi et gc.

Dans ce réseau $\mathcal{P} = (\mathcal{X}, \mathcal{C})$:

- $\{(x_0, gc), (x_1, n), (x_2, gi), (x_3, gf), (x_4, gc), (x_5, gi)\}$ (figure 1.3) est une instanciation localement cohérente partielle. En effet, elle est valide et toutes les contraintes qu'elle couvre, à savoir $c_{x_0x_1}$, $c_{x_0x_3}$, $c_{x_1x_2}$, $c_{x_1x_3}$, $c_{x_2x_3}$, $c_{x_3x_4}$, $c_{x_3x_5}$ et $c_{x_4x_5}$, sont satisfaites. Elle est partielle car toutes les variables n'ont pas été assignées.
- $\{(x_0, gc), (x_1, n), (x_2, gi), (x_3, gf), (x_4, gc), (x_5, gf), (x_6, gi)\}$ (figure 1.4) est une instanciation incohérente partielle. En effet, parmi les contraintes couvertes, les contraintes $c_{x_2x_6}$ et $c_{x_5x_6}$ ne sont pas satisfaites. Toute instanciation complète obtenue par extension de cette instanciation sera forcément incohérente.
- $\{(x_0, gc), (x_1, n), (x_2, gi), (x_3, gf), (x_4, gc), (x_5, gi), (x_6, gc), (x_7, n), (x_8, gf), (x_9, gi), (x_{10}, n), (x_{11}, gf), (x_{12}, gi)\}$ (figure 1.5) est une solution. Toutes les variables du réseau ont été assignées et toutes les contraintes sont satisfaites.

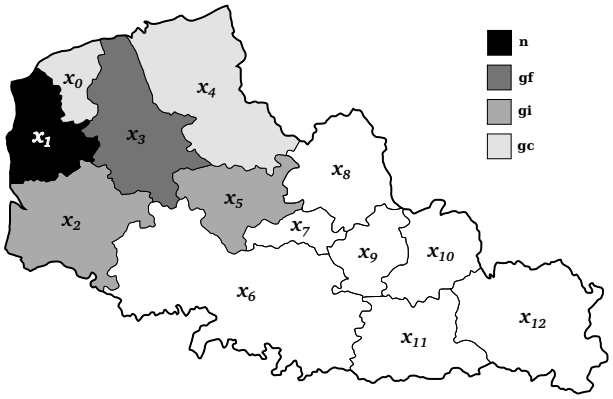


FIG. 1.3 – Une instanciation localement cohérente partielle.

Le réseau de contraintes défini par $(\mathcal{X}, \mathcal{C})$ est normalisé. En effet, toutes les contraintes appartenant à $\mathcal{C}$ ont des portées strictement différentes. De plus, il est satisfaisable, il possède au moins une solution représentée par la figure 1.5.

La représentation graphique d'un réseau de contraintes normalisé peut être obtenue à partir de son hypergraphe, appelé aussi graphe (primaire) de contraintes dans le cadre d'un réseau de contraintes binaire. Il s'agit d'une macro-structure du réseau de contraintes où seules les variables et les portées

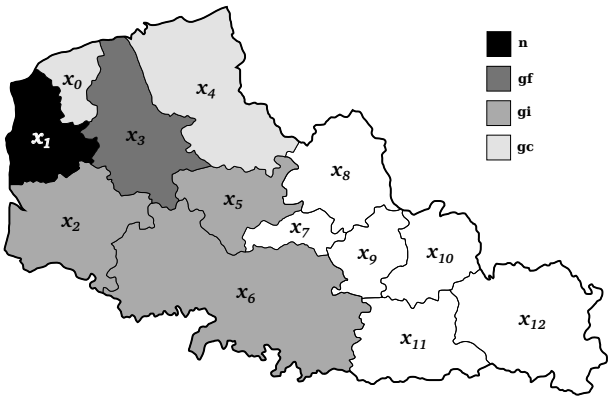


FIG. 1.4 – Une instanciation globalement incohérente (nogood).

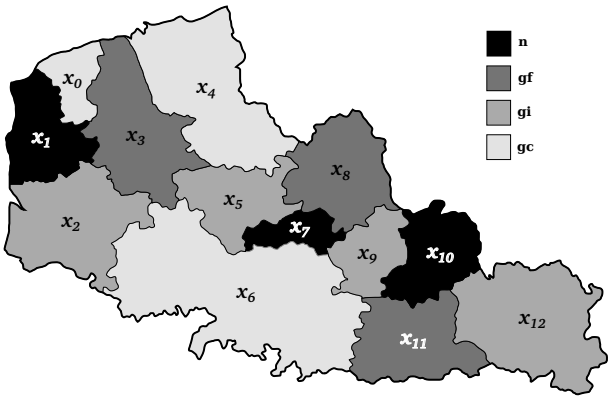


FIG. 1.5 – Une instanciation localement cohérente complète (solution).

des contraintes du réseau sont représentées. Pour les réseaux binaires, les sommets représentent les variables et les arêtes représentent les contraintes. Deux variables appartenant à la portée d'une même contrainte sont liées par une arête. Le graphe de contraintes associé au réseau de contraintes du problème de coloriage de carte est donné par la figure 1.6. Par ailleurs, nous représenterons également parfois les contraintes par des micro-structures appelées graphes de compatibilité [Jégou, 1993]. Comme illustré dans la figure 1.7 pour la contrainte $c_{x_8x_9}$, les deux variables x_8 et x_9 sont représentées avec les valeurs de leurs domaines respectifs et les tuples supports sont représentés par une arête.

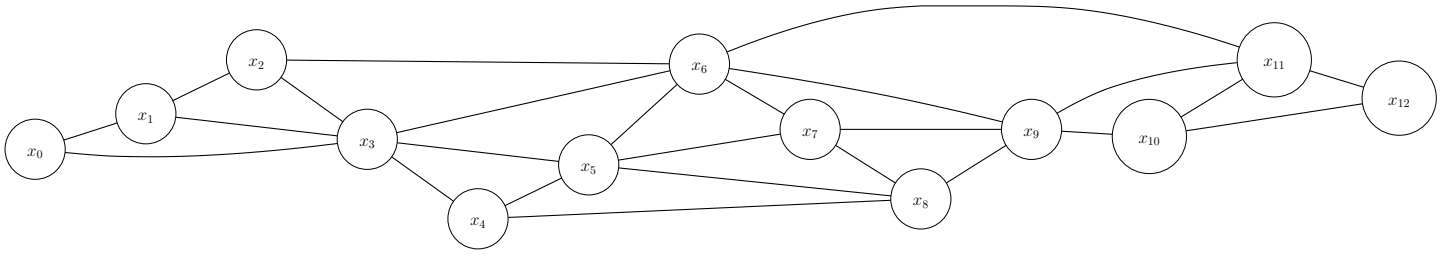


FIG. 1.6 – Le graphe de contraintes pour le problème de coloriage de carte.

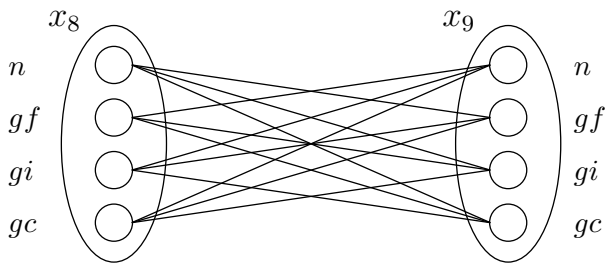


FIG. 1.7 – Le graphe de compatibilité de la contrainte $c_{x_8x_9}$ du problème de coloriage de carte.

1.1.3 Résolution d'un réseau de contraintes

La résolution d'un réseau de contraintes consiste (généralement) à trouver une solution ou prouver l'absence de solution(s). Il s'agit du problème de satisfaction de contraintes (appelé aussi CSP pour *Constraint Satisfaction Problem*). La démarche consiste donc à modéliser un problème sous la forme d'un réseau de contraintes (abordé dans la section précédente), puis de le résoudre dans le cadre de la programmation par contraintes [Rossi *et al.*, 2006] par un programme appelé solveur. Nous abordons dans cette partie la complexité que représente la résolution du problème de satisfaction de contraintes et nous présentons le solveur $AbsCon$ qui a été utilisé pour implémenter et expérimenter les différentes contributions présentées dans ce manuscrit.

Problème de décision `NP-Complet`

Le problème de satisfaction de contraintes est un problème de décision pour lequel la question que l'on se pose est : existe-t-il une instanciation complète satisfaisant toutes les contraintes ? En d'autres termes, le réseau de contraintes est-il satisfaisable ou insatisfaisable ? Un problème de décision est décidable s'il existe un algorithme qui peut répondre en un nombre d'étapes fini à la question posée par ce problème, cependant il est intéressant de savoir si cet algorithme peut donner la réponse efficacement ou non. La théorie de la complexité, branche fondamentale de l'informatique théorique qui s'intéresse principalement aux problèmes de décision, permet de caractériser ces problèmes en termes de quantité de ressources en temps et en espace nécessaires pour les algorithmes qui les résolvent. Les problèmes sont ainsi regroupés par classe de complexité. Dans le cadre du problème de satisfaction de contraintes, nous nous intéressons plus particulièrement aux classes `P`, `NP`, `NP-Difficile` et `NP-Complet`. La classe `P` (Polynomial) regroupe les problèmes de décision pouvant être décidés par un algorithme s'exécutant en temps polynomial (par rapport à la taille de l'entrée) sur une machine de Turing déterministe [Turing, 1936]. Par exemple, le problème de *connexité dans un graphe*, qui consiste à déterminer si toutes les paires de sommets sont reliées par un chemin, appartient à la classe `P` : en effet, il existe un algorithme en temps polynomial ($O(n^2)$, avec n le nombre de sommets) pour répondre à ce problème. La classe `NP` (Non-déterministe Polynomial) regroupe les problèmes de décision pouvant être décidés par un algorithme en temps d'exécution polynomial sur une machine de Turing non-déterministe [Papadimitriou, 1994]. Il existe pour ces problèmes un algorithme en temps d'exécution polynomial sur une machine de Turing déterministe qui permet de tester la validité d'une solution potentielle. Pour les problèmes `NP` qui ne sont pas dans `P`, il n'existe donc pas d'algorithme polynomial pour les résoudre si bien sûr `P` $\neq$ `NP` comme on le conjecture à l'heure actuelle, le problème de savoir si `P` $=$ `NP` ou `P` $\neq$ `NP` restant non résolu. Le problème de satisfaction de contraintes fait partie de la classe des problèmes `NP-Completes`, c'est à dire les problèmes qui sont `NP` et `NP-Difficiles`. Plus précisément, il s'agit des problèmes pour lesquels : 1) il existe un algorithme en temps d'exécution polynomial permettant de tester la validité d'une solution potentielle (propriété `NP`), 2) tous les problèmes de la classe `NP` se ra-

mènent à ces problèmes via une réduction polynomiale [Black, 1999] (propriété `NP-Difficile`). Il s'agit donc d'un problème `NP` qui est au moins aussi difficile que tous les problèmes appartenant à la classe `NP`. Contrairement à un problème `NP-Complet`, un problème `NP-Difficile` n'a pas besoin d'être dans `NP` et peut correspondre à un problème d'optimisation comme expliqué dans la section 2.2.3. Une illustration des inclusions de ces différentes classes de complexité est proposée dans la figure 1.8.

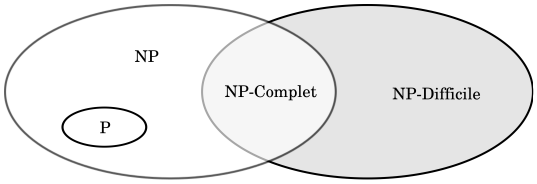


FIG. 1.8 – Représentation des inclusions des classes `P`, `NP`, `NP-Difficile` et `NP-Complet` dans la conjecture $\mathtt{P} \neq \mathtt{NP}$.

La preuve de `NP-Complétude` d'un problème se fait généralement en prouvant qu'il existe une réduction polynomiale d'un problème déjà prouvé `NP-Complet` vers ce problème. La première preuve de `NP-Complétude` a été démontrée par le théorème de Cook [Cook, 1971] pour le problème de satisfaisabilité propositionnelle *SAT* (pour *problème de SATisfaisabilité booléenne*) et depuis, bon nombre de problèmes ont pu être prouvés `NP-Complets` par réduction polynomiale de ce problème. C'est le cas notamment pour le problème de satisfaction de contraintes.

Un solveur générique : AbsCon

Dans le cadre de la programmation par contraintes, la modélisation et la résolution d'un CSP sont deux étapes distinctes. Nous utilisons un programme appelé solveur de contraintes pour résoudre un réseau de contraintes. Les solveurs sont des programmes qui traitent des réseaux de contraintes et qui prouvent soit la satisfaisabilité d'un réseau en retournant au moins une solution, soit son insatisfaisabilité. On trouve trois catégories d'utilisateurs des CSP : ceux qui modélisent les problèmes (vision du solveur comme une boîte noire), ceux qui programment les solveurs chargés de résoudre ces problèmes et ceux qui réalisent les deux. Cependant, dans une recherche d'efficacité, ces différentes catégories d'utilisateurs doivent avoir une vision globale de la programmation par contraintes. En effet, même si les étapes de modélisation et de résolution sont relativement indépendantes, la manière de modéliser un problème a un impact sur l'efficacité de sa résolution, tout comme la façon de programmer un solveur peut avoir des conséquences sur l'efficacité pour des problèmes bien particuliers. Il existe ainsi des solveurs dédiés, développés pour la résolution efficace de problèmes spécifiques, et des solveurs génériques qui vont être capables de traiter tout type de problème dans une efficacité relativement correcte par rapport aux solveurs spécifiques dédiés. Différents solveurs existent : des solveurs propriétaires comme $Comet$[2] et $ILOG$[3], et des solveurs libres comme $Choco$[4], $Gecode$[5] et $Minion$[6]. Dans le cadre des contributions présentées dans ce manuscrit, c'est le solveur $AbsCon$[7] qui a été utilisé. Il s'agit d'une plateforme générique de satisfaction de contraintes développée en Java. Le format de modélisation pour les problèmes traitables par $AbsCon$ est le XCSP [Roussel et Lecoutre, 2009]. La plateforme $AbsCon$ permet de traiter des problèmes des cadres CSP et WCSP en offrant des possibilités de recherche complète et de recherche

[2]Comet technical papers website : http ://dynadec.com/resources/comet-technical-papers/
[3]ILOG Solver website : www.ilog.com
[4]Choco constraint programming system website : http ://choco.sourceforge.net
[5]Generic constraint development environment website : http ://www.gecode.org
[6]Minion Open Source software website : http ://minion.sourceforge.net/
[7]AbsCon website : http ://www.cril.univ-artois.fr/ lecoutre/software.html

locale ainsi que la mise en œuvre des différentes techniques d'inférence abordées dans les sections 1.2 et 2.3 de la partie I. À noter que la plateforme $AbsCon$ permet également de travailler sur des problèmes des cadres Max-CSP et COP (non présentés dans ce manuscrit).

Généralement, les solveurs de contraintes résolvent des CSP en alternant des phases d'inférence et de recherche. Nous présentons les spécificités de ces phases dans les deux sections suivantes.

1.2 Inférence

Dans le cadre de la programmation par contraintes, toutes les combinaisons ne peuvent pas être testées jusqu'à trouver une solution (ou absence de solution) sans entraîner une explosion combinatoire. L'inférence consiste à faire des déductions qui vont permettre de simplifier le réseau de contraintes à résoudre, l'objectif étant d'éviter de parcourir des zones inutiles de l'espace de recherche. Ces déductions vont en effet permettre l'identification et la suppression de valeurs ne pouvant participer à aucune solution pour le réseau. Certaines instanciations globalement incohérentes vont ainsi pouvoir être identifiées et évitées. Par exemple, considérons l'instanciation $\{(x_0, gc),(x_1, n),(x_2, gi),(x_3, gf),(x_4, gc),(x_5, gi)\}$ illustrée dans la figure 1.3 dans le cadre du problème de coloriage de carte (section 1.1.2). L'inférence va alors consister à supprimer, ou filtrer, pour chaque variable certaines valeurs de son domaine ne permettant pas d'étendre cette instanciation à une instanciation localement cohérente. Pour rappel, nous avons $dom(x_6) = dom(x_7) = dom(x_8) = \{n, gf, gi, gc\}$ dans le réseau après l'instanciation. Les déductions qui peuvent être ainsi faites en considérant cette instanciation partielle sont représentées dans la figure 1.9.

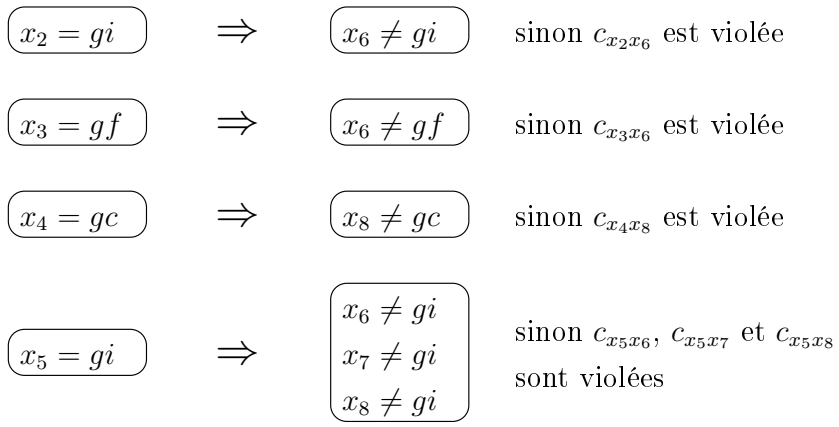


FIG. 1.9 – Déductions faites via un mécanisme d'inférence simple.

Suite à cette instanciation partielle et au processus d'inférence, nous obtenons ainsi un réseau simplifié dans lequel $dom(x_6) = \{n, gc\}$, $dom(x_7) = \{n, gf, gc\}$ et $dom(x_8) = \{n, gf\}$. À noter que les déductions faites sur la variable x_6 (réduction de son domaine à $\{n, gc\}$) nous permettent notamment d'éviter l'instanciation globalement incohérente illustrée par la figure 1.4 car la valeur gi n'est plus présente dans $dom(x_6)$ suite à l'inférence. Nous verrons même qu'il est possible de poursuivre le raisonnement et de propager des déductions par un mécanisme appelé *propagation de contraintes*. Même si dans le pire des cas le problème CSP n'est pas solvable en temps polynomial (selon la conjecture actuelle $\mathsf{P} \neq \mathsf{NP}$), la majorité des algorithmes d'inférence s'exécutent en temps polynomial. L'inférence est donc centrale dans le cadre de la programmation par contraintes et permet d'aider à la résolution d'un grand nombre d'instances du problème CSP. En pratique, on cherche à établir une propriété particulière sur le CN appelée *cohérence*. Cette propriété peut être établie sur le réseau initial uniquement (pré-traitement)

ou/et maintenue lors du processus de résolution.

1.2.1 Notion de cohérence dans un réseau de contraintes

Définition 25 (k-cohérence) *Soit P un réseau de contraintes et k un entier tel que $1 \leq k < n = |vars(P)|$. Le réseau P est k-cohérent [Freuder, 1978] si et seulement si pour tout ensemble X de $k-1$ variables, une instanciation I sur X qui est localement cohérente sur P peut être étendue, considérant une variable $y \in vars(P) \setminus X$ et une valeur $a \in dom(y)$, en une instanciation localement cohérente I' telle que $I' = I \cup \{(y, a)\}$. Sinon, il est k-incohérent.*

Afin de définir la cohérence globale, introduisons à présent la définition de k-cohérence forte.

Définition 26 (k-cohérence forte) *Un réseau de contraintes P est k-cohérent fort, avec $1 \leq k < n$ et n le nombre de variables du réseau, si et seulement si P est i-cohérent pour tout $1 \leq i \leq k$.*

Définition 27 (Cohérence globale) *Un réseau de contraintes est globalement cohérent si et seulement si il est n-cohérent fort, n étant le nombre total de variables du réseau. Un réseau de contraintes globalement cohérent est satisfaisable.*

Pour établir la k-cohérence, un algorithme d'inférence va modifier le réseau initial. On parle alors de filtrage par k-cohérence [Freuder, 1978]. Le réseau obtenu correspond généralement à un réseau équivalent plus explicite (ou simplifié). Nous noterons ϕ la cohérence cible et $\phi(P)$ le réseau obtenu en établissant ϕ sur P. Si la cohérence ϕ est vérifiée sur P, on dit que P vérifie ϕ.

Afin d'illustrer ces notions de cohérence et leur maintien, nous allons dans un premier temps nous intéresser plus particulièrement à la 2-cohérence dans le cadre de réseaux binaires (normalisés). Appelée aussi arc-cohérence, il s'agit d'une cohérence fondamentale en programmation par contraintes. Nous verrons également que de nombreuses tentatives d'optimisation pour établir l'arc-cohérence ont été proposées depuis un bon nombre d'années à travers des algorithmes plus ou moins efficaces. Dans un deuxième temps, nous nous intéresserons dans la section 1.2.3 à l'arc-cohérence généralisée dans le cadre de réseaux n-aires (normalisés).

1.2.2 Arc-cohérence et algorithmes

D'après la notion générale de k-cohérence, la 2-cohérence ou arc-cohérence correspond à la propriété que toute instanciation localement cohérente d'une variable peut être étendue en une instanciation localement cohérente de deux variables dans un réseau binaire. Elle est définie pour les valeurs, puis étendue aux contraintes et aux réseaux de contraintes.

Définition 28 (Valeur arc-cohérente) *Soit c une contrainte binaire telle que $scp(c) = \{x, y\}$. Une valeur $a \in dom(x)$ est arc-cohérente dans c si et seulement si il existe au moins une valeur $b \in dom(y)$ telle que $(a, b) \in rel(c)$. La valeur b est appelée support de a pour la contrainte c. De plus, par la propriété de bi-directionnalité [Bessière* et al.*, 1999], a est réciproquement un support de b pour la contrainte c.*

Définition 29 (Contrainte arc-cohérente) *Une contrainte c est arc-cohérente si toutes les valeurs présentes dans le domaine des variables de c sont arc-cohérentes dans c.*

Définition 30 (Réseau arc-cohérent) *Un réseau est arc-cohérent si et seulement si toutes les contraintes de ce réseau sont arc-cohérentes.*

Considérons un exemple de réseau P représenté par son graphe de compatibilité à la figure 1.10. Ce réseau se compose de trois variables x, y et z avec $dom(x) = dom(y) = dom(z) = \{a, b, c\}$. Trois contraintes binaires définies en extension portent sur ces variables : $c_{xz} : \{(a, b), (a, c), (b, c)\}$, $c_{xy} : \{(b, a), (c, a), (c, b)\}$, $c_{yz} : \{(a, c), (b, b), (c, a)\}$. Le réseau P n'est pas arc-cohérent car certaines valeurs appartenant aux domaines de x, y et z ne sont pas arc-cohérentes. Par exemple, la valeur c dans $dom(x)$ n'a pas de support parmi les valeurs dans $dom(z)$ pour la contrainte c_{xz}. Même constat pour la valeur a dans $dom(x)$ pour la contrainte c_{xy}. Cela implique que les contraintes c_{xz} et c_{xy} ne sont pas arc-cohérentes (au contraire de c_{yz}). Afin d'établir l'arc-cohérence sur P, nous allons effectuer un filtrage par arc-cohérence. Il faut d'abord établir l'arc-cohérence pour les contraintes c_{xy} et c_{xz}.

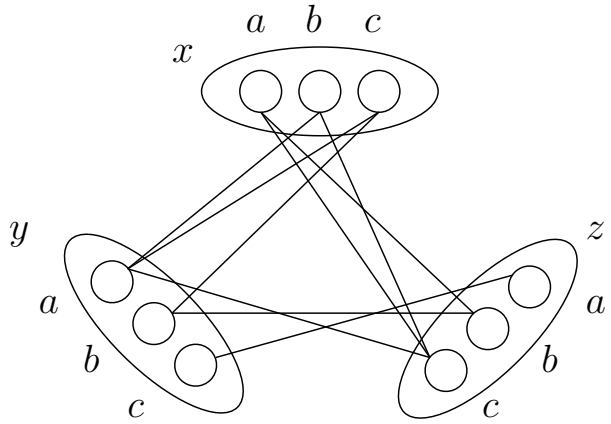


FIG. 1.10 – Un réseau P non arc-cohérent.

Pour cela, les valeurs (x, c) et (z, a), qui n'ont pas de support pour c_{xz}, sont supprimées (figure 1.11). La suppression de ces valeurs entraîne également la perte d'un support dans c_{xy} pour les valeurs (y, a) et (y, b) et dans c_{yz} pour (y, c), ce qui est représenté par des traits en pointillés dans la figure. Établir l'arc-cohérence d'une contrainte peut avoir des répercussions sur les autres contraintes du réseau. Dans notre exemple, la contrainte c_{yz} qui était à l'origine arc-cohérente ne l'est plus car la valeur (y, c) n'a plus aucun support pour c_{yz}. Ce phénomène correspond au processus de propagation de contraintes. Cette valeur n'ayant plus de support, contrainte redevient alors non arc-cohérente. Le réseau P obtenu (figure 1.12) n'est toujours pas arc-cohérent.

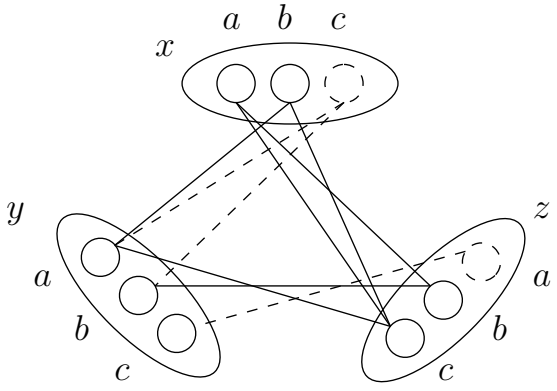


FIG. 1.11 – On traite la contrainte c_{xz}.

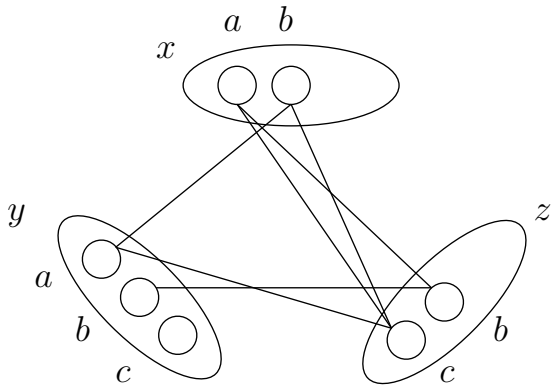


FIG. 1.12 – Un réseau non arc-cohérent.

Dans un second temps, l'arc-cohérence des contraintes c_{xy} et c_{yz} doit être établie. Les valeurs (x, a), (y, b) et (y, c) sont supprimées, ce qui a pour but de faire perdre des supports aux valeurs (z, b) et (z, c) (figure 1.13). La valeur (z, b) n'ayant plus de support pour c_{yz}, l'arc-cohérence de c_{yz} est établie en supprimant (z, b). Le réseau résultant $\phi(P)$ illustré par la figure 1.14, avec ϕ étant l'arc-cohérence, est arc-cohérent : en effet, les contraintes c_{xz}, c_{xy} et c_{yz} sont toutes arc-cohérentes.

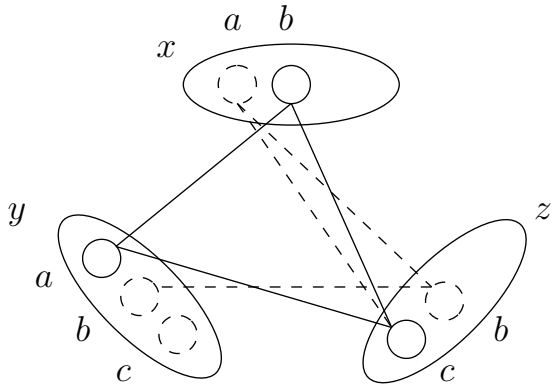


FIG. 1.13 – On traite les contraintes c_{xy} et c_{yz}.

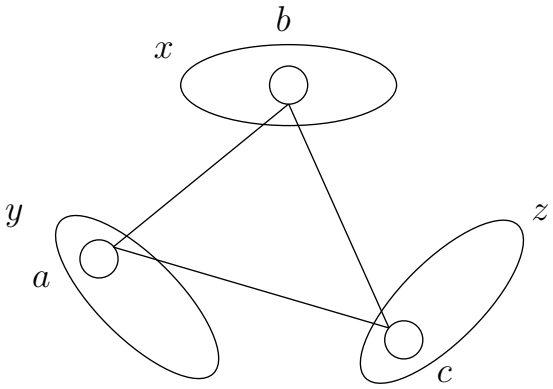


FIG. 1.14 – $\phi(P)$ est un réseau arc-cohérent.

Nous avons vu que des contraintes à l'origine arc-cohérentes pouvaient devenir non arc-cohérentes lors du filtrage par arc-cohérence. Les contraintes peuvent ainsi passer d'un état à l'autre durant tout le processus de filtrage. Le processus de filtrage s'exécute tant qu'il n'a pas atteint un point fixe, c'est à dire que l'arc-cohérence est établie pour toutes les contraintes du réseau, incluant le cas où un domaine vide apparaît. Il est important de noter que ce processus de filtrage est confluent par la propriété de stabilité par union [Bessiere, 2006]. Le réseau obtenu est arc-cohérent et correspond au plus grand sous-réseau équivalent au réseau d'origine : il est unique et représente la fermeture par arc-cohérence.

Bon nombre d'algorithmes ont été proposés dans le but d'établir de façon efficace l'arc-cohérence dans un réseau. La plupart d'entre eux se composent d'un algorithme de propagation incorporant une procédure de révision. Ils sont regroupés en deux familles : les algorithmes comme AC3 [Mackworth, 1977], AC2001/3.1 [Bessière *et al.*, 2005], AC3$_d$ [van Dongen, 2002], AC3.2/3.3 [Lecoutre *et al.*, 2003], AC3^{rm} [Lecoutre et Hemery, 2007], dits à gros grain, et les algorithmes comme AC4 [Mohr et Henderson, 1986], AC6 [Bessiere, 1994], AC7 [Bessière *et al.*, 1999], dits à grain fin. La distinction entre approche à grain fin ou gros grain se fait à travers la nature et le niveau des données stockées et traitées lors de la propagation : triplets *contrainte-variable-valeur* pour le grain fin, couples *contrainte-variable* pour le gros grain. Les procédures de révision des algorithmes à grain fin sont plus fines et apportent certaines optimisations. Cependant, les structures qu'elles nécessitent sont souvent coûteuses en espace et requièrent une implémentation relativement complexe. Nous présentons plus en détail ici les algorithmes AC3, AC2001/3.1 et AC3^{rm}.

L'algorithme de propagation *doAC* (algorithme 1) est commun pour AC3, AC2001/3.1 et AC3^{rm}, l'algorithme *reviser* (appelé à la ligne 4 de l'algorithme 1) étant quant à lui spécifique à chaque approche. L'algorithme *doAC* prend en entrée un réseau de contraintes $P = (\mathcal{X}, \mathcal{C})$ et retourne un booléen : vrai si la fermeture de P par arc-cohérence permet d'identifier une incohérence globale (apparition d'un domaine vide), faux sinon. Il parcourt un ensemble Q contenant, pour chaque contrainte $c \in \mathcal{C}$ et chaque variable $x \in \mathcal{X}$, un couple (c, x) appelé aussi arc (c, x). Tous les arcs (c, x) présents dans Q sont révisés via l'algorithme *reviser* : les valeurs appartenant à $dom(x)$ n'ayant pas de support pour c sont supprimées de $dom(x)$. Si la révision est effective (au moins une valeur supprimée), alors l'algorithme *reviser* retourne vrai. Si tel est le cas et qu'en plus le domaine de la variable x est vide, cela veut dire qu'une incohérence

Algorithm 1: doAC ($P = (\mathcal{X}, \mathcal{C})$: CN) : Booléen

```
1 Q = {(c, x) | c ∈ C ∧ x ∈ scp(c)}
2 tant que Q ≠ ∅ faire
3     Choisir et éliminer (c, x) de Q
4     si reviser(c, x) alors
5         si dom(x) = ∅ alors Retourner faux
6         Q ← Q ∪ {(c', y) | c' ∈ C ∧ x ∈ scp(c') ∧ y ∈ scp(c') ∧ y ≠ x ∧ c' ≠ c}
7 Retourner vrai
```

globale est détectée (faux est retourné). Si ce n'est pas le cas, alors on met à jour Q. Toutes les contraintes impliquant la variable x dans leur portée doivent être révisées : les arcs correspondants sont ainsi ajoutés dans Q.

Algorithme AC3 [Mackworth, 1977]

L'algorithme AC3 propose la technique de révision *reviser-3* (algorithme 2) qui est la plus simple. Considérant un arc (c, x), on vérifie que chaque valeur appartenant à $dom(x)$ possède un support dans la contrainte c. Dans le cas contraire, la valeur est supprimée de $dom(x)$. Si au moins une valeur appartenant à $dom(x)$ est supprimée, alors $vrai$ est retourné (car $suppression = vrai$) : la révision est effective, c'est à dire que le domaine de x a perdu au moins une valeur.

Algorithm 2: reviser-3(c : contrainte avec $scp(c) = \{x, y\}$, x : variable) : Booléen

```
1 suppression ← faux
2 pour chaque a ∈ dom(x) faire
3     si ∄b ∈ dom(y) tel que (a, b) ∈ rel(c) alors
4         dom(x) ← dom(x) \ {a}
5         suppression = vrai
6 Retourner suppression
```

Dans le cadre d'un réseau binaire, l'algorithme de cohérence d'arc AC3 a une complexité en temps de $O(ed^3)$, où e correspond au nombre de contraintes dans P et d la taille maximale parmi les domaines des variables. Il n'est pas optimal en temps comme nous allons le voir dans le paragraphe suivant avec l'algorithme AC2001/3.1.

Algorithme AC2001/3.1 [Bessière et Régin, 2001, Zhang et Yap, 2001, Bessière *et al.*, 2005]

Une explication de la non-optimalité de AC3 est la recherche systématique de supports pour toutes les valeurs appartenant à $dom(x)$ dans c pour un arc (c, x), et cela en recommençant au début du domaine à chaque fois. Nous savons qu'une contrainte peut être révisée plusieurs fois pour établir l'arc-cohérence. Lors de chaque nouvelle révision, il suffit de repartir du dernier support trouvé. C'est à partir de ce constat qu'a été proposé l'algorithme AC2001/3.1.

Soit l'arc (c, x) avec $scp(c) = \{x, y\}$. L'idée est de chercher pour chaque valeur (x, a) un support dans $dom(y)$ pour c uniquement lors de la première révision de (c, x), puis de le stocker dans une structure $last[c, x, a]$: le support de (x, a) dans le domaine de y pour c. Lors de la révision suivante, avant de chercher un support pour la valeur (x, a), on teste si le dernier support trouvé est encore valide,

Algorithm 3: $reviser$-2001(c : contrainte avec $scp(c) = \{x, y\}$, x : variable) : Booléen

```
1 suppression ← faux
2 pour chaque a ∈ dom(x) faire
3     si last[c, x, a] ∉ dom(y) alors
4         si ∄b ∈ dom(y) | b > last[c, x, a] ∧ (a, b) ∈ rel(c) alors
5             dom(x) ← dom(x) \ {a}
6             suppression = vrai
7         sinon
8             last[c, x, a] ← b
9 Retourner suppression
```

c'est à dire si $last[c, x, a] \in dom(y)$. Si ce support est toujours valide, on passe à la valeur suivante de x, sinon on cherche à partir de $last[c, x, a]$ dans $dom(y)$, appelé point de reprise, un nouveau support pour (x, a) (de par la propriété d'uni-directionnalité [Bessière *et al.*, 1999] selon laquelle la recherche de supports est effectuée dans une seule direction). Tout nouveau support est stocké. S'il existe au moins une valeur appartenant à $dom(x)$ pour laquelle aucun support n'a été trouvé, $vrai$ est retourné : la révision est effective. Dans le cadre d'un réseau binaire, l'algorithme de cohérence d'arc AC2001/3.1 a une complexité en temps $O(ed^2)$ et une complexité en espace $O(ed)$. Il est optimal en temps ($O(ed^2)$ au lieu de $O(ed^3)$ pour AC3) mais nécessite une structure supplémentaire $last$.

Algorithme AC3^{rm} [Lecoutre et Hemery, 2007]

Algorithm 4: $reviser$-3^{rm} (c : contrainte avec $scp(c) = \{x, y\}$, x : variable) : Booléen

```
1  suppression ← faux
2  pour chaque a ∈ dom(x) faire
3      si supp[c, x, a] ∉ dom(y) alors
4          si ∄b ∈ dom(y) tel que (a, b) ∈ rel(c) alors
5              dom(x) ← dom(x) \ {a}
6              suppression = vrai
7          sinon
8              supp[c, x, a] ← b ;
9              supp[c, y, b] ← a ;
10 Retourner suppression
```

Pour améliorer AC2001/3.1 en pratique, l'algorithme AC3^{rm} (rm pour résidus multi-directionnels) exploite partiellement la bi-directionnalité des contraintes (multi-directionnalité dans le cas non binaire) déjà introduite dans AC3.2 : si une valeur (y, b) est un support de la valeur (x, a) pour une contrainte c_{xy}, alors la valeur (x, a) est réciproquement un support de la valeur (y, b) pour c_{xy}. Les supports stockés sont appelés supports résiduels ou plus simplement résidus. L'idée est que lorsque un support $\tau = (a, b)$ est trouvé pour une valeur (x, a) dans la contrainte c telle que $scp(c) = \{x, y\}$, il va être également enregistré pour les autres valeurs appartenant à τ : (y, b) va être ainsi enregistré comme résidu de (x, a) et (x, a) comme résidu de (y, b). Comme dans AC2001, une structure additionnelle est nécessaire : $supp$

permet de stocker pour chaque valeur le dernier résidu bi-directionnel trouvé. Lors de chaque révision d'arc, comme dans AC2001, la validité du résidu est testée, et dans le cas où il n'est plus valide, un nouveau tuple support doit être trouvé. Dans le cadre de la bi-directionnalité des contraintes, la propriété d'uni-directionnalité ne peut pas être exploitée. C'est à dire que lorsqu'un support est devenu invalide, nous n'avons pas de point reprise et on doit rechercher un nouveau support à partir de zéro (début du domaine). En effet, avec la bi-directionnalité nous n'avons aucune certitude que le dernier support trouvé corresponde au dernier plus petit support trouvé. Dans le cadre d'un réseau binaire, l'algorithme de cohérence d'arc AC3^{rm} a une complexité en temps $O(ed^3)$ et une complexité en espace $O(ed)$.

Pour résumer, nous avons donc présenté trois algorithmes à gros grain : un algorithme non optimal AC3, puis un algorithme optimal qui est AC2001/3.1, et enfin l'algorithme AC3^{rm}. L'algorithme AC3^{rm} n'est pas optimal en théorie. Cependant il est mieux adapté et donc plus efficace en pratique, notamment pour des réseaux comportant des contraintes dont la dureté est faible ou forte (c'est à dire acceptant beaucoup ou peu de supports) [Lecoutre et Hemery, 2007]. Il est tout à fait compétitif vis à vis de AC2001/3.1. Comme nous le verrons dans la section 1.3.5, consacrée à la résolution, les algorithmes établissant l'arc-cohérence (et la cohérence en général) peuvent être utilisés en pré-traitement et/ou en cours de la recherche. Dans ce cadre, maintenir la cohérence d'arc à l'aide de l'algorithme AC3^{rm} à chaque étape d'une recherche s'avère notamment être plus efficace que de maintenir la cohérence d'arc AC2001/3.1.

1.2.3 Arc-cohérence généralisée et algorithmes

La notion d'arc-cohérence s'applique également aux réseaux non binaires : on parle d'arc-cohérence généralisée (notée GAC pour Generalized Arc Consistency) ou d'hyper arc-cohérence [Krzysztof, 2003].

Définition 31 (Réseau arc-cohérent généralisé) *Soit c une contrainte d'arité $r > 2$ et une variable $x \in scp(c)$. Une valeur (x, a) est arc-cohérente généralisée dans c si et seulement si il existe un tuple $\tau \in rel(c)$ tel que τ soit valide et $\tau[x] = a$. La contrainte c est arc-cohérente généralisée si et seulement si toute valeur (x, a) est arc-cohérente généralisée, avec $x \in scp(c)$ et $a \in dom(x)$. Un réseau est arc-cohérent généralisé si et seulement si toutes ses contraintes sont arc-cohérentes généralisées.*

Dans le cadre de ce manuscrit, nous représenterons les contraintes n-aires de grande arité par des contraintes tables. Pour rappel, ces contraintes sont définies en extension par une table positive (resp. négative) contenant la liste des tuples autorisés (resp. interdits) pour la contrainte. Pour établir GAC via ces algorithmes sur ces contraintes tables, il existe plusieurs techniques de recherche de supports pour une valeur. Nous distinguons deux types d'approches : les techniques classiques traitant la liste originale des tuples autorisés valides et invalides pour une contrainte, puis des techniques optimisées traitant uniquement la liste des tuples autorisés valides via l'utilisation de structures additionnelles. À noter que ces différentes techniques peuvent être utilisées en pré-traitement ou/et au cours de la recherche.

Algorithmes classiques

Parmi les algorithmes classiques, une première approche appelée généralement *GAC-valid* consiste à itérer les tuples valides pour une contrainte jusqu'à ce qu'un tuple soit autorisé. De manière réciproque, une deuxième approche appelée généralement *GAC-allowed* consiste cette fois-ci à parcourir la liste des tuples autorisés jusqu'à ce qu'un tuple soit valide. Malheureusement, parcourir la liste des tuples valides ou autorisés peut s'avérer coûteux et très pénalisant, le nombre de tuples pouvant augmenter de façon exponentielle avec l'arité de la contrainte. Pour pallier cet inconvénient, une troisième approche proposée consiste à alterner parcours des tuples valides et parcours des tuples autorisés (méthode appelée naturellement *GAC-valid+allowed*). Le principe est de chercher le premier tuple valide contenant cette valeur

et de chercher parmi la liste des tuples autorisés supérieurs ou égal à ce tuple un tuple autorisé. Si un tel tuple existe, alors un support a été trouvé. L'intérêt de cette approche est clairement de réaliser des sauts dans les listes de tuples et ainsi éviter de considérer bon nombre de tuples [Lecoutre et Szymanek, 2006].

Les algorithmes AC3, AC2001 et AC3^{rm} abordés dans la section précédente ont été présentés dans le cadre de contraintes binaires. Ils sont généralisables pour des contraintes n-aires et sont appelés GAC3, GAC2001 et GAC3^{rm}. Concernant les techniques *GAC-valid*, *GAC-allowed* et *GAC-valid+allowed*, elles peuvent être exploitées au niveau des algorithmes de révision 2, 3 et 4.

Réduction tabulaire simple STR [Ullmann, 2007] et STR2 [Lecoutre, 2009, Lecoutre, 2011]

L'un des algorithmes de filtrage les plus efficaces pour établir l'arc-cohérence généralisée sur des contraintes tables positives non binaires est appelé technique de réduction tabulaire simple (STR pour Simple Tabular Reduction) [Ullmann, 2007]. À la différence des approches classiques présentées ci-dessus, le principe de STR est de maintenir dynamiquement la liste des tuples autorisés et valides pour une contrainte et permettre ainsi de trouver plus rapidement un support pour une valeur. L'une des particularités de STR est qu'il permet, au delà de maintenir les tuples autorisés valides durant l'avancée de la recherche, de récupérer efficacement la liste des tuples autorisés valides lors d'un retour en arrière (la notion générale de retour en arrière dans un arbre de recherche est abordée dans la section 1.3.4). Pour cette raison, nous présenterons l'algorithme STR maintenant GAC dans un contexte de recherche.

Soit une contrainte table c définie par la table $table[c]$. L'algorithme STR utilise les quatre structures additionnelles suivantes :

- Un tableau $position[c]$ de taille $t = table[c].length$ qui fournit un accès indirect aux tuples appartenant à $table[c]$. Les valeurs dans $position[c]$ représentent une permutation de $\{0, 1, ..., t-1\}$. Le i$^{\text{ème}}$ tuple valide (et autorisé) dans c est accessible par $table[c][position[c][i]]$.
- Une valeur $currentLimit[c]$ qui correspond à la position du dernier tuple valide dans $table[c]$. La table courante de c est composée d'exactement $currentLimit[c] + 1$ tuples. Les valeurs dans $position[c]$ aux indices de 0 à $currentLimit[c]$ sont les positions des tuples courants de c.
- Un tableau $levelLimits[c]$ de taille $n + 1$ (n étant le nombre de variables dans le réseau) où $levelLimits[c][l]$ correspond à la position du premier tuple invalide dans $table[c]$ supprimé quand la recherche était au level l, le level correspondant au nombre de variables assignées durant la recherche. Autrement dit, $levelLimits[c][l]$ est une sauvegarde de la valeur $currentLimit[c]$ au level l dans le cas où au moins un tuple est supprimé au level l, sinon $levelLimits[c][l] = -1$. Exploité pour les retours en arrière, $levelLimits[c]$ est indicé de 0 à n, le level 0 correspondant aux tuples supprimés lors d'une éventuelle phase de pré-traitement. Les tuples supprimés au level l sont accessibles via les valeurs dans $position[c]$ aux indices allant de $currentLimit[c] + 1$ à $levelLimits[c][l]$.
- Un tableau $valeursGAC$ de taille égale à l'arité de c et contenant pour chaque variable appartenant à $scp(c)$ les valeurs prouvées comme étant GAC-cohérentes.

Pour résumer, le principe de STR est de découper une table en deux sous-ensembles tels qu'après chaque étape de la recherche, chaque tuple appartient à un et un seul de ces sous-ensembles. L'un de ces sous-ensembles correspond à la table courante et regroupe tous les tuples courants, c'est à dire les tuples autorisés et valides pour une contrainte (autrement dit les supports). Commençons par présenter l'algorithme doGAC^{str} (algorithme 5).

Contrairement aux approches présentées précédemment, l'algorithme doGAC^{str} (algorithme 5) maintient globalement GAC sur chacune des contraintes du réseau et n'effectue pas de révisions successives d'arcs *contrainte-variable*. Il est appliqué au réseau dès qu'une décision (positive ou négative, voir section 1.3.3) a été prise sur une variable x passée en paramètre. Une queue de propagation Q contient

Algorithm 5: doGAC^{str} ($P = (\mathcal{X}, \mathcal{C})$: CN, x : variable) : Booléen

```
1 Q = {x}
2 tant que Q ≠ ∅ faire
3     Choisir et éliminer x de Q
4     pour chaque contrainte c ∈ C | x ∈ scp(c) faire
5         X_reduits ← STR(P, c)
6         pour chaque variable x ∈ X_reduits faire
7             si dom(x) = ∅ alors Retourner faux
8             Q ← Q ∪ {x}
9 Retourner vrai
```

initialement la variable x et contiendra les variables dont le domaine a été modifié durant l'établissement de GAC. Tant que la queue de propagation n'est pas vide, une variable en est extraite et STR est appliqué (algorithme 6) à chaque contrainte contenant cette variable dans sa portée. L'ensemble $X_{reduits}$ de variables dont le domaine a été réduit durant l'application de STR sur une contrainte est retourné : si au moins l'une de ces variables a un domaine vide alors le réseau est prouvé non GAC-cohérent (faux est retourné), sinon chacune de ces variables est ajoutée dans la queue de propagation. Si aucun domaine n'est vide après l'établissement de GAC, alors le réseau est GAC-cohérent (vrai est retourné).

Après avoir initialisé la structure $valeursGAC$ à vide pour toutes les variables non assignées de la portée, l'algorithme STR (algorithme 6) parcourt les tuples aux indices allant de 0 à $currentLimit[c]$ dans $position[c]$. Si le tuple est valide (vérifié par l'algorithme 7), alors pour chaque variable x on ajoute dans $valeursGAC[x]$ (si elles ne s'y trouvent pas déjà) les valeurs respectives dans le tuple pour lesquelles un support vient d'être trouvé et qui sont donc GAC-cohérentes. Si par contre le tuple est invalide, alors il est supprimé de la table courante via un appel à l'algorithme 8 avec comme paramètre la position du tuple et le level courant, c'est à dire le nombre de variables déjà assignées. Si c'est le premier tuple supprimé durant ce level, alors $levelLimits[c]$ est mis à jour avec $currentLimit[c]$. Un tuple est supprimé en inversant sa position dans $position[c]$ avec le dernier tuple courant pointé par $currentLimit[c]$ ($swap$ à la ligne 3), et en décrémentant ensuite la valeur de $currentLimit[c]$. Une fois les tuples courants parcourus, on vérifie pour chaque variable si toutes les valeurs de son domaine sont GAC-cohérentes (autant de valeurs dans le domaine que dans $valeursGAC$). Si au moins une valeur n'est pas GAC-cohérente, le domaine de la variable est mis à jour pour contenir uniquement les valeurs GAC-cohérentes. De plus, si le domaine est vide, le réseau n'est pas GAC-cohérent et c'est un échec (FAILURE est retourné), sinon la variable est ajoutée dans l'ensemble $X_{reduits}$ contenant les variables pour lesquelles le domaine a été réduit.

L'algorithme 9 est appelé lors d'un retour en arrière pour restaurer les tuples supprimés dans un level précédent. Cette restauration de tuples se fait en manipulant la structure $levelLimits[c]$ au niveau de ce level. Si des tuples ont étés supprimés lors de ce level, alors on met à jour la limite des tuples courants $currentLimit[c]$ afin qu'elle pointe vers le premier tuple supprimé lors de ce level qui est référencé par $levelLimits[c]$, puis on réinitialise $levelLimits[c]$ à -1.

Il est à noter que les décisions négatives ne sont pas considérées pour le retour en arrière. En effet, nous considérons la stratégie de recherche qui consiste pour chaque variable à d'abord affecter une valeur, puis ensuite la réfuter (branchement binaire, voir section 1.3.3). Une fois ces deux décisions prises, on a tout exploré pour cette variable. Si un échec apparaît, le retour en arrière doit être alors effectué sur une assignation de variable antérieure (décision positive). C'est notamment la raison pour laquelle la taille de $levelLimits[c]$ est égale au nombre de variables (pour une variable, une affectation de valeur à la fois).

Algorithm 6: STR ($P = (\mathcal{X}, \mathcal{C})$: CN, c : contrainte) : ensemble de variables

pour chaque *variable* $x \in scp(c) \mid x \notin assigned(P)$ **faire**
  $valeursGAC[x] \leftarrow \emptyset$
$i \leftarrow 0$
tant que $i \leq currentLimit[c]$ **faire**
  $index \leftarrow position[c][i]$
  $\tau \leftarrow table[c][index]$
  si $estUnTupleValide(c, \tau)$ **alors**
    pour chaque *variable* $x \in scp(c) \mid x \notin assigned(P)$ **faire**
      si $\tau[x] \notin valeursGAC[x]$ **alors**
        $valeursGAC[x] \leftarrow valeursGAC[x] \cup \{\tau[x]\}$
    $i \leftarrow i + 1$
  sinon
    $supprimerTuple(c, i, |assigned(P)|)$
$X_{reduits} \leftarrow \emptyset$
pour chaque *variable* $x \in scp(c) \mid x \notin assigned(P)$ **faire**
  si $|valeursGAC[x]| < |dom(x)|$ **alors**
    $dom(x) \leftarrow valeursGAC[x]$
    si $dom(x) = \emptyset$ **alors**
      Retourner **FAILURE**
    $X_{reduits} \leftarrow X_{reduits} \cup \{x\}$
Retourner $X_{reduits}$

Algorithm 7: estUnTupleValide (c : contrainte, τ : tuple) : Booléen

pour chaque *variable* $x \in scp(c)$ **faire**
  si $\tau[x] \notin dom(x)$ **alors** Retourner faux
Retourner vrai

Algorithm 8: supprimerTuple (c : contrainte, i : entier, p : entier)

si $levelLimits[c][p] = -1$ **alors**
  $levelLimits[c][p] \leftarrow currentLimit[c]$
$swap(position[c][i], position[c][currentLimit[c]])$
$currentLimit[c] \leftarrow currentLimit[c] - 1$

Algorithm 9: restaurerTuples (c : contrainte, p : entier)

si $levelLimits[c][p] \neq -1$ **alors**
  $currentLimit[c] \leftarrow levelLimits[c][p]$
  $levelLimits[c][p] \leftarrow -1$

La complexité en temps de STR (algorithme 6) pour une contrainte c correspond à $O(r'd + rt')$ avec r' le nombre de variables non assignées dans $scp(c)$ et t' la taille de la table courante de c (nombre de tuples). On y retrouve les complexités respectives $O(r')$, $O(rt')$ et $O(r'd)$ des trois boucles qui constituent STR. La complexité spatiale pour cette même contrainte c est de $O(n + rt)$ avec n pour la taille de la structure $levelLimits[c]$, rt pour $table[c]$, r pour $valeursGAC$ et t pour $position[c]$.

Afin de bien comprendre comment fonctionne l'algorithme STR, nous allons considérer une contrainte ternaire et l'évolution de ces différentes structures lorsque l'algorithme STR est appliqué à cette contrainte au cours d'une recherche.

Exemple 9 *Soit c_{xyz} une contrainte table positive ternaire avec $scp(c_{xyz}) = \{x, y, z\}$ et $dom^{init}(x) = dom^{init}(y) = dom^{init}(z) = \{a, b, c\}$. Elle est définie par $rel(c_{xyz}) = \{(a, b, a), (a, c, a), (a, c, b), (a, c, c), (b, c, a), (b, c, c), (c, a, b)\}$. La figure 1.15 illustre la contrainte table c_{xyz} ainsi que les structures STR à l'initialisation.*

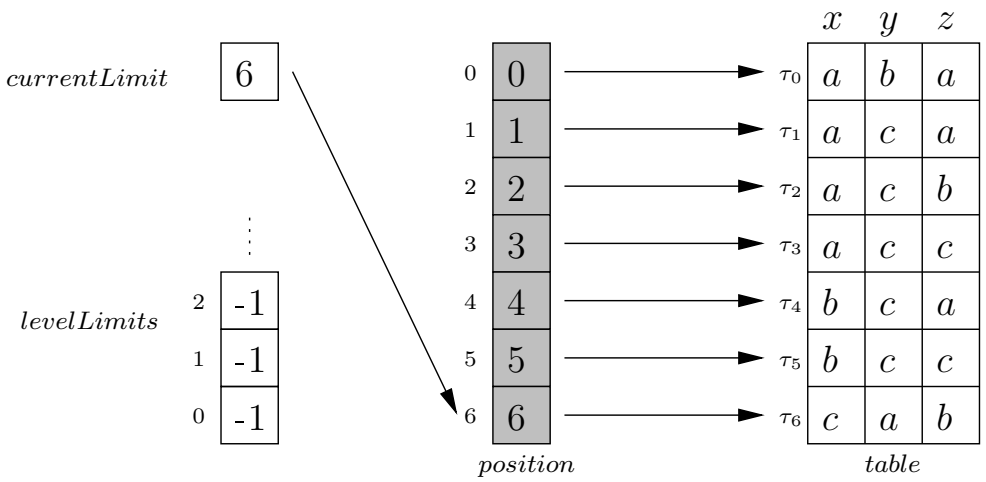


FIG. 1.15 – Les structures STR initialisées pour la contrainte c_{xyz}.

Avant le début de la recherche, tous les tuples autorisés dans c_{xyz} sont valides : la table courante, représentée par la partie grisée dans le tableau $position[c_{xyz}]$, regroupe donc tous les tuples appartenant à $table[c_{xyz}]$. De ce fait, la position du dernier tuple courant stockée dans $currentLimit[c_{xyz}]$ correspond à la position du dernier tuple de $table[c_{xyz}]$. Le tuple référencé par $position[c_{xyz}][0]$ correspond au premier tuple valide dans c_{xyz}, c'est à dire $\tau_0 : (a, b, a)$. La recherche n'ayant pas encore commencé, aucun tuple n'a été supprimé et donc le tableau $levelLimits$ est initialisé à -1. Considérons maintenant que la phase de recherche débute par la décision $y = c$: nous sommes au level 1, c'est à dire lors de l'assignation de la première variable.

L'algorithme STR doit être appliqué à c_{xyz}. Le tableau $position[c_{xyz}]$ est parcouru et les tuples aux différentes positions sont analysés. Le premier tuple considéré τ_0 à la position 0 n'est plus valide : il est alors échangé avec le dernier tuple courant, en l'occurrence le tuple τ_6 à la position $currentLimit[c_{xyz}] = 6$ et $levelLimits[c_{xyz}][1]$ est mis à jour (figure 1.16). De plus, le pointeur $currentLimit[c_{xyz}]$ est décrémenté (figure 1.17). Ensuite, c'est le tuple τ_6 qui est considéré (tout juste échangé, il arrive à présent en tête de $position[c_{xyz}]$). Le tuple τ_6 n'est pas valide. Il est échangé lui aussi avec le dernier tuple valide, en l'occurrence le tuple τ_5 à la position $currentLimit[c_{xyz}]$. Les autres tuples de $table[c_{xyz}]$ sont valides. Le résultat de STR appliqué à la contrainte c_{xyz} après la décision $y = c$ est illustré dans la figure 1.18. À noter que les tuples supprimés au cours du level 1 correspondent aux tuples aux positions allant

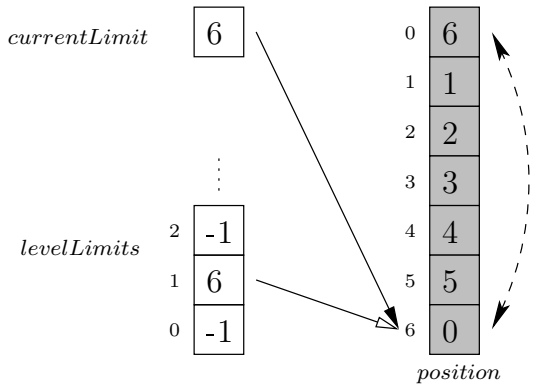


FIG. 1.16 – Inversion dans $position$ des tuples aux positions 0 et 6 et mise à jour de $levelLimits$ à 6 pour le niveau 1.

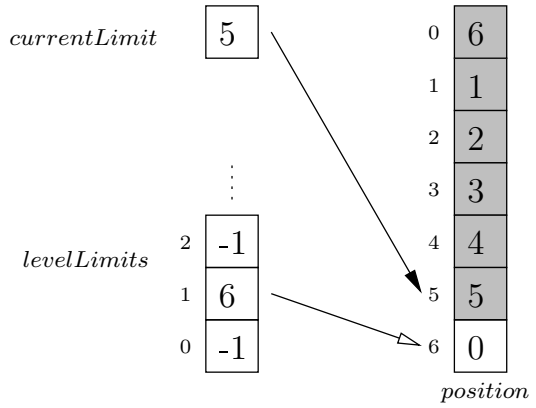


FIG. 1.17 – $currentLimit$ est décrémenté. La table courante ne contient plus que 6 tuples.

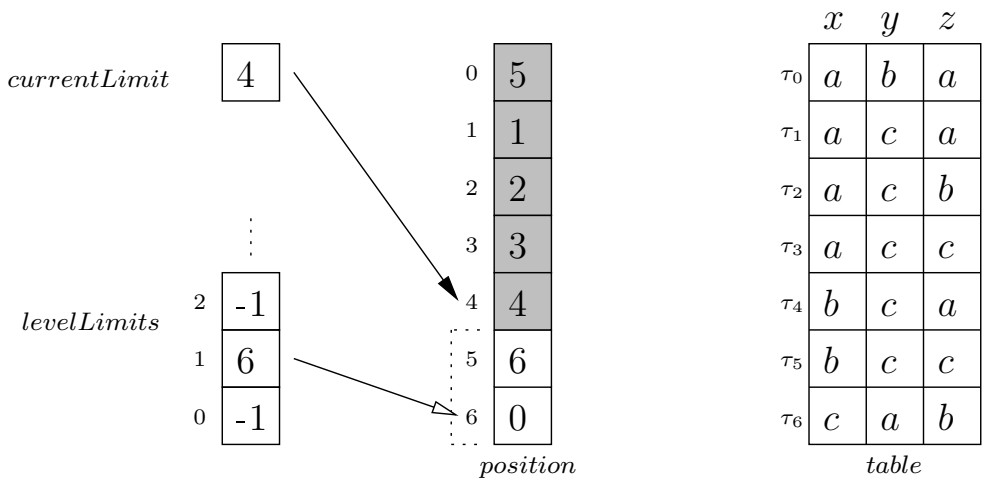


FIG. 1.18 – STR appliqué à la contrainte c_{xyz} après la décision $y = c$.

de $currentLimit[c_{xyz}] + 1$ à $levelLimits[c_{xyz}][1]$, à savoir les tuples des positions 5 à 6 (en pointillés sur la figure).

Après une seconde assignation de variable (level 2), imaginons que l'information $x \neq a$ soit propagée à c_{xyz} depuis une autre contrainte du réseau. Selon le même principe qu'au level 1, les tuples non valides sont échangés avec les tuples valides et la valeur $currentLimit[c_{xyz}]$ est décrémentée. De plus, la valeur (z, b) n'a plus de support : en effet son seul support $\tau_2 = (a, c, b)$ a été supprimé durant l'exécution de l'algorithme STR. Le résultat de cette deuxième exécution est illustré dans la figure 1.19. Imaginons maintenant qu'un retour en arrière d'un level soit réalisé à partir du level 2. La structure $levelLimits[c_{xyz}][2]$ permet de restaurer en temps constant les tuples qui avaient été supprimés lors de ce level : en effet, si on revient en arrière sur la décision ayant amené la propagation de l'information $x \neq a$, ces tuples redeviennent valides : on affecte alors à $currentLimit[c_{xyz}]$ la position stockée dans $levelLimits[c_{xyz}][2]$ pour englober à nouveau ces tuples dans la table courante et la valeur $levelLimits[c_{xyz}][2]$ est remise à -1. L'état des structures STR après un retour en arrière depuis le level 2 est illustré dans la figure 1.20. On peut remarquer à la fin que les tuples valides ne sont pas ordonnés de la même façon qu'ils ne l'étaient lors de l'initialisation. Ce n'est pas important dans la technique STR :

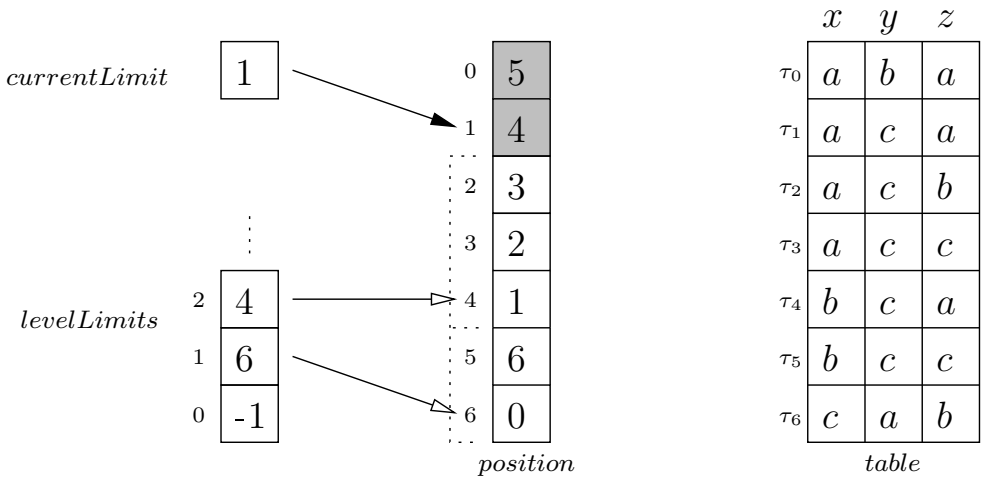


FIG. 1.19 – STR appliqué à la contrainte c_{xyz} après la propagation de $x \neq a$ depuis une autre contrainte. La valeur (z, b) n'ayant plus de support, elle est supprimée du domaine de z.

ce qui est primordial, c'est réellement le partitionnement entre tuples valides et tuples non valides.

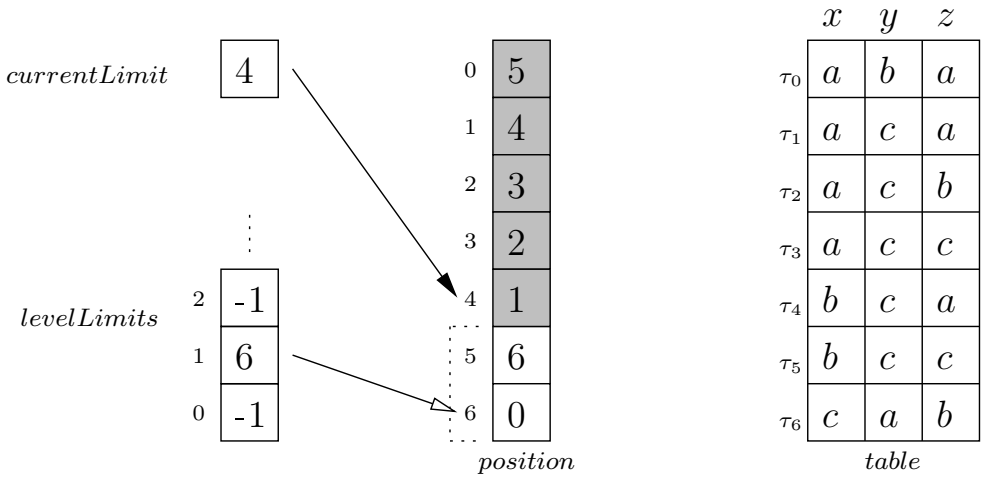


FIG. 1.20 – État des structures STR pour la contrainte c_{xyz} après la restauration conséquente au retour en arrière au level 1.

Une version optimisée de STR appelée STR2 [Lecoutre, 2009, Lecoutre, 2011] a été proposée. Deux remarques pouvant être faites sur STR sont à l'origine de cette nouvelle version :

- Il est inutile de chercher des supports pour les valeurs du domaine d'une variable si toutes les valeurs du domaine ont déjà étés prouvées GAC-cohérentes.
- Après l'exécution de STR sur une contrainte, tous les tuples restants sont des tuples valides. Cela veut dire que tant que le domaine d'une variable n'est pas réduit, les tuples restent valides par rapport à cette variable et donc les tests de validité des valeurs pour cette variable n'est pas utile.

Pour éviter ces recherches de supports et ces tests de validité inutiles, l'algorithme exploite trois structures additionnelles :

- Un ensemble S^{sup} qui contient les variables non assignées dont le domaine contient au moins une valeur pour laquelle aucun support n'a encore été trouvé.

– Un ensemble S^{val} qui contient les variables non assignées dont le domaine a été réduit depuis le dernier appel de STR2.
– Un tableau $lastSize[c][x]$ qui contient pour chaque variable x la taille de son domaine après chaque appel à STR2.

L'exploitation de ces structures pour optimiser STR (algorithme 6) peut être observée dans STR2 (algorithme 10). La structure S^{val} est initialisée avec la dernière variable assignée ($lastAssigned(P)$) si elle appartient à la portée de la contrainte. En effet, cette variable assignée (c'est la seule) doit être contenue dans S^{val} car son domaine a été réduit. La structure S^{sup} est quant à elle initialisée avec toutes les variables non assignées de la portée de la contrainte. Le parcours des tuples courant est identique à STR, cependant le test de validité d'un tuple diffère : en effet, l'algorithme 11 considère uniquement les variables présentes dans S^{val}. Dès qu'il est vérifié que toutes les valeurs du domaine d'une variable sont GAC-cohérentes, alors cette variable est supprimée de S^{sup}. Pour finir, on ne s'intéresse qu'à la recherche d'un support pour les variables appartenant à S^{sup}.

La complexité en temps de STR2 (algorithme 10) pour une contrainte c correspond à $O(r'd + r''t')$ avec r'' le nombre de variables non assignées dans $scp(c)$ pour lesquelles les tests de validité sont nécessaires (taille de l'ensemble S^{val}) et t' la taille de la table courante de c. On y retrouve les complexités respectives $O(r')$, $O(r''t')$ (le test de la validité d'un tuple dans STR2 correspond à $O(r'')$ au lieu de $O(r)$ pour STR) et $O(r'd)$ des trois boucles qui constituent STR2. Tout comme pour STR, la complexité spatiale pour cette même contrainte c dans STR2 est de $O(n + rt)$: la complexité $O(n + rt)$ héritée des structures STR, associée aux complexités spatiales de $O(r)$ pour les structures $lastSize$, S^{sup} et S^{val} (S^{sup} et S^{val} devant être partagées par toutes les contraintes).

1.3 Stratégies de recherche

À ce niveau du manuscrit, nous avons présenté les réseaux de contraintes et différentes techniques d'inférence qui permettent de simplifier un réseau par un processus de déduction : la propagation de contraintes. Il arrive quelques fois que l'inférence appliquée en pré-traitement suffise pour prouver la satisfaisabilité ou l'insatisfaisabilité d'un problème (apparition d'un domaine vide, etc.), mais cela reste rare. Généralement, une phase de recherche est donc nécessaire pour résoudre un réseau de contraintes. On distingue deux types de recherche : recherche complète et recherche incomplète.

Parmi les méthodes de recherche incomplète, la recherche locale consiste à partir d'une instanciation complète initiale et de passer de manière itérative à des instanciations qui sont évaluées meilleures que les précédentes (c'est à dire satisfaisant plus de contraintes). On parle de parcours du voisinage d'une instanciation, chaque instanciation voisine étant obtenue par une transformation plus ou moins légère de l'instanciation courante. La transformation utilisée ainsi que le nombre d'itérations sont paramétrables et très souvent ces critères sont spécifiques aux problèmes et empêchent le développement d'algorithmes de recherche locale génériques efficaces. De plus, les recherches locales peuvent amener à des *minimaux locaux* (instanciations évaluées comme les meilleures parmi celles dans le voisinage considéré) : il faut alors accepter de re-définir ce voisinage et changer de *localité* pour tenter d'atteindre des instanciations encore meilleures (si elles existent). Enfin, on parle de recherche incomplète car rien ne garantit de trouver une solution et le fait de trouver une solution permet de prouver que le réseau est satisfaisable, mais ne pas trouver de solution ne prouve pas l'inverse. Nous ne détaillerons pas plus dans la section CSP les approches incomplètes qui sont hors du contexte de nos contributions, cependant nous y reviendrons plus en détail dans le contexte d'optimisation des CSP pondérés (sections 2.2.3 et 2.5).

Dans le cadre des CSP, une recherche complète de type arborescente est guidée généralement par un schéma de branchement consistant à traverser l'espace de recherche, c'est à dire le produit cartésien des domaines des variables du réseau, en assignant les variables les unes après les autres jusqu'à

Algorithm 10: STR2 ($P = (\mathcal{X}, \mathcal{C})$: CN, c : contrainte) : ensemble de variables

```
1  S^sup ← ∅
2  S^val ← ∅
3  si lastAssigned(P) ∈ scp(c) alors
4      S^val ← S^val ∪ {lastAssigned(P)}
5  pour chaque variable x ∈ scp(c) | x ∉ assigned(P) faire
6      valeursGAC[x] ← ∅
7      S^sup ← S^sup ∪ {x}
8      si |dom(x)| ≠ lastSize[c][x] alors
9          S^val ← S^val ∪ {x}
10         lastSize[c][x] ← |dom(x)|
11 i ← 0
12 tant que i ≤ currentLimit[c] faire
13     index ← position[c][i]
14     τ ← table[c][index]
15     si estUnTupleValide(c, S^val, τ) alors
16         pour chaque variable x ∈ S^sup faire
17             si τ[x] ∉ valeursGAC[x] alors
18                 valeursGAC[x] ← valeursGAC[x] ∪ {τ[x]}
19                 si |valeursGAC[x]| = |dom(x)| alors
20                     S^sup ← S^sup \ {x}
21         i ← i + 1
22     sinon
23         supprimerTuple(c, i, |assigned(P)|)
24 X_reduits ← ∅
25 pour chaque variable x ∈ S^sup faire
26     dom(x) ← valeursGAC[x]
27     si dom(x) = ∅ alors
28         Retourner FAILURE
29     X_reduits ← X_reduits ∪ {x}
30     lastSize[c][x] ← |dom(x)|
31 Retourner X_reduits
```

Algorithm 11: estUnTupleValide (c : contrainte, S^{val} : ensemble de variables, τ : tuple) : Booléen

```
1  pour chaque variable x ∈ S^val faire
2      si τ[x] ∉ dom(x) alors Retourner faux
3  Retourner vrai
```

l'obtention d'une solution où jusqu'à ce que l'espace de recherche soit totalement parcouru. Nous commencerons dans un premier temps par présenter une approche basique et naïve de recherche complète non optimisée appelée *Générer et Tester*. Ensuite, nous présenterons le modèle BPRA (*Branchement Propagation Retour-arrière Apprentissage*) sur lequel sont basés les algorithmes de recherche complète les plus efficaces. Par une utilisation combinée de ses quatre composants, ce modèle propose de mettre en œuvre des techniques pour parcourir efficacement l'espace de recherche (Branchement) tout en filtrant l'espace de recherche durant l'exploration (Propagation) et en effectuant des retours en arrière dès l'apparition d'échecs (Retour-arrière). De plus, bon nombre d'informations pourront être enregistrées (Apprentissage) durant la recherche et exploitées par les autres composants de ce modèle. Durant toute cette section, nous ne parlerons donc plus simplement de méthodes de recherche mais véritablement de méthodes de résolution des CSP.

1.3.1 Méthode Générer et Tester

La méthode *Générer et Tester* est une méthode de résolution complète naïve et non optimisée. Elle consiste comme son nom l'indique à générer toutes les instanciations complètes possibles pour un réseau de contraintes les unes après les autres et de tester si elles sont des solutions pour le réseau. De ce fait, cette méthode permet de trouver toutes les solutions pour un réseau de contraintes, mais également de prouver l'insatisfaisabilité d'un réseau si aucune des instanciations générées n'est solution. L'ensemble des instanciations générées et testées correspond au produit cartésien des domaines des variables du réseau. Il est évident que cette méthode n'est pas du tout efficace.

1.3.2 Le modèle de recherche BPRA

Les algorithmes de résolution complète de CSP les plus efficaces sont basés sur le modèle BPRA (*Branchement Propagation Retour-arrière Apprentissage*) [Jussien *et al.*, 2002, Lecoutre, 2009]. L'efficacité obtenue par la mise en place de ce modèle réside généralement dans une utilisation combinée des différentes techniques proposées par ses composants :

- Branchement : le parcours de l'espace de recherche (branchement binaire, non-binaire, profondeur d'abord, largeur d'abord, etc.) et quelles décisions prendre à chaque étape (heuristiques statiques et dynamiques de choix de variables, valeurs, etc.).
- Propagation : mécanismes d'inférence mis en œuvre, les cohérences plus ou moins fortes établies pour filtrer le réseau après la prise de chaque décision et réduire l'espace de recherche.
- Retour-arrière : la manière, plus ou moins intelligente, de revenir en arrière lorsqu'un échec est atteint durant la recherche arborescente : dernière décision prise ou décision plus ancienne et réellement responsable de l'échec, etc.
- Apprentissage : les informations enregistrées durant la propagation (valeurs supprimées) et lorsqu'un échec est rencontré (nogood), puis la manière de les exploiter durant le branchement, la propagation et le retour en arrière.

Nous présentons plus en détail dans la suite de cette section différentes techniques des composants Branchement, Propagation et Retour-arrière qui sont généralement utilisées. Nous avons fait le choix de ne pas nous étendre sur le composant Apprentissage car, même s'il s'agit d'une partie importante dans la résolution des CSP, il représente plutôt un composant complémentaire aux autres et nous avons pensé qu'il n'était pas indispensable de l'aborder plus en détail dans le cadre de cette thèse. Nous verrons tout de même dans la section 1.3.4 que la technique basée sur les conflits pour un retour en arrière plus pertinent exploite des informations apprises durant le mécanisme de propagation, en l'occurrence les explications des différentes suppressions de valeurs (conflits). Il est donc important de souligner que

différentes techniques d'apprentissage existent et qu'elles sont utiles pour améliorer la résolution des CSP. De plus amples informations sont accessibles depuis les ouvrages de référence : mémorisation de nogoods [Dechter, 1990, Frost et Dechter, 1994, Lecoutre *et al.*, 2007], gestion des explications de conflits [Prosser, 1993, Ginsberg, 1993] présentée dans la section 1.3.4, etc.

1.3.3 Branchement et heuristiques pour orienter les choix

Recherche arborescente par branchement binaire

Deux schémas principaux de branchement existent : le branchement binaire (appelé aussi 2-*way branching*) et le branchement non-binaire (appelé aussi d-*way branching*). À chaque étape d'un branchement non-binaire, une variable non assignée est choisie et toutes les assignations de cette variable par les valeurs de son domaine sont considérées (d-*way branching* car *d* branches possibles pour une variable associée à un domaine de taille *d*). Concernant le branchement binaire, une variable non assignée ainsi qu'une valeur de son domaine sont choisies à chaque étape et l'affectation puis la réfutation de cette valeur pour cette variable sont considérées (2-*way branching* car exactement 2 branches possibles). À noter que ces deux schémas garantissent une exploration complète de l'espace de recherche et que le branchement binaire a été montré théoriquement dans [Hwang et Mitchell, 2005] comme le plus efficace pour prouver l'insatisfaisabilité de réseaux dans le cadre des CSP. C'est naturellement celui-ci qui a été suivi dans le cadre de nos contributions et que nous présentons ici. Ensuite, plusieurs techniques d'exploration de l'arbre de recherche existent : en profondeur d'abord (depth-first search), en largeur d'abord (breadth-first search), etc. Nous considérons ici une approche basée sur une exploration en profondeur d'abord qui représente, par rapport à une exploration en largeur d'abord, un meilleur compromis entre espace mémoire utilisé et efficacité pour trouver (rapidement) une solution.

La figure 1.21 représente l'arbre de recherche binaire partiel pour le problème de coloriage de carte présenté en section 1.1.2. La racine de l'arbre (nœud noirci) représente le réseau de contraintes P au début de la résolution. Chaque branche de l'arbre correspond à une décision. Généralement, les branches gauches de l'arbre représentent des décisions positives ($x_0 = n$), tandis que les branches droites représentent des décisions négatives ($x_0 \neq n$). Chaque nœud de l'arbre représente le réseau modifié (et simplifié) par la prise en compte d'une part de l'ensemble des décisions qui ont amené à ce nœud, puis par la propagation des contraintes d'autre part. Par exemple, le nœud grisé représente le réseau $P|_{x_0=n,x_1\neq gf,x_2\neq gi}$ obtenu à partir de P après les décisions $x_0 = n$, $x_1 \neq gf$ et $x_2 \neq gi$. Dans le cadre d'un branchement binaire, la réfutation d'une valeur pour une variable peut être suivie par l'affectation d'une valeur à une autre variable ($x_1 \neq gf$ suivie de $x_2 = gi$), ce qui donne de la flexibilité à l'exploration.

Le branchement binaire n'est pas suffisant en lui-même pour optimiser la résolution d'un CSP. Encore faut-il prendre les bonnes décisions : dans quel ordre assigner les variables et pour chacune de ces variables, dans quel ordre lui affecter les valeurs de son domaine ? Clairement, les bonnes décisions sont celles qui vont permettre soit de se diriger très rapidement vers une solution, soit de se diriger très rapidement vers un échec et ainsi éviter de parcourir inutilement des zones de l'espace de recherche. Il existe des heuristiques qui guident les choix de variables et de valeurs pour un parcours plus ou moins rapide de l'espace de recherche en fonction du problème considéré.

Heuristiques statiques, non adaptatives dynamiques et adaptatives de choix de variable

On distingue trois catégories d'heuristiques de choix de variables : statiques, non adaptatives dynamiques et adaptatives. Le type d'une heuristique dépend de la nature des informations utilisées pour estimer et évaluer parmi les variables celles permettant d'améliorer les performances de parcours de l'espace de recherche. Cependant, la plupart des heuristiques de choix de variable sont basées sur le principe

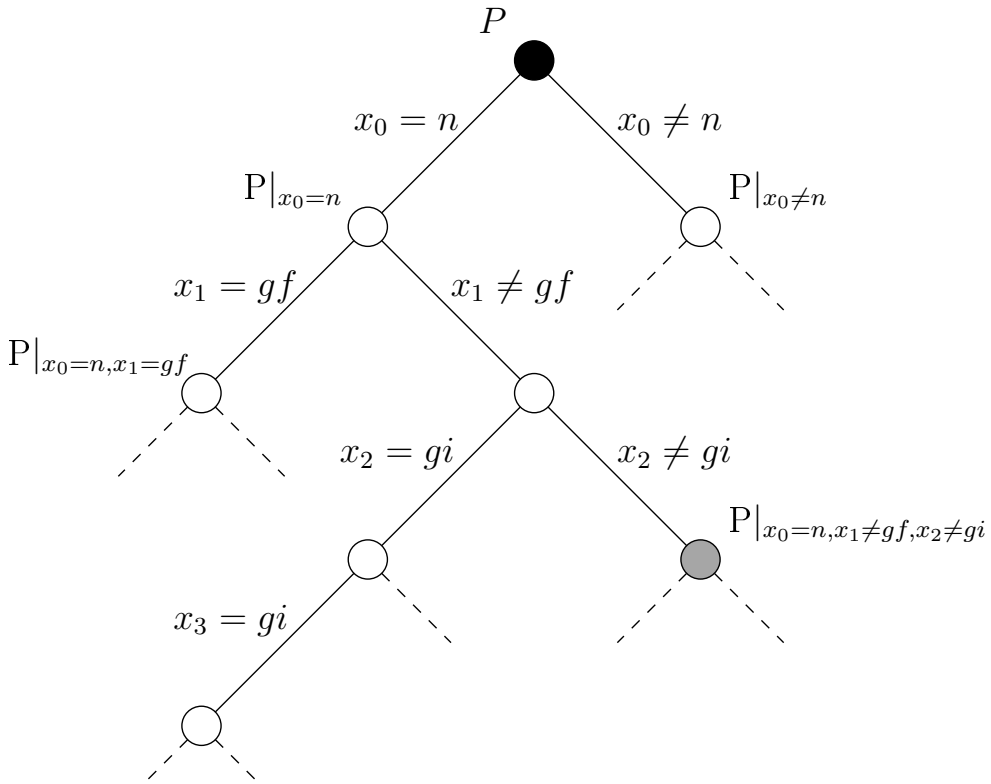


FIG. 1.21 – Arbre de recherche partiel avec schéma binaire pour le problème de coloriage de carte.

de *fail-first* émanant du célèbre "Pour réussir, essaies d'abord là où tu as le plus de chance d'échouer" [Haralick et Elliott, 1980]. En d'autres termes, la tendance de ces heuristiques est de choisir d'assigner en priorité les variables ayant le plus de chance de mener à un échec et ainsi ne pas trop s'enfoncer dans l'arbre de recherche. Nous présentons brièvement ces différentes catégories ainsi que certaines d'entre elles.

Les heuristiques statiques, ou catégorie *SVO* pour Static Variable Ordering, sont des heuristiques qui exploitent l'état initial du réseau à résoudre pour ordonner les variables : domaine initial et degré initial des variables, etc. Durant toute la recherche, cet ordre reste inchangé malgré les simplifications et modifications du réseau qui peuvent se produire.

- *lexico* : les variables sont choisies par ordre lexicographique de leurs noms, c'est à dire par exemple que la variable nommée x_i sera choisie avant la variable nommée x_j pour tout $i < j$. L'intérêt de cette heuristique réside uniquement dans la manière dont a été modélisé le problème : si une réflexion particulière a été menée et a abouti à nommer d'abord les variables les plus contraintes du réseau, alors cette heuristique est intéressante. Sinon, elle peut être utilisée en complément d'une heuristique plus pertinente afin de casser l'égalité (tout comme l'heuristique *random* consistant à choisir une variable aléatoirement).
- *deg* (*maxdeg*) [Dechter et Meiri, 1989] : les variables sont choisies par ordre décroissant de leur degré initial. Il est naturel de penser que les variables impliquées dans le plus grand nombre de contraintes correspondent aux variables les plus contraintes et intuitivement celles ayant le plus de chance de mener à un échec. C'est tout l'intérêt de cette heuristique.

Les heuristiques (non adaptatives) dynamiques, ou catégorie *DVO* pour Dynamic Variable Ordering, sont des heuristiques qui exploitent l'état courant du réseau à résoudre pour ordonner les variables : do-

maine courant et degré dynamique (courant) des variables, etc. Le degré dynamique d'une variable est le nombre de contraintes dans lesquelles elle est impliquée et dont la portée contient au moins une autre variable qui ne soit pas singleton. Une critique pouvant être faite aux heuristiques statiques est le fait qu'elles exploitent la structure initiale du réseau alors que celui-ci évolue tout au long de la recherche : réductions des domaines des variables, degrés des variables qui diminuent lorsque des contraintes deviennent universelles, etc. En d'autres termes, une variable qui était considérée prioritaire au début de la recherche l'est peut être moins à un autre instant de la recherche. Il est indéniable que prendre des décisions basées sur l'état courant d'un réseau améliore les performances de résolution.

- *dom* (*mindom*) [Haralick et Elliott, 1980] : les variables sont choisies par ordre croissant de la taille de leur domaine courant. L'idée est que si une variable contient peu de valeurs dans son domaine, cela signifie que le nombre d'assignations possibles à considérer pour cette variable est faible et qu'ainsi donc un échec éventuel peut être détecté plus tôt.
- *ddeg* : les variables sont choisies par ordre décroissant de leur degré dynamique. Le principe est le même que pour l'heuristique statique sauf que la déduction est basée ici sur les degrés courants.
- *dom/ddeg* [Bessiere et Régin, 1996] : les variables sont choisies par ordre croissant du ratio entre la taille du domaine courant et le degré dynamique. À noter qu'une variante appelée *dom/deg* existe et qu'elle est basée sur le ratio entre le domaine courant et le degré initial.
- *dom+ddeg* ou *Brelaz* [Brélaz, 1979] : les variables sont choisies par ordre croissant de la taille de leur domaine courant, et en cas d'égalité elles sont choisies par ordre croissant de leur degré courant. Cette heuristique a pour but de casser les égalités de taille de domaines.

Les heuristiques adaptatives sont des heuristiques qui exploitent d'une part la structure courante du réseau et d'autre part les enseignements de ce qui a été fait précédemment : on parle d'adaptation par rapport à l'expérience acquise durant la recherche.

- *wdeg* [Boussemart *et al.*, 2004] : l'idée de cette heuristique est d'associer à chaque contrainte un poids, initialisé à 1, et de l'incrémenter dès que cette contrainte est violée durant la recherche. L'intuition ici est que plus une contrainte est violée, plus elle doit être difficile à satisfaire : plus le poids d'une contrainte est élevé, plus cette contrainte est dure. Ainsi, on va parler de degré pondéré pour les variables : il correspond pour une variable à la somme des poids associés aux contraintes impliquant cette variable et possédant dans sa portée au moins une variable non singleton. L'heuristique consiste donc à considérer les variables par ordre décroissant de leur degré pondéré. Le fait de se concentrer sur les contraintes les plus difficiles à satisfaire va permettre aussi de se focaliser sur des noyaux de contraintes difficiles et conduire à une détection précoce d'éventuels noyaux insatisfaisables.
- *dom/wdeg* : les variables sont choisies par ordre croissant du ratio entre la taille de leur domaine courant et leur degré pondéré.
- *impact* [Refalo, 2004] : cette heuristique est basée sur le concept d'impact d'une variable. L'impact d'une variable dans un réseau correspond au nombre de valeurs supprimées dans les domaines des (autres) variables de ce réseau (en d'autres termes, l'ampleur de la réduction de l'espace de recherche) suite à l'assignation de cette variable et à la propagation des contraintes. Plus précisément, l'impact d'une variable correspond à la somme des impacts de chaque assignation de cette variable par une valeur de son domaine. Les différents impacts mesurés durant la recherche pour une variable sont enregistrés et, au niveau du nœud courant, ce sont les impacts moyens de ces variables qui sont exploités pour le choix de la variable à assigner, c'est à dire les moyennes des différents impacts mesurés pour ces variables durant la recherche jusqu'au nœud courant. Les variables sont choisies par ordre décroissant de leur impact moyen.
- *last conflict* [Lecoutre *et al.*, 2006] : cette heuristique consiste à s'intéresser aux décisions qui ont conduit à un échec. Plus précisément, le principe est de choisir en priorité la variable ayant mené au dernier échec rencontré. Clairement, l'objectif de cette heuristique est de guider la recherche

vers la source des conflits.

Heuristiques de choix de valeur

Juxtaposées aux heuristiques de choix de variable, la plupart des heuristiques de choix de valeur sont basées sur le principe de *succeed-first* [Smith, 1996]. En d'autres termes, la tendance de ces heuristiques est d'affecter en priorité aux variables les valeurs ayant le plus de chance de mener à une solution.

- *lexico* : les valeurs sont choisies selon l'ordre dans lequel elles se trouvent dans le domaine courant de la variable. Comme pour les variables, l'heuristique *lexico* permet d'effectuer des choix pertinents s'il y a eu un effort qui a été fait dans ce sens lors de la modélisation du problème, mais généralement elle est utilisée uniquement pour casser les égalités (tout comme l'heuristique *random* consistant à choisir une valeur aléatoirement).
- *min-conflits* [Minton *et al.*, 1992, Frost et Dechter, 1995] : les valeurs sont choisies par ordre croissant du nombre total de conflits qui leur est associé. Pour une valeur, le nombre total de conflits correspond à la somme du nombre de conflits (valeurs non supports) pour cette valeur présentes dans les domaines courants des variables voisines à cette variable.
- *impact* : les valeurs sont choisies par ordre croissant de leur impact, c'est à dire l'impact de leur affectation à la variable associée.

Pour conclure, il est évident que chacune de ces heuristiques fonctionnera plus ou moins efficacement en fonction du problème traité, les spécificités de chacune d'entre elles et des comparaisons pouvant être consultées dans les différentes publications associées et référencées précédemment.

1.3.4 Retour en arrière : techniques de look-back

Lors du parcours de l'arbre de recherche, il peut arriver de rencontrer un échec : instanciation partielle courante incohérente ou ne pouvant être étendue par au moins une variable sans violation d'une contrainte, ou encore apparition d'un domaine vide pour une variable lorsqu'une cohérence est appliquée sur le réseau courant comme nous le verrons dans la section 1.3.5. Il est alors évident qu'étendre une telle instanciation partielle en solution est impossible. Au lieu de continuer d'avancer, on effectue alors un retour en arrière dans l'arbre afin d'éviter de parcourir toute zone de l'espace de recherche que l'on sait stérile. Différentes approches dites techniques de *look-back* [Dechter et Frost, 2002] existent et constituent une approche rétrospective (on regarde vers l'arrière) visant à améliorer le parcours de l'espace de recherche. D'un point de vue général, ces techniques sont basées sur l'idée "Souviens toi de ce que tu as fait pour éviter de répéter les mêmes erreurs" [Haralick et Elliott, 1980]. On cherche à identifier pour chacune des valeurs supprimées du domaine d'une variable l'explication à l'origine de cette suppression. En effet, l'idée est de déterminer la ou les décisions anciennes déjà prises et qui sont responsables de l'échec rencontré, d'y revenir et de les réparer en prenant d'autres décisions. Ne pas prendre en compte les décisions qui ont amené cet échec n'empêchera pas de retomber dessus une prochaine fois. Nous présentons les trois techniques de retour en arrière les plus connues.

Retour en arrière standard (SBT pour Standard BackTracking)

Le retour en arrière standard, appelé aussi retour en arrière chronologique, est le retour en arrière de base. Lors d'un échec, il consiste à revenir dans l'état avant la dernière décision positive prise, considérée alors comme la décision coupable de l'impasse. Les structures du réseau modifiées suite à cette décision coupable sont restaurées et on passe à la décision venant après cette décision coupable selon le branchement et les heuristiques utilisées. Considérons l'exemple basé sur le problème de coloriage de carte introduit dans la sous-section 1.1.2 et illustré à la figure 1.22. Les assignations $x_0 = n$ et

$x_1 = n$, qui ont amené au réseau représenté par le nœud grisé et dans un état où $dom(x_0) = \{n\}$ et $dom(x_1) = \{n\}$, génèrent une incohérence car la contrainte $c_{x_0x_1}$ est violée. La dernière décision $x_1 = n$ est considérée comme la décision coupable et le retour en arrière standard revient juste avant cette décision. De ce fait, le domaine $dom(x_1)$ impacté par cette dernière décision est restauré et correspond à $dom(x_1) = \{n, gf, gi, gc\}$ (en effet, la décision $x_1 = n$ ayant entrainé la suppression des valeurs gf, gi et gc de $dom(x_1)$, ces valeurs doivent être ré-intégrées), puis on passe à la décision suivante $x_1 \neq n$.

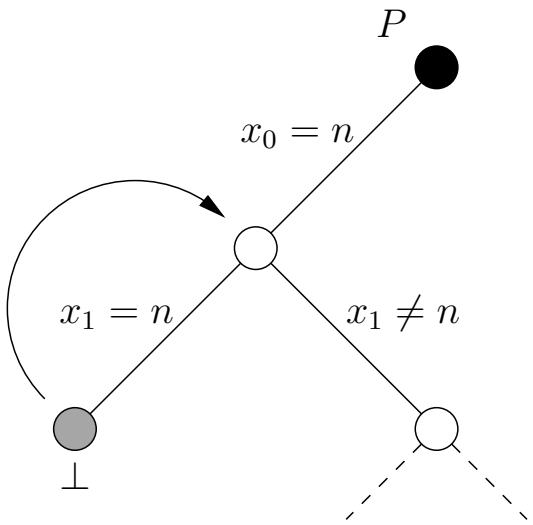


FIG. 1.22 – Apparition d'un échec après les assignations $x_0 = n$ et $x_1 = n$ et retour en arrière standard dans le problème de coloriage de carte.

Saut en arrière basé sur les conflits (CBJ pour Conflict-directed BackJumping) [Prosser, 1993]

Le retour en arrière standard est limité dans le sens où la dernière décision prise n'est pas forcément la cause de l'échec rencontré, et dans ce cas revenir juste avant cette décision n'empêchera pas de tomber à nouveau dans la même impasse et de faire du *thrashing* (reproduire les mêmes erreurs constamment). La technique de saut en arrière basée sur les conflits, appelée aussi retour en arrière intelligent, consiste donc à s'intéresser plus précisément aux décisions qui ont conduit à cet échec. Pour chaque suppression de valeur (conflit) d'un domaine, une explication d'élimination est enregistrée : il s'agit des décisions à l'origine de cette suppression. Lorsqu'un échec est rencontré, l'ensemble des explications d'élimination à l'origine de l'échec est analysé et le retour en arrière est effectué au niveau de la décision la plus récente (la plus profonde dans l'arbre) parmi ces explications.

Afin d'illustrer cette technique de retour en arrière, nous allons considérer dans un exemple le réseau de contrainte $P : \{\mathcal{X}, \mathcal{C}\}$ défini par $\mathcal{X} = \{w, x, y, z\}$ avec $dom(w) = dom(x) = dom(y) = dom(z) = \{a, b\}$ et $\mathcal{C} = \{c_{wz}, c_{xz}\}$ avec $c_{wz} : \{w \neq z\}$, $c_{xz} : \{x \neq z\}$. Soit $\{w = a, x = b, y = a\}$ l'instanciation partielle courante illustrée dans la figure 1.23. Quand l'algorithme de recherche tente d'étendre cette instanciation par l'assignation de la variable z, un échec est rencontré car aucune valeur du domaine de z n'est compatible avec l'instanciation partielle ce qui a pour conséquence l'apparition d'un domaine vide pour z. En effet, l'assignation $z = a$ viole la contrainte c_{wz} et l'assignation $z = b$ viole la contrainte c_{xz}. Il faut donc effectuer un retour en arrière. Effectuer un retour en arrière standard consisterait à revenir à la dernière décision, c'est à dire $y = a$. Cependant, si on prend effectivement la nouvelle décision $y \neq a$ et plus précisément $y = b$, on s'aperçoit que l'on rencontre la même impasse : en effet, aucune valeur dans

le domaine de z n'est compatible avec l'instanciation partielle courante $\{w = a, x = b, y = b\}$. Ceci est dû au fait que la variable y n'est pas à l'origine de conflits avec la variable z. Considérons maintenant un saut en arrière basé sur les conflits. Lors de la recherche, les explications d'élimination des valeurs des domaines de z sont enregistrées (illustrées sur la figure 1.23 par une flèche avec le libellé *explication*) : $z \neq a$ (ou élimination de a dans $dom(z)$) est expliqué par la décision $w = a$ pour satisfaire la contrainte c_{wz}, puis $z \neq b$ est expliqué par la décision $x = b$ pour satisfaire la contrainte c_{xz}. Le domaine vide obtenu pour z correspond alors à l'explication globale $\{w = a, x = b\}$. Un saut en arrière basé sur les conflits ayant amené à $dom(z) = \emptyset$ consiste à revenir à la décision la plus récente dans les explications, à savoir dans l'exemple la décision $x = b$ (figure 1.24), et ne plus considérer le parcours effectué depuis cette décision : cette portion de l'espace de l'arbre de recherche (hachurée sur la figure) peut être coupée.

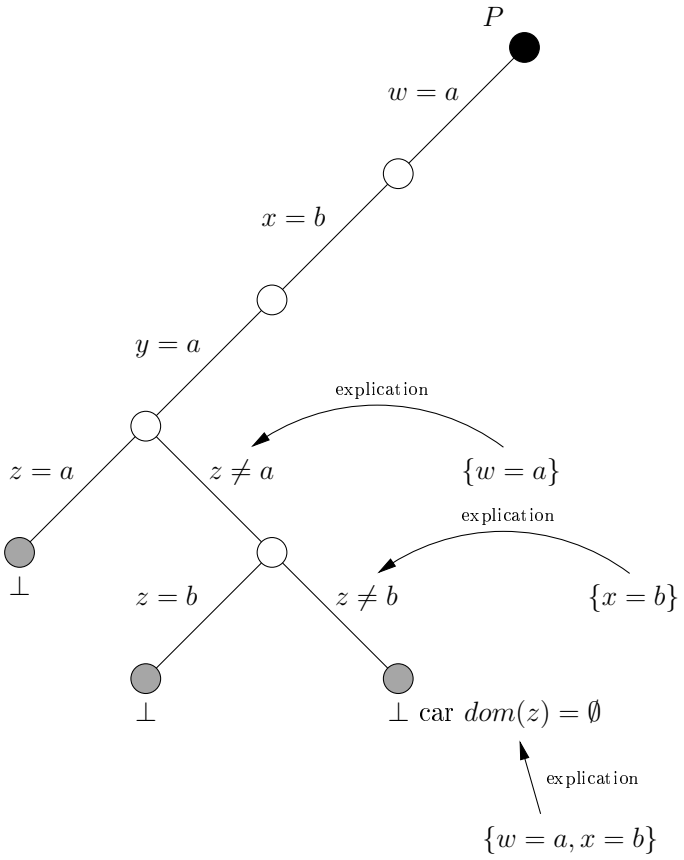


FIG. 1.23 – Apparition d'un échec avec les assignations $z = a$ et $z = b$ car aucune valeur compatible ($dom(z) = \emptyset$) avec l'instanciation partielle $\{w = a, x = b, y = a\}$.

À noter que l'heuristique adaptative *last conflict*, présentée dans la section 1.3.3, simule d'une certaine manière cette technique de saut en arrière basée sur les conflits.

Retour en arrière dynamique (DBT pour Dynamic BackTracking) [Ginsberg, 1993]

Le retour en arrière dynamique exploite également les explications d'élimination de valeurs pour déterminer la décision dite coupable sur laquelle revenir. Cependant, contrairement au saut en arrière basé sur les conflits, le retour en arrière dynamique supprime la décision coupable tout en conservant les autres décisions qui ont été prises. Le mécanisme est illustré à la figure 1.25. Tout comme pour le saut en arrière, les explications sont enregistrées et lorsque l'échec est rencontré ($dom(z) = \emptyset$), la décision

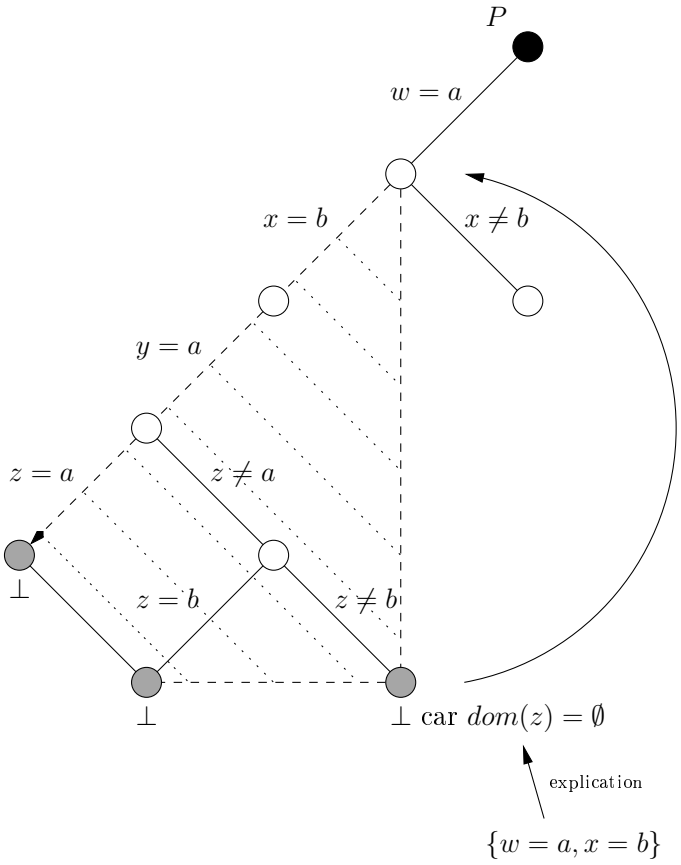


FIG. 1.24 – Saut en arrière basé sur les conflits à l'origine de $dom(z) = \emptyset$.

désignée coupable est $x = b$. Cette décision est supprimée, les autres décisions sont conservées et la décision $x \neq b$ est ajoutée comme décision la plus récente.

1.3.5 Propagation : techniques de look-ahead

Après avoir présenté des méthodes de *look-back* pour améliorer les phases de retour en arrière lors de la résolution d'un CSP, nous allons nous intéresser maintenant aux méthodes dites *look-ahead* [Dechter, 2003] ou approches prospectives (on regarde vers l'avant) qui vont permettre d'améliorer les phases d'exploration. Basées sur l'idée de "Prévoir et anticiper le futur afin de réussir dans le présent" [Haralick et Elliott, 1980], ces techniques d'inférence consistent à évaluer pour un réseau les conséquences d'une décision, c'est à dire l'impact d'une instanciation partielle courante sur les domaines des variables qui ne sont pas encore assignées. Après une décision, c'est à dire à chaque nœud de l'arbre de recherche, une cohérence plus ou moins forte est établie/maintenue sur le réseau et les valeurs localement incohérentes sont détectées et supprimées. Ces techniques permettent donc de conserver après filtrage uniquement les valeurs localement cohérentes et ainsi réduire potentiellement la taille de l'espace de recherche et les branches à explorer. À noter également que le filtrage est effectué durant la recherche après chaque décision, mais il est possible également d'établir une cohérence sur le réseau avant le début de la recherche : on appelle ça du filtrage en pré-traitement. Le but de cette opération est de simplifier le réseau avant toute exploration par suppression de toutes les valeurs localement incohérentes selon cette cohérence et par chance directement prouver soit la satisfaisabilité du réseau (domaine singleton pour chaque variable), soit son insatisfaisabilité (apparition d'un domaine vide pour une variable). Nous

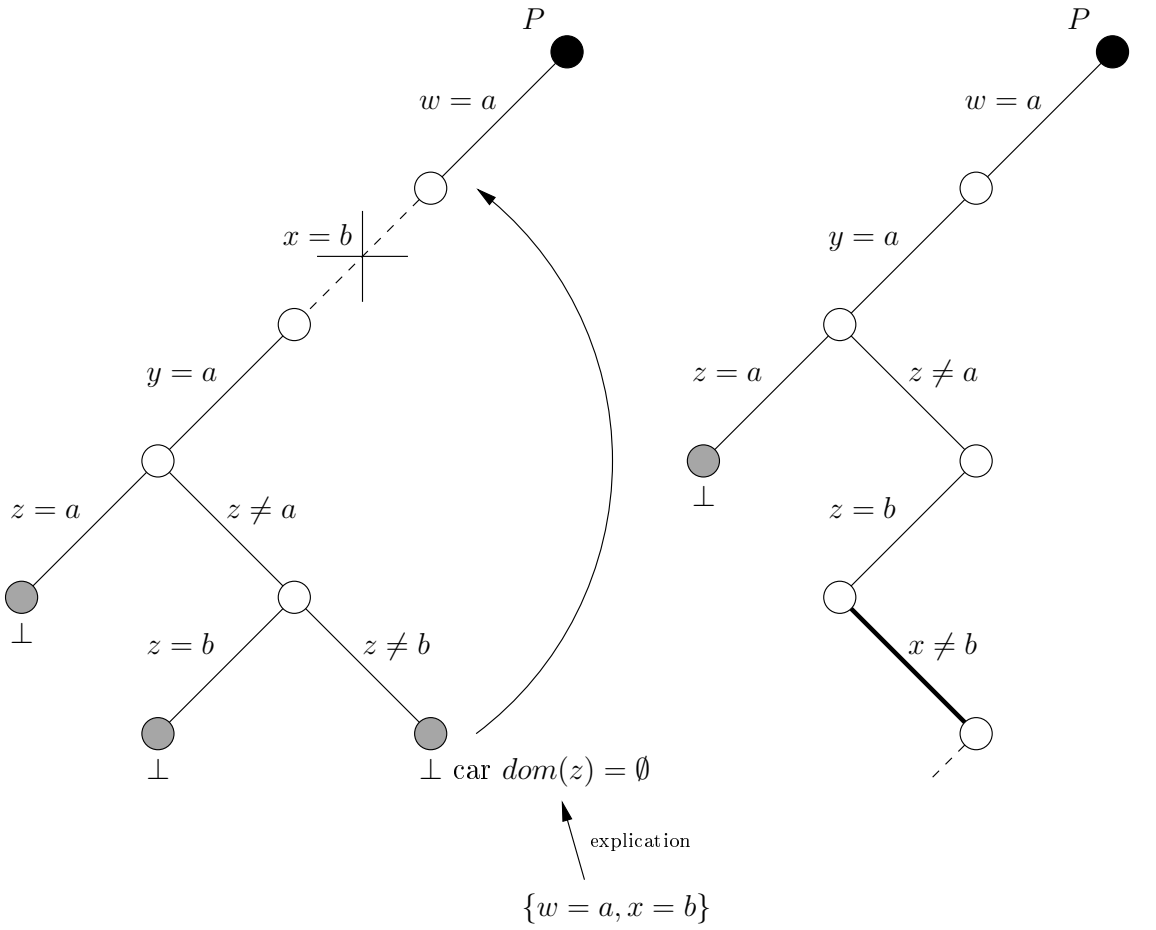


FIG. 1.25 – Retour en arrière dynamique basé sur les explications des éliminations à l'origine de $dom(z) = \emptyset$.

n'aborderons pas plus ici cette phase de pré-traitement. Nous présentons donc les trois techniques de look-ahead les plus connues. Nous considérons dans les explications suivantes que ces techniques de *look-ahead* sont incorporées dans un algorithme de recherche utilisant le retour en arrière standard.

Backward checking (BC)

Le *backward checking*, ou test de cohérence par l'arrière, est la technique de *look-ahead* de base. Elle consiste tout simplement à vérifier après l'affectation d'une valeur à une variable que l'instanciation partielle courante obtenue suite à cette nouvelle assignation est cohérente, c'est à dire que toutes les contraintes couvertes par cette instanciation sont satisfaites. Un exemple de *backward checking* est illustré à la figure 1.22. Suite à l'assignation $x_1 = n$, un test de cohérence est réalisé sur l'instanciation $\{x_0 = n, x_1 = n\}$. Une incohérence est détectée car la contrainte $c_{x_0x_1}$ est violée. On revient en arrière et on passe à la décision suivante qui est $x_1 \neq n$: la valeur n est supprimée du domaine de x_1.

Forward checking (FC ou partial look-ahead) [Haralick et Elliott, 1980]

Le *forward checking* applique une forme partielle d'arc-cohérence sur le réseau après chaque décision. On parle d'arc-cohérence partielle, ou *partial look-ahead*, dans le sens où seules les variables

voisines de la variable qui vient d'être assignée vont être révisées. Considérons à nouveau le problème de coloriage de carte et le parcours de recherche illustré à la figure 1.26. Une première décision $x_0 = n$ est effectuée. Un premier filtrage par *forward checking* est effectué et a pour conséquence la révision des variables voisines de x_0, à savoir les variables x_1 et x_3 pour les contraintes $c_{x_0x_1}$ et $c_{x_0x_3}$. Les valeurs en conflit avec l'assignation $x_0 = n$ sont supprimées des domaines de x_1 et x_3 : ici la valeur n est supprimée de $dom(x_1)$ et de $dom(x_3)$. Une deuxième décision $x_1 = gf$ est prise et après un deuxième *forward checking*, ce sont les domaines des variables voisines à x_1, à savoir x_2 et x_3, qui sont révisés. Puis la décision $x_2 = gi$ entraîne la révision des variables x_3 et x_6 et plus précisément on s'aperçoit alors que le domaine de x_3 ne contient plus que la valeur gc. Lorsque le *forward checking* est appliqué au réseau après la décision $x_6 = gc$, on tombe dans une impasse car la révision des variables voisines de x_6 a entrainé l'apparition d'un domaine vide, en l'occurrence le domaine de x_3. Un retour en arrière est donc effectué.

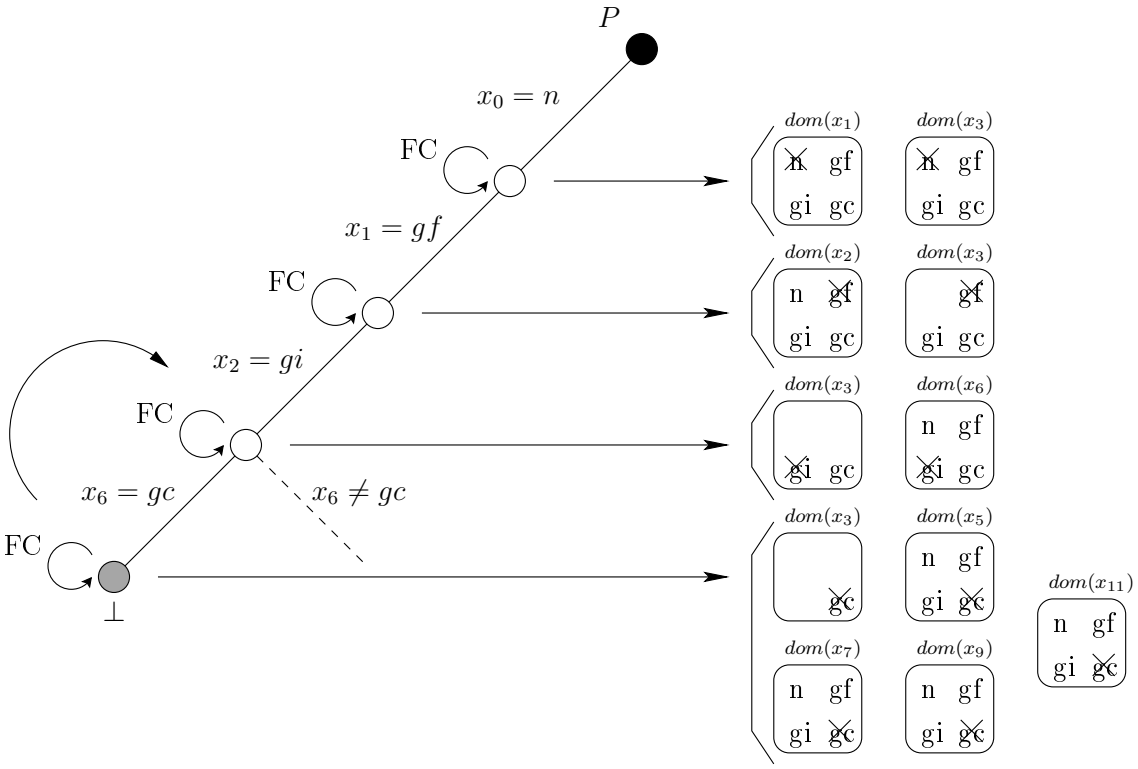


FIG. 1.26 – Forward checking appliqué après chaque décision et évolution des domaines des variables dans le problème de coloriage de carte. Apparition d'un domaine vide ($dom(x_3)$) suite à la décision $x_6 = gc$ et retour en arrière standard.

MAC (Maintien de la cohérence d'arc ou full look-ahead) [Sabin et Freuder, 1994]

On parle de MAC pour maintien de l'arc-cohérence pendant la recherche. À la différence du *forward checking*, l'arc-cohérence va être établie pour l'ensemble du réseau et pas seulement pour les variables voisines de la variable qui vient juste d'être assignée. Considérons à nouveau l'exemple du problème de coloriage de carte et cette fois-ci le parcours de recherche illustré à la figure 1.27. L'arc-cohérence (AC) est maintenue à chaque nœud de l'arbre après chaque décision. Les deux premières décisions $x_0 = n$ et $x_1 = gf$ entraînent la révision des domaines de leurs variables voisines, mais aucune déduction

(propagation de contraintes) ne peut être effectuée à partir de ces révisions pour les autres variables du réseau. Pour la décision $x_2 = gi$, c'est tout à fait différent. Dans un premier temps, l'arc-cohérence appliquée au réseau après cette décision filtre les domaines des variables voisines de x_2 en supprimant la valeur gi des domaines de x_3 et x_6. Dans un deuxième temps, la propagation de contraintes filtre les domaines des variables x_4, x_5 et à nouveau x_6 suite au fait qu'il reste uniquement la valeur 3 dans le domaine de x_3. Le fait que le domaine de x_3 ne contienne plus que la valeur gc et que par propagation de contraintes cette valeur ait été supprimé du domaine de x_6, on voit que la décision suivante qui est prise pour x_6, à savoir $x_6 = gf$, évite d'avoir l'apparition d'un domaine vide qui n'avait pas été anticipée avec l'utilisation du *forward checking* pendant la recherche dans la figure 1.26. En d'autres termes, maintenir l'arc-cohérence durant la recherche permet de mieux anticiper que le *forward checking*. Cependant, la démarche est plus coûteuse car elle nécessite un nombre plus important de révisions de filtrage.

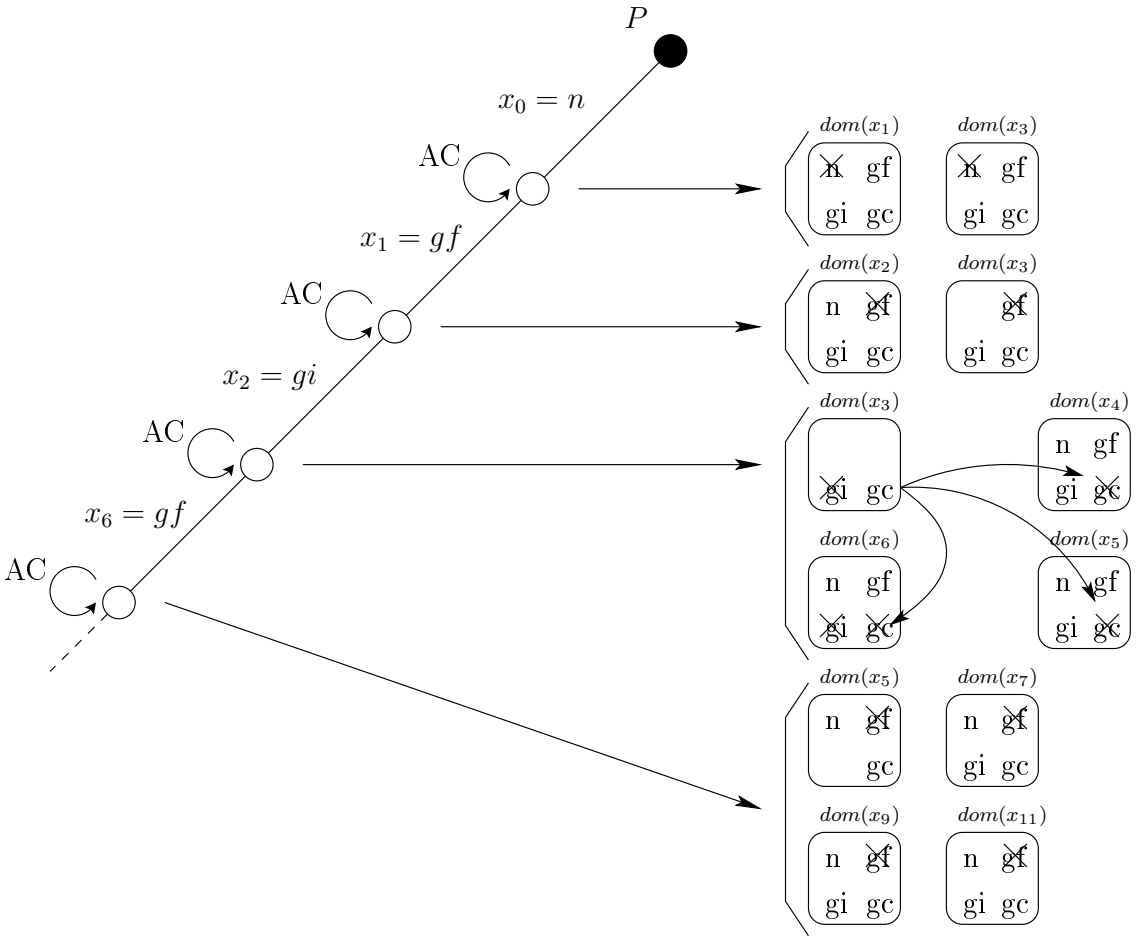


FIG. 1.27 – Maintien de AC après chaque décision et évolution des domaines des variables dans le problème de coloriage de carte. Le phénomène de propagation de contraintes se déclenche suite à la décision $x_2 = 2$.

Pour conclure cette partie concernant les stratégies de recherche, comme il a été dit en début de section, l'efficacité d'un algorithme de résolution de CSP dépend de la manière de combiner les différentes techniques proposées par les composants du modèle BPRA (*Branchement Propagation Retour-arrière Apprentissage*), plus précisément la manière d'associer techniques de *look-back* et techniques de *look-ahead*. Sans avoir la prétention de présenter en détail les avantages et les inconvénients de toutes les

combinaisons possibles de ces techniques, il est intéressant de connaître les associations qui constituent les algorithmes les plus connus (généralement les plus efficaces) pour la résolution des CSP. Ces algorithmes sont résumés dans le tableau 1.1.

Algorithme	Technique de look-ahead	Technique de look-back	auteur
BT	BC	SBT	-
CBJ	BC	CBJ	[Prosser, 1993]
DBT	BC	DBT	[Ginsberg, 1993]
FC	FC	SBT	[Haralick et Elliott, 1980]
MAC	MAC	SBT	[Sabin et Freuder, 1994]

TAB. 1.1 – Différentes associations des techniques de look-ahead et de look-back présentées.

Chapitre 2

Le problème de satisfaction de contraintes pondérées WCSP

Sommaire

Le cadre CSP permet de modéliser et de résoudre un grand nombre de problèmes. Cependant, d'autres problèmes demandant plus de souplesse ne peuvent pas être modélisés uniquement dans le cadre CSP, c'est à dire uniquement sous la forme d'instanciations strictement autorisées ou strictement interdites par les contraintes couvertes. De nombreuses extensions du cadre CSP ont été proposées pour proposer cette flexibilité et permettent d'appréhender divers facteurs : prise en compte d'incertitude et de probabilité (CSP probabiliste [Fargier et Lang, 1993]), de préférences (CSP flou [Fargier *et al.*, 1993]), de priorités (CSP possibiliste [Schiex, 1992]), de violations (Max-CSP [Larrosa *et al.*, 1999]), de pénalités ou coûts (CSP pondéré [Larrosa et Schiex, 2004]), etc. Il ne s'agit plus cette fois-ci de problèmes de satisfaction mais plus précisément de problèmes d'optimisation avec un objectif à satisfaire de façon optimale. Le cadre CSP et toutes ses extensions peuvent être généralisés et englobés dans deux cadres généraux : le cadre SCSP (Semiring-based CSP) et VCSP (Valued CSP). Nous présentons brièvement

les cadres SCSP et VCSP, pour ensuite aborder en détail une instanciation du cadre VCSP qui nous a intéressé tout particulièrement dans le cadre de cette thèse : le cadre WCSP (ou CSP pondéré).

2.1 Présentation des cadres généraux SCSP et VCSP

Les cadres généraux SCSP et VCSP ont été proposés pour englober le cadre CSP et ses différentes extensions. Basés sur des propriétés générales permettant d'exprimer ces différents cadres, SCSP et VCSP proposent des algorithmes suffisamment *génériques* pour être exploitables efficacement dans ces différents formalismes, tout en assurant de la flexibilité pour les problèmes qui nécessitent plus de *spécificité*. Ces deux cadres généraux sont basés sur des structures mathématiques qui permettent de représenter ces différents paradigmes. Ces structures mathématiques sont constituées d'un ensemble de valeurs pouvant représenter des probabilités, des préférences, des priorités ou des coûts en fonction du cadre considéré. Dans le cadre SCSP, ces valeurs vont correspondre à des niveaux de satisfaction, tandis que dans le cadre VCSP elles correspondent à des niveaux d'insatisfaction (ou pénalités dans le sens de la logique des pénalités [Pinkas, 1991, De Saint-Cyr *et al.*, 1994]). Ensuite, des opérations caractérisées par certaines propriétés ou axiomes permettent de combiner ces valeurs entre les différentes contraintes d'un réseau dans le cadre considéré. Naturellement, les objectifs sont différents : pour le cadre SCSP, l'objectif est de maximiser la satisfaction de l'ensemble des contraintes tandis que pour le cadre VCSP l'objectif est de minimiser l'insatisfaction. Pour des raisons de précision et en amont de ce qui sera présenté dans la suite de ce manuscrit, nous considérons ici que les valeurs contenues dans la structure mathématique sont associées aux tuples des contraintes et représentent donc pour chaque tuple une probabilité, ou un coût, etc.

SCSP ou CSP basé sur un *semi-anneau* (*semiring*) [Bistarelli *et al.*, 1995, Bistarelli *et al.*, 1996]

Ce cadre permet la généralisation du cadre CSP par l'utilisation de la structure *c-semi-anneau* (*c-semiring*). Il s'agit de la structure mathématique appelée *semi-anneau* à laquelle ont été ajoutées des propriétés supplémentaires sur les opérations.

Définition 32 (c-semi-anneau) *Un semi-anneau $\mathcal{A}$ est un quintuplet $(E, \oplus, \otimes, 0, 1)$ tel que :*

- *E est un ensemble tel que $0 \in E$, $1 \in E$*
- *$\oplus$ est appelée opération additive*
  - *elle est interne à E ($a, b \in E \Rightarrow a \oplus b \in E$)*
  - *elle est commutative ($\forall a, b \in E, a \oplus b = b \oplus a$)*
  - *elle est associative ($\forall a, b, c \in E, a \oplus (b \oplus c) = (a \oplus b) \oplus c$)*
  - *son élément neutre est 0 ($\forall a \in E, a \oplus 0 = a$)*
  - *son élément absorbant est 1 ($\forall a \in E, a \oplus 1 = 1$)*
- *$\otimes$ est appelée opération multiplicative*
  - *elle est interne à E ($a, b \in E \Rightarrow a \otimes b \in E$)*
  - *elle est associative ($\forall a, b, c \in E, a \otimes (b \otimes c) = (a \otimes b) \otimes c$)*
  - *son élément neutre est 1 ($\forall a \in E, a \otimes 1 = a$)*
  - *son élément absorbant est 0 ($\forall a \in E, a \otimes 0 = 0$)*
  - *elle est distributive par rapport à $\oplus$ ($\forall a, b, c \in E, a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$)*

Un c-semi-anneau est un semi-anneau tel que 1) $\oplus$ est idempotente ($a \in E \Rightarrow a \oplus a = a$), 2) $\otimes$ est commutative

Définition 33 (SCSP) *Un Semi-anneau CSP (SCSP) est un triplet $P = (\mathcal{X}, \mathcal{C}, \mathcal{A})$, où $\mathcal{X}$ représente l'ensemble des variables, $\mathcal{C}$ l'ensemble des contraintes et $\mathcal{A}$ la structure c-semi-anneau associée. Chaque*

contrainte dans $\mathcal{C}$ est définie comme une fonction qui associe à chaque élément du produit cartésien des domaines des variables de sa portée un élément appartenant à $\mathcal{A}$.

Au delà d'unifier dans un même cadre les différents formalismes CSP au travers de la spécialisation de cette structure générale *c-semi-anneau*, le cadre général SCSP généralise également le concept de cohérence locale (plus précisément la k-cohérence introduite dans la section 1.2 du chapitre 1) dont l'intérêt n'est plus à prouver dans le cadre CSP. Ici aussi, la cohérence a pour intérêt de simplifier un réseau en un réseau équivalent en filtrant des valeurs incohérentes et de potentiellement découvrir si le réseau est insatisfaisable. Cette extension au contexte d'optimisation de la k-cohérence [Bistarelli *et al.*, 1996] est fondée notamment sur des propriétés d'équivalence et d'indépendance d'ordre qui requièrent une spécificité au niveau des opérations du *c-semi-anneau* : l'opération multiplicative doit être idempotente. Si l'idempotence est vérifiée pour le cadre considéré, le concept de cohérence locale (et les propriétés générales qui sont liées) peut être exploité directement avec succès et dans une complexité semblable (à un facteur polynomial près) à la k-cohérence dans le cadre CSP. Sinon, aucune garantie n'est donnée sur l'établissement de la cohérence (algorithme pouvant ne pas se terminer ou réseau obtenu non équivalent). À noter que la condition de terminaison est nécessaire également pour l'utilisation de la k-cohérence, cependant elle est toujours vérifiée car elle est implicite de par la structure *c-semi-anneau* : les opérations doivent être internes et c'est le cas dans cette structure.

Considérons par exemple le cadre CSP : il s'agit d'un SCSP dont le *c-semi-anneau* est défini par $(\{0,1\}, \vee, \wedge, 0, 1)$, c'est à dire avec $E = \{0,1\}$. Dans un CSP, les tuples dans les contraintes sont soit autorisés (valeur 1 dans le *c-semi-anneau*), soit interdits (valeur 0 dans le *c-semi-anneau*). L'opération multiplicative de cet anneau, considérée comme la manière de combiner et joindre les valeurs associées aux tuples des différentes contraintes, correspond pour le cadre CSP à l'opérateur *et logique* $\wedge$ (en considérant que la valeur 0 corresponde à $faux$ et que la valeur 1 corresponde à $vrai$). La commutativité (2) de cet opérateur est vérifiée et permet d'assurer que la considération dans un ordre quelconque des différentes contraintes ne donnera pas un résultat différent. L'opération additive de cet anneau, considérée comme la manière de déterminer pour une contrainte la meilleure valeur parmi celles associées aux tuples de cette contrainte, correspond pour le cadre CSP à l'opérateur *ou logique* $\vee$. L'idempotence de cet opérateur (1) est vérifiée ($0 \vee 0 = 0$ et $1 \vee 1 = 1$ avec $0, 1 \in E$) et permet d'établir un ordre partiel sur $E = \{0,1\}$ défini par $0 \leq_E 1$ car $0 \vee 1 = 1 \vee 0 = 1$: la valeur 1 est dite meilleure que 0, dans le sens où entre 0 et 1 cet opérateur choisit 1, et représente dans ce cas précis l'élément absorbant, c'est à dire $\forall a \in E, a \leq_E 1$. À noter que les opérations $\oplus$ et $\otimes$ sont monotones par rapport à l'ordre partiel $\leq_E$ induit par l'idempotence de $\oplus$: $\forall * \in \{\oplus, \otimes\}, \forall a, b, c \in E, (a \leq_E b) \Rightarrow (a * c) \leq_E (b * c)$. Le concept généralisé de cohérence proposé par le cadre SCSP est donc applicable au cadre CSP. En effet, cela est vérifié par le fait que $\wedge$ est idempotente et que les propriétés d'équivalence et d'indépendance d'ordre [Bistarelli *et al.*, 1996] sont donc vérifiées.

Selon le même principe, nous définissons le WCSP comme un SCSP dont le *c-semi-anneau* est défini par $(\mathbb{N} \cup \{+\infty\}, min, +, +\infty, 0)$. Directement, on peut s'apercevoir que le concept généralisé de cohérence proposé par le cadre SCSP n'est pas applicable dans l'état au cadre WCSP : en effet, l'opérateur $+$ n'est pas idempotent et donc les propriétés d'équivalence et d'indépendance d'ordre ne sont pas satisfaites. Au delà de la cohérence, d'autres propriétés sont proposées par le cadre général et peuvent être exploitées par tout cadre défini par un *c-semi-anneau* : c'est le cas notamment de la propriété α-cohérence locale et globale [Bistarelli *et al.*, 1996] qui est très utile dans les méthodes de recherche arborescente par séparation et évaluation très connues dans le contexte des problèmes d'optimisation et que nous aborderons dans la suite de ce manuscrit.

VCSP ou CSP valué [Schiex *et al.*, 1995]

Contrairement au cadre SCSP, le cadre VCSP permet la généralisation des différents formalismes présentés précédemment par l'utilisation d'une structure mathématique appelée *monoïde*. Plus précisément, il s'agit d'un *monoïde* positif totalement ordonné considéré aussi comme une *structure de valuation*, constituée d'un ensemble totalement ordonné de valeurs avec une loi de composition interne sur cet ensemble.

Définition 34 (Structure de valuation) *Une structure de valuation $\mathcal{V}$ est un triplet $(E, \oplus, \succ)$ tel que :*
- *E est un ensemble d'éléments dont l'élément minimal est noté $\bot$ et l'élément maximal est noté $\top$*
- *E est totalement ordonné par $\succ$*
- *$\oplus$ est une opération binaire définie sur E interne, commutative, associative et monotone*
- *L'élément neutre de $\oplus$ est $\bot$: $\forall a \in E, a \oplus \bot = a$*
- *L'élément absorbant de $\oplus$ est $\top$: $\forall a \in E, a \oplus \top = \top$*

Définition 35 (VCSP) *Un CSP valué (VCSP) est un triplet $P = (\mathcal{X}, \mathcal{C}, \mathcal{V})$, où $\mathcal{X}$ représente l'ensemble des variables, $\mathcal{C}$ l'ensemble des contraintes et $\mathcal{V}$ la structure de valuation associée. Chaque contrainte dans $\mathcal{C}$ est définie comme une fonction qui associe à chaque élément du produit cartésien des domaines des variables de sa portée un élément (appelé aussi valuation) appartenant à $\mathcal{V}$. La valuation $\mathcal{V}_P(I)$ d'une instanciation I dans P est définie par $\mathcal{V}_P(I) = \bigoplus_{c \in \mathcal{C}} c(I)$.*

Comme dans le cadre SCSP, une version de cohérence étendue au contexte d'optimisation a été proposée dans le cadre VCSP (plus précisément l'arc-cohérence introduite dans la section 1.2 du chapitre 1). Le même constat peut être fait que pour le cadre SCSP : cette extension de l'arc-cohérence [Schiex *et al.*, 1995] peut être exploitée en temps polynomial pour les cadres où l'opérateur de la structure de valuation est idempotent (c'est le cas par exemple pour les cadres CSP classique et CSP possibiliste). Dans le cas où l'opérateur est strictement monotone ($\forall a, b, c \in E, (a \succ c), (b \neq \top) \Rightarrow (a \oplus b) \succ (c \oplus b)$), c'est à dire notamment pour les cadres Max-CSP et WCSP, établir et vérifier l'arc-cohérence représente un problème `NP-Complet`.

Considérons par exemple le cadre CSP : il s'agit d'un VCSP dont la structure de valuation est définie par $(\{t, f\}, \wedge, \succ)$ avec $\succ$ défini par $t \prec f$. La valeur t correspond à $\bot$ et la valeur f correspond à $\top$. Les valuations sont combinées via l'opérateur $\wedge$. Bien évidemment l'arc-cohérence est applicable en temps polynomial dans ce cas précis : on vérifie facilement que l'opérateur $\wedge$ est idempotent et strictement monotone (c'est le seul cas de figure où ces deux propriétés sont vérifiées simultanément pour une structure de valuation).

Selon le même principe, nous définissons le WCSP comme un VCSP dont la structure de valuation est définie par $(\mathbb{N} \cup \{+\infty\}, +, <)$. Directement, on peut s'apercevoir que le concept d'arc-cohérence proposé par le cadre VCSP n'est pas applicable en temps polynomial dans l'état au cadre WCSP : l'opérateur $+$ n'est pas idempotent mais strictement monotone. Le cadre VCSP propose alors un algorithme général pour établir l'arc-cohérence en temps polynomial pour les cadres dont l'opérateur est strictement monotone [Schiex, 2000]. Cet algorithme s'appuie aussi sur la notion d'équivalence dans le cadre VCSP entre un réseau et le réseau simplifié obtenu à partir de ce réseau après avoir établi l'arc-cohérence.

Définition 36 (VCSP équivalents [Cooper et Schiex, 2004]) *Soient deux VCSP $P = (\mathcal{X}, \mathcal{C}, \mathcal{V})$ et $P' = (\mathcal{X}, \mathcal{C}', \mathcal{V})$. P et P' sont équivalents si et seulement si $\mathcal{V}_P(I) = \mathcal{V}_{P'}(I)$ pour toute instanciation complète I sur $\mathcal{X}$.*

L'arc-cohérence est établie sur un réseau en filtrant et en supprimant les valeurs des domaines qui ne sont pas arc-cohérentes. Dans le cas de la version étendue de l'arc-cohérence [Schiex *et al.*, 1995], une

valeur est arc-cohérente si elle représente une valuation différente de $\top$ et s'il existe au moins un tuple support de valuation $\bot$ contenant cette valeur dans chaque contrainte qui la couvre. L'algorithme proposé détecte efficacement les valeurs incohérentes en vérifiant ces deux conditions grâce à des opérations de *Projection* et d'*Extension*. Le principe est de concentrer les valuations au niveau des variables, plus précisément au niveau des valeurs dans les domaines, pour détecter d'une part les valeurs incohérentes à supprimer, et d'autre part pour détecter les tuples qui contiennent des valeurs supprimées et qui ne sont donc pas des supports. L'opération de *projection* transfère les valuations des contraintes vers les valeurs (via l'ajout ou la mise à jour de contraintes unaires pour les variables contenant ces valeurs) et l'opération d'*extension* transfère les valuations des valeurs vers les contraintes. Comme nous sommes dans le cas d'un opérateur non idempotent et que la préservation de l'équivalence est primordiale, les opérations de *projection* et d'*extension* doivent être accompagnées d'une opération de compensation afin d'éviter de comptabiliser plusieurs fois les mêmes valuations. En ce sens, elles sont appelées transformations préservant l'équivalence d'un VCSP.

Définition 37 (Transformation préservant l'équivalence d'un VCSP [Cooper et Schiex, 2004]) *Une transformation préservant l'équivalence d'un VCSP P est une opération qui transforme P en un VCSP équivalent.*

Le cadre VCSP propose alors une deuxième propriété qui permet d'effectuer ces transformations préservant l'équivalence : la structure de valuation juste.

Définition 38 (Structure de valuation juste [Cooper et Schiex, 2004]) *Dans une structure de valuation $(E, \oplus, \succ)$, s'il existe $a, b \in E$ tels que $a \prec b$ et qu'il existe un élément $c \in E$ tel que $a \oplus c = b$, alors c représente la différence entre b et a. La structure de valuation est juste si pour n'importe quelle paire $a, b \in E$ avec $a \prec b$, il existe une différence maximale entre b et a. Cette différence maximale entre b et a est notée $b \ominus a$.*

Établir l'arc-cohérence générale ne peut se faire que sur des structures de valuation justes permettant la préservation de l'équivalence des réseaux. Les formalismes les plus connus englobés dans le cadre VCSP, que l'opérateur soit idempotent ou strictement monotone, vérifient la propriété de structure de valuation juste. De plus, l'arc-cohérence établie propose toutes les propriétés connues de l'arc-cohérence (stabilité par union, etc.) présentées dans le cadre CSP sauf l'unicité du réseau arc-cohérent obtenu (fermeture par arc-cohérence) : en effet, la séquence d'opérations de projections ou extensions (ou transferts) de valuations n'est pas unique et le réseau final dépend directement de l'ordre suivi. Cette unicité est rétablie dans le cadre des opérateurs idempotents.

De plus, dans le contexte d'optimisation que peut naturellement représenter le cadre VCSP, il s'avère qu'une borne inférieure de très bonne qualité (primordiale pour une recherche par séparation et évaluation efficace) peut être obtenue à partir du réseau arc-cohérent, c'est à dire meilleure que les bornes obtenues par les techniques d'estimation proposées (notamment dans [Larrosa *et al.*, 1999]). Cette borne inférieure correspond à la valuation minimale de toute instanciation complète dans le réseau arc-cohérent et elle est définie par $lb = \bigoplus_{x \in \mathcal{X}} [min_{a \in dom(x)} \mathcal{V}(a)]$ où l'on considère pour chaque variable la valeur de valuation minimale, $\mathcal{V}(a)$ représentant la valuation de la valeur a. Elle correspond en fait à la somme des valuations minimales que représente l'instanciation de chacune des variables. Indirectement, comme les valuations sont transférées et concentrées au niveau des variables, les informations de valuation du réseau deviennent plus directes et accessibles et plus la borne est de qualité : c'est à dire qu'elle permet une très bonne estimation car plus proche des valuations en provenance de la totalité du réseau. Elle représente au plus près le réseau en terme de valuation. Nous verrons plus tard dans le manuscrit que la qualité de cette borne est primordiale pour résoudre efficacement certains VCSP et que sa qualité dépend

de l'ordre des séquences d'opérations pour établir l'arc-cohérence. Dans cette optique, nous verrons que différents algorithmes visant à obtenir une borne optimale ont été proposés.

Au delà du cadre CSP et de son extension WCSP abordés précédemment, d'autres extensions du cadre CSP peuvent être exprimées par une structure de valuation. On spécifie alors pour chacune de ces classes VCSP l'ensemble E des éléments de valuation, l'ordre établi sur E ainsi que l'opérateur $\oplus$ dont les caractéristiques (notamment la monotonie et l'idempotence) déterminent les propriétés du cadre VCSP pouvant être exploitées. Les définitions de certains paradigmes (CSP possibilistes, CSP probabilistes, CSP lexicographiques, etc.) depuis le cadre VCSP, ainsi que des informations pointues et des comparaisons sur les cadres généraux SCSP et VCSP, sont accessibles dans [Bistarelli *et al.*, 1996]. Nous reviendrons également plus en détail sur les concepts d'arc-cohérence établies via des opérations préservant l'équivalence et sur le calcul de borne dans un réseau arc-cohérent proposés dans le cadre VCSP à travers l'étude en détail du cadre WCSP, à savoir une classe de VCSP où l'opérateur est strictement monotone.

2.2 Réseaux de contraintes pondérées

Après avoir défini les réseaux de contraintes (CN), nous allons présenter à présent les réseaux de contraintes pondérées.

2.2.1 Définitions et notations

Définition 39 (Réseau de contraintes pondérées) *Un réseau de contraintes pondérées se compose d'un ensemble fini de variables, d'un ensemble fini de contraintes souples (pondérées) portant sur ces variables et d'un entier naturel k strictement positif ou égal à $+\infty$ représentant le coût interdit pour ce réseau.*

Dans un réseau de contraintes pondérées, une contrainte souple porte sur un ensemble de variables (sa portée) et peut être définie soit en intention, soit en extension, comme une contrainte dure. Cependant, à la différence d'une contrainte dure, elle est définie comme une fonction de coût qui associe un coût entier de $[0, \ldots, k]$ à chaque élément du produit cartésien des domaines des variables appartenant à sa portée. Dans un réseau de contraintes pondérées, pour obtenir le coût global d'une instanciation, les coûts sont combinés grâce à l'opérateur $\oplus$ défini par :

$$\forall \alpha, \beta \in [0, \ldots, k], \alpha \oplus \beta = min(k, \alpha + \beta) \tag{2.1}$$

L'opérateur $\ominus$, inverse partiel de l'opérateur $\oplus$ dans le réseau, est défini par :

$$\forall \alpha, \beta \in [0, \ldots, k[, 0 \leq \beta \leq \alpha < k \Rightarrow \alpha \ominus \beta = \alpha - \beta \tag{2.2}$$

$$\forall \alpha \in [0, \ldots, k[, k \ominus \alpha = k \tag{2.3}$$

Nous distinguons clairement dans un réseau de contraintes pondérées les éléments de la structure de valuation caractérisant les VCSP : l'ensemble totalement ordonné des valuations correspond à $[0, \ldots, k]$, les éléments neutres et absorbants sont respectivement $\bot = 0$ et $\top = k$, et $\oplus$ est l'opérateur strictement monotone pour combiner les valuations.

Nous utiliserons la notation $P = (\mathcal{X}, \mathcal{C}, k)$ pour définir le réseau de contraintes pondérées (WCN pour Weighted Constraint Network) P composé de l'ensemble des variables $\mathcal{X}$, de l'ensemble des contraintes souples $\mathcal{C}$, et caractérisé par le coût interdit k. Le coût associé à un tuple τ (resp. valeur a) par une

contrainte souple binaire ou n-aire (resp. contrainte souple unaire) c est noté $c(\tau)$ (resp. $c(a)$). Pour une contrainte c_S, nous notons $l(S)$ l'ensemble des tuples possibles sur les domaines des variables présentes dans la portée S. Il est admis l'existence dans le réseau d'une contrainte 0-aire notée $c_\emptyset$ (son arité est nulle et elle correspond à une valeur numérique) qui représente un coût constant dans le réseau. Plus précisément, la valeur contenue dans $c_\emptyset$ représente ce que coûtera au minimum n'importe quelle solution pouvant être obtenue depuis ce réseau : on l'utilise souvent comme une borne inférieure. De plus, l'existence d'une contrainte unaire portant sur chaque variable du réseau est également admise. Si ces contraintes ne sont pas définies pour un réseau, alors nous pouvons les ajouter en fixant $c_\emptyset = 0$, puis $c_x(a) = 0$ pour chaque variable x appartenant au réseau et pour chaque valeur a appartenant à $dom^{init}(x)$.

Dans le cadre de cette thèse, nous nous sommes plus particulièrement intéressés aux contraintes souples définies en extension, appelées aussi contraintes tables souples.

Définition 40 (Contrainte table souple) *Une contrainte table souple est une contrainte définie en extension par une liste de tuples, une liste d'entiers correspondant aux coûts de ces tuples et un coût par défaut. Les tuples présents dans la table sont appelés tuples explicites tandis que les tuples non présents sont appelés tuples implicites et ont un coût égal au coût par défaut défini pour la contrainte table souple.*

D'un point de vue modélisation, une contrainte table souple correspond donc à un ensemble de paires (τ, c) qui associent à chaque tuple (explicite) τ son coût c dans la contrainte. Nous faisons l'hypothèse que le coût associé à un tuple explicite est différent du coût par défaut défini pour la contrainte. Dans le cadre de cette thèse, nous avons fait le choix (guidé par l'implémentation) de représenter une contrainte table souple c_S par un tableau noté *table*[c_S] (ou simplement *table* quand il n'y a pas d'ambiguïté possible) et par un tableau noté *couts*[c_S] (ou *couts*). Le coût associé au tuple à la position i dans *table* ($table[i]$) correspond au coût à la position i dans *couts* ($couts[i]$). Nous illustrons plus précisément ces contraintes tables souples dans l'exemple concret présenté à la section 2.2.2.

Définition 41 (Tuple autorisé (resp. interdit)) *Un tuple τ est autorisé (resp. interdit) dans une contrainte souple c si et seulement si $c(\tau) \neq k$ (resp. $c(\tau) = k$). De plus, τ est dit totalement autorisé si $c(\tau) = 0$.*

La même définition s'applique dans le cadre d'une contrainte unaire où une valeur est un 1-tuple.

Exemple 10 *Soient c_x, c_y et c_z trois contraintes souple unaires, c_{xz} et c_{yz} deux contraintes souples binaires, avec $dom^{init}(x) = dom^{init}(y) = dom^{init}(z) = \{a, b\}$ dans un réseau avec un coût interdit $k = 5$. Une première représentation de ce réseau est proposée à la figure 2.1. La contrainte $c_\emptyset$ est égale à 1 (valeur calculée par exemple suite à un pré-traitement), c'est à dire qu'une solution dans ce réseau coûte au moins 1. La contrainte c_z autorise la valeur (z, b) avec un coût de 2 ($c_z(b) = 2$) tandis qu'elle autorise totalement la valeur (z, a) avec un coût de 0 ($c_z(a) = 0$). La contrainte c_{yz} interdit le tuple (a, a) ($c_{yz}((a, a)) = 5 = k$), autorise le tuple (a, b) avec un coût de 1 ($c_{yz}((a, b)) = 1$) et autorise totalement le tuple (b, a) avec un coût de 0 ($c_{yz}((b, a)) = 0$). Une deuxième représentation de ce réseau, cette fois-ci sous la forme d'un graphe, est proposée à la figure 2.2. Les coûts unaires sont indiqués dans les cercles des valeurs des domaines. Le coût d'un tuple dans une contrainte binaire est représenté par un segment reliant les valeurs de ce tuple. Le coût correspond au libellé porté par ce segment, avec un coût de 1 si aucun libellé précisé. Un tuple constitué de valeurs non reliées par un segment a un coût de 0.*

Définition 42 (Coût étendu d'un tuple) *Le coût étendu d'un tuple τ dans une contrainte c_S englobe le coût $c_S(\tau)$ du tuple τ dans la contrainte c_S, les coûts des contraintes unaires portant sur les valeurs*

k

5

$c_\emptyset$

1

c_x

x	cout
a	1
b	0

c_{xz}

x	z	cout
a	a	1
a	b	0
b	a	1
b	b	0

c_{yz}

y	z	cout
a	a	5
a	b	1
b	a	0
b	b	1

c_y

y	cout
a	1
b	0

c_z

z	cout
a	0
b	2

FIG. 2.1 – Un réseau de contraintes pondérées constitué d'une contrainte 0-aire $c_\emptyset$, de trois contraintes unaires c_x, c_y et c_z et de deux contraintes binaires c_{xz} et c_{yz}.

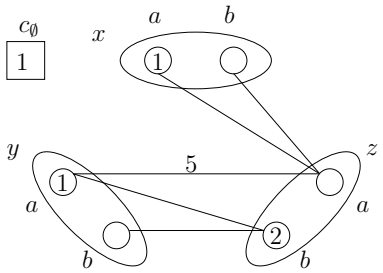


FIG. 2.2 – Représentation sous la forme d'un graphe du réseau illustré dans la figure 2.1.

appartenant à τ, ainsi que $c_\emptyset$. Ce coût est donc défini par :

$$\mathcal{V}(\tau) = c_\emptyset \oplus \bigoplus_{x \in S} c_x(\tau[x]) \oplus c_S(\tau) \tag{2.4}$$

Dans l'exemple 10, le coût étendu $\mathcal{V}((a,b))$ du tuple (a,b) dans la contrainte c_{yz} est égal à $1 \oplus 1 \oplus 2 \oplus 1 = 5$: la contrainte 0-aire $c_\emptyset$ vaut 1, les contraintes unaires c_y et c_z autorisent respectivement les valeurs a et b avec un coût de 1 et 2, puis la contrainte binaire c_{yz} autorise le tuple (a,b) avec un coût de 1. Le coût interdit k étant égal à 5, le tuple (a,b) est donc interdit car $\mathcal{V}((a,b)) = 5$, par contre le tuple (b,b) est autorisé avec $\mathcal{V}((b,b)) = 1 \oplus 0 \oplus 2 \oplus 1 = 4$.

Définition 43 (Coût d'une instanciation) *Si c_S est une contrainte souple et que I représente l'instanciation d'un ensemble de variables $X \supseteq S$, alors $c_S(I)$ est considéré comme égal à $c_S(I_{\downarrow S})$ (en d'autres termes, les projections sont implicites). Pour un réseau de contraintes pondérées et une instanciation complète sur ce réseau, le coût de I correspond à :*

$$\mathcal{V}(I) = c_\emptyset \oplus \sum_{c_S \in \mathcal{C}} c_S(I) \tag{2.5}$$

Définition 44 (Solution / solution optimale) *Une solution pour un réseau de contraintes pondérées P est une instanciation complète de coût strictement inférieur à k. Il s'agit d'une solution optimale dans P si et seulement si son coût est inférieur ou égal au coût de toute autre solution du réseau P.*

Notons qu'une solution dont le coût est égal à 0 satisfait totalement le réseau associé.

Définition 45 (Réseaux de contraintes pondérées équivalents) *Deux réseaux de contraintes pondérées $P : (\mathcal{X}, \mathcal{C}, k)$ et $P' : (\mathcal{X}, \mathcal{C}', k)$ sont équivalents si et seulement si $\mathcal{V}_P(I) = \mathcal{V}_{P'}(I)$ pour toute instanciation complète I sur $\mathcal{X}$.*

Définition 46 (Réseau de contraintes pondérées satisfaisable) *Un réseau de contraintes pondérées est satisfaisable si et seulement si il admet au moins une solution (de coût strictement inférieur à k).*

2.2.2 Un exemple concret : le problème des mots croisés pondérés

Nous allons considérer un exemple concret de réseau de contraintes pondérées pour illustrer les différentes notions présentées dans la section précédente. Il est inspiré du problème des mots croisés pondérés, appelé aussi *crossoft* [Lecoutre *et al.*, 2012], issu du problème de mots croisés classique introduit dans le cadre CSP [Ginsberg *et al.*, 1990, Beacham *et al.*, 2001].

Tout le monde connaît le jeu des mots croisés consistant à remplir une grille vierge (contenant éventuellement des cases noires) avec des mots du dictionnaire, à partir de définitions dont les indices présents dans la grille indiquent les suites de cases vides adjacentes allouées à la réponse (partie gauche de la figure 2.3). Nous nous intéressons ici au problème inverse : concevoir cette grille de mots croisés, c'est à dire déterminer les mots que les joueurs auront à trouver. En d'autres termes, à partir d'une liste de mots proposés dans un dictionnaire, l'objectif est de remplir dans une grille vierge donnée les cases vides (c'est à dire les cases qui ne sont pas noires) par des mots. Bien évidemment, deux mots qui se croisent doivent avoir au moins une lettre en commun, en l'occurrence celle située à la position de leur croisement dans la grille. De plus, nous apportons une nuance à la conception de cette grille : les mots utilisés seront choisis dans un dictionnaire proposant des noms propres et des noms communs. L'utilisation d'un nom commun n'entraînera aucune pénalité, tandis que l'utilisation d'un nom propre entraînera un coût correspondant à la taille de ce mot : autrement dit, un nom commun est préféré à un nom propre. L'objectif est simple : remplir la grille de sorte que le coût total soit minimal.

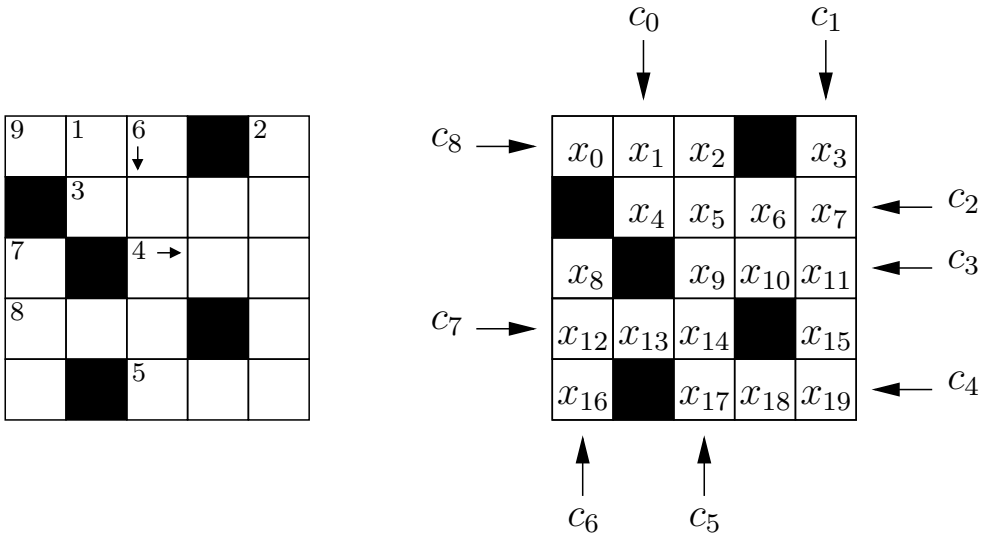


FIG. 2.3 – Une grille vierge et une modélisation possible en réseau de contraintes.

Ce problème se modélise tout naturellement en réseau de contraintes pondérées. Une représentation possible est de considérer chaque case vide à remplir comme une variable du réseau, soit un total de 20 variables nommées x_0, x_1, ..., x_{19}. Naturellement, les domaines de ces variables sont constitués des 26 lettres de l'alphabet et pour chacune de ces variables on a donc $dom(x_0) = \{a, b, \ldots, z\}$, ...,

$dom(x_{19}) = \{a, b, \ldots, z\}$. Ensuite, nous considérons une contrainte par suite de cases adjacentes vides représentant un mot à trouver. Dans la figure 2.3, il y a 9 mots à trouver, il y a donc 9 contraintes nommées $c_0, c_1, \ldots, c_8$. Ces contraintes portent sur les variables correspondant aux différentes cases allouées pour le mot à trouver et ont des arités correspondant à la taille de ce mot. Nous avons donc par exemple $scp(c_0) = \{x_1, x_4\}$, $scp(c_1) = \{x_3, x_7, x_{11}, x_{15}, x_{19}\}$, $\ldots$, $scp(c_8) = \{x_0, x_1, x_2\}$. Dans chacune de ces contraintes, les tuples représentent des mots du dictionnaire et ont une taille égale à l'arité de la contrainte. Nous allons exprimer la préférence des noms communs par rapport aux noms propres par une stratification des tuples correspondants en deux niveaux de coûts. Nous choisissons naturellement d'attribuer un coût de 0 (aucune pénalité) aux tuples représentant les noms communs et un coût égal à l'arité de la contrainte (autrement dit la taille du mot) pour les tuples représentant les noms propres. Nous représentons donc ces contraintes par des contraintes tables souples dont les tuples explicites contenus dans la table correspondent aux différents noms disponibles dans le dictionnaire, communs ou propres, avec leurs coûts respectifs. Notons que dans cet exemple nous n'avons pas défini de contrainte $c_\emptyset$ ($c_\emptyset = 0$). De même, nous n'avons pas défini de contraintes unaires portant sur les lettres de l'alphabet ($\forall 0 \leq i \leq 19, c_{x_i}(a) = c_{x_i}(b) = \cdots = c_{x_i}(z) = 0$). Une illustration partielle des contraintes du réseau est proposée à la figure 2.4. Par exemple, dans la contrainte c_1, le tuple (L,A,U,R,E), qui correspond à un nom propre du dictionnaire, est associé à un coût de 5 (taille du mot et arité de la contrainte), tandis que le tuple (V,O,I,L,E) est associé à un coût de 0 car c'est un nom commun. Les mots n'apparaissant pas dans le dictionnaire sont strictement interdits et sont représentés implicitement par le coût par défaut égal à $+\infty$ pour toutes les contraintes de ce réseau, qui correspond également au coût interdit de ce réseau de contraintes pondérées.

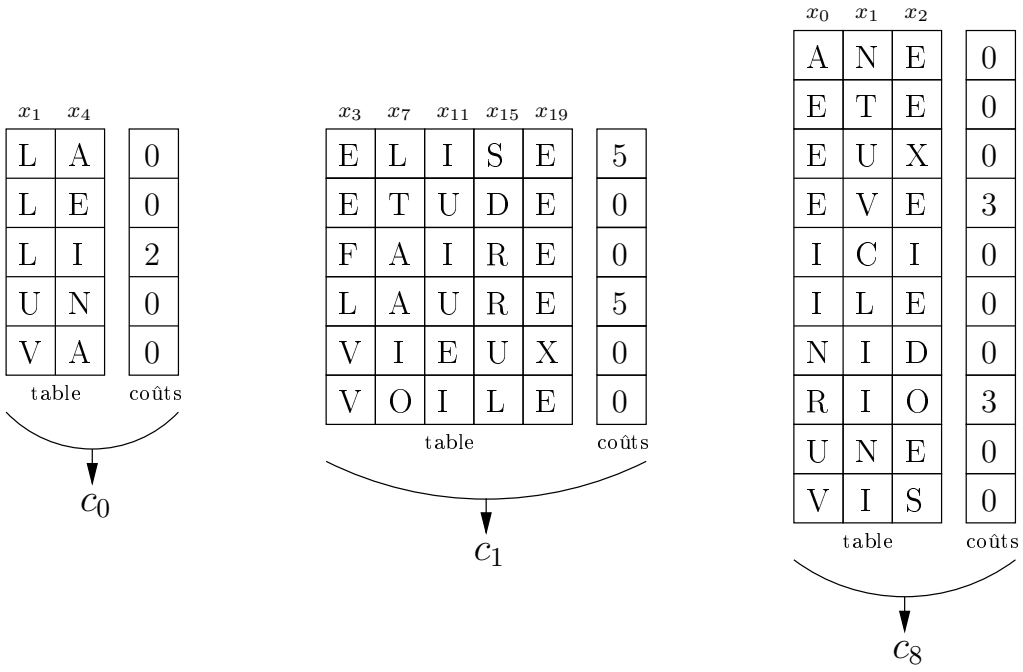


FIG. 2.4 – Les représentations des contraintes c_0, c_1 et c_8 en contraintes tables souples.

L'instanciation complète illustrée dans la figure 2.5 représente un coût égal à 12. En effet, les tuples sélectionnés pour les contraintes c_2 (E,L,S,A), c_5 (E,L,I,S,E) et c_6 (E,V,E) sont associés respectivement à des coûts 4, 5 et 3. Pour les autres contraintes du réseau, les tuples choisis ont tous un coût égal à 0 : par exemple le tuple (E,T,E) pour la contrainte c_4, le tuple (V,I,S) pour c_7, etc. Considérant toutes les contraintes, le coût total de cette solution est donc égal à $0 \oplus 0 \oplus 4 \oplus 0 \oplus 0 \oplus 5 \oplus 3 \oplus 0 \oplus 0 =$

12. Le coût de cette instanciation est donc différent du coût interdit $+\infty$: il s'agit donc d'une solution. Cependant, il ne s'agit pas d'une solution de coût optimal. Par contre, l'instanciation illustrée à la figure 2.6 est une solution optimale de coût 0. Pour terminer, l'instanciation complète illustrée à la figure 2.7 n'est pas une solution acceptable pour ce réseau de contraintes. En effet, les contraintes c_1 et c_8 sont violées : les tuples (V,I,E,U,E) et (A,L,E) ne sont pas dans les tables de c_1 et c_8, il s'agit donc de tuples implicites et leurs coûts correspondent au coût par défaut, c'est à dire $+\infty$. Ces tuples sont donc interdits et le coût total de cette instanciation est donc $+\infty$.

I	L	E	■	F
■	E	L	S	A
E	■	I	C	I
V	I	S	■	R
E	■	E	T	E

FIG. 2.5 – Une solution de coût 12.

I	L	E	■	V
■	E	T	U	I
A	■	U	N	E
N	I	D	■	U
E	■	E	U	X

FIG. 2.6 – Une solution optimale de coût 0.

A	L	E	■	V
■	E	T	U	I
A	■	U	N	E
N	I	D	■	U
E	■	E	T	E

FIG. 2.7 – Une instanciation interdite (coût $+\infty$) : les contraintes c_1 et c_8 sont violées.

2.2.3 Résolution d'un réseau de contraintes pondérées

L'objectif dans la résolution d'un réseau de contraintes pondérées ou problème de satisfaction de contraintes pondéré (appelé aussi WCSP pour Weighted Constraint Satisfaction Problem) est de trouver une solution optimale pour ce réseau. Tout comme dans le cadre des problèmes de satisfaction de contraintes, une fois un problème modélisé sous la forme d'un réseau de contraintes pondérées (abordé dans la section précédente), la démarche consiste à le faire résoudre par un solveur. Nous ne reviendrons pas plus en détail ici sur le solveur $AbsCon$ utilisé pour résoudre les WCSP dans le cadre de la thèse et déjà présenté dans la section 1.1.3. Cependant, précisons que nous avons choisi le solveur $ToulBar2$[8] (version 0.9) à titre de comparaison pour les différentes contributions que nous avons proposées dans le cadre WCSP. $ToulBar2$ est actuellement le solveur le plus efficace pour la résolution des WCSP. Il s'agit d'un solveur libre développé en C++ tenant son nom de l'effort commun de développement par des équipes de *Tou*louse et de *Bar*celone. Il propose des techniques de stratégies de recherche complète mais également des stratégies de recherche incomplète à travers la bibliothèque incorporée appelée IN-

[8]ToulBar2 weighted constraint satisfaction solver website : https ://mulcyber.toulouse.inra.fr/projects/toulbar2/

COP [Neveu et Trombettoni, 2003]. Nous allons aborder dans cette partie la complexité représentant la résolution d'un WCSP et les différentes techniques qui existent pour le résoudre.

Problème d'optimisation NP-Difficile

En passant du cadre des CSP au cadre des WCSP, nous avons changé de catégorie de problème. Il ne s'agit plus ici d'un problème de décision comme c'est le cas pour les CSP, mais plutôt d'un problème d'optimisation. Les WCSP font partie de la classe des problèmes NP-Difficiles. Comme expliqué dans la section 1.1.3, un problème NP-Difficile n'est pas obligatoirement dans NP et il est au moins aussi difficile que tous les problèmes appartenant à la classe NP (en ce sens, il est au moins aussi difficile qu'un problème NP-Complet). En d'autres termes, le problème WCSP est au moins aussi difficile que le problème CSP : en effet, on ne se contente pas de trouver une solution à un WCSP, on cherche la meilleure solution !

Les WCSP correspondent à des problèmes d'optimisation combinatoire : il s'agit de trouver dans l'ensemble fini des solutions (généralement conséquent) de ces problèmes la solution la meilleure selon une fonction d'*objectif* ou *économique* qui est le critère caractérisant la qualité d'une solution pour ce problème. Dans la majorité des cas, les techniques pour résoudre ces problèmes consistent à optimiser, c'est à dire minimiser ou maximiser selon le cas, cette fonction d'objectif. La solution optimisant la fonction d'objectif définie pour un problème représente une solution optimale à ce problème. Un algorithme basé sur la méthode de *séparation et évaluation* (appelée aussi $B\&B$ pour *Branch and Bound*) représente pour le moment la meilleure approche complète pour trouver la solution optimale à des problèmes d'optimisation combinatoire. Pour les problèmes vraiment difficiles, des approches approximatives (et donc incomplètes) existent et permettent généralement de trouver une "bonne" solution (c'est à dire proche de la solution optimale ou la solution optimale elle-même dans certains cas) dans un temps raisonnable. Nous présentons ces deux types de méthodes dans les sections suivantes.

Méthode de résolution complète : Séparation et Évaluation (B&B pour Branch & Bound)

La méthode complète de *séparation et évaluation* permet donc notamment de résoudre des problèmes d'optimisation dont le but est défini par :

$$min\ F(s)_{s\in\mathbb{S}_p} \tag{2.6}$$

ou

$$max\ F(s)_{s\in\mathbb{S}_p} \tag{2.7}$$

où F est la fonction d'objectif à minimiser (maximiser) et $\mathbb{S}_p$ représente l'ensemble des solutions du problème P. Elle se compose de trois procédures principales [Teghem, 2012] :

– Procédure de *séparation* : elle consiste à diviser chaque problème (ensemble de solutions) en sous-problèmes (sous-ensembles de solutions). Ces divisions successives sont représentées par un arbre dont la racine représente le problème d'origine et les autres nœuds les sous-problèmes qui en résultent. La manière de diviser et le nombre de sous-problèmes obtenu à l'issue de chaque division sont variables et ajustables selon le problème traité. Aucune solution n'est supprimée durant la séparation qui vérifie :

$$\bigcup_{i=1}^{n} \mathbb{S}_p^{(i)} = \mathbb{S}_p \tag{2.8}$$

où $\mathcal{S}_p^{(i)}$ représente un sous-ensemble de solutions parmi les n sous-ensembles obtenus après la séparation de l'ensemble de solutions $\mathcal{S}_p$.

– Procédure d'*évaluation* : elle consiste à obtenir un minorant de la valeur optimale de la fonction d'objectif d'un sous-problème : il s'agit d'une borne inférieure (resp. supérieure) dans le cas d'une minimisation (resp. maximisation) définie par :

$$lb_i \leq \tilde{F}_i = min\ F(s)_{s \in \mathcal{S}_p^{(i)}} \tag{2.9}$$

Soit lb_i la borne retournée par la fonction d'évaluation pour le sous-problème $\mathcal{S}_p^{(i)}$ et $\tilde{F}_i$ la valeur optimale de la fonction objectif du sous-problème $\mathcal{S}_p^{(i)}$. L'efficacité d'un algorithme de *séparation et évaluation* dépend directement de la qualité de la borne retournée par la fonction d'évaluation, autrement dit l'écart $\tilde{F}_i - lb_i$ doit être le plus faible possible. Soit $\hat{s}$ la solution optimale parmi les solutions déjà trouvées pour le problème P avec $\hat{F} = F(\hat{s})$. Un nœud est dit *sondé* si $lb_i \geq \hat{F}$. Si un nœud analysé est qualifié de nœud *sondé*, alors il est inutile de continuer à progresser dans ce sous-problème dont la meilleure solution que l'on puisse espérer y trouver est moins bonne, d'après la valeur du minorant, que la meilleure solution trouvée à ce moment de l'exploration. La branche de ce sous-problème peut donc être ignorée (coupée) sans inquiétude et on revient en arrière dans l'arbre de parcours. Si le nœud n'est pas *sondé*, on peut toujours espérer améliorer la solution optimale trouvée en continuant d'explorer les branches résultantes de la *séparation* de ce nœud.

– Procédure de *cheminement* : elle représente la manière d'explorer l'ensemble des solutions du problème d'origine et elle est dépendante des procédures d'évaluation et de séparation.

Nous verrons donc dans la section 2.4 que la stratégie de recherche complète la plus utilisée pour le WCSP consiste effectivement en une recherche arborescente de type *séparation et évaluation*. À partir du problème d'origine, la *séparation* en sous-problèmes à chaque nœud de l'arbre sera réalisé par les choix d'affectation de valeur pour les variables, plus ou moins efficacement par le biais d'heuristiques de choix. Chacun des sous-problèmes ainsi obtenu pourra être *évalué* par des méthodes de calcul de minorants abordées dans la section 2.3. Une borne inférieure pour chaque sous-problème, soit obtenue à partir du sous-problème devenu cohérent (comme expliqué dans les sections 2.3.2 et 2.3.3), soit calculée par des fonctions d'estimation n'utilisant pas les cohérences locales souples (comme expliqué dans la section 2.3.5), sera exploitée pour évaluer le sous-problème. En effet, les sous-problèmes remontant une borne inférieure ne permettant pas d'améliorer la meilleure solution déjà trouvée ne seront pas *séparés* et la branche correspondante dans l'arbre ne sera pas explorée : un retour en arrière sera alors réalisé.

Méthodes de résolution incomplètes : heuristiques et méta-heuristiques

Lorsque les problèmes d'optimisation combinatoires sont complexes au point que la méthode de *séparation et évaluation* ne permet pas d'obtenir dans un temps acceptable une solution prouvée comme étant solution optimale (voir même aucune solution), les méthodes incomplètes (*heuristiques* et *méta-heuristiques*) parviennent généralement à déterminer une "bonne" solution, c'est à dire une solution dont la valeur de la fonction objectif peut s'avérer être proche (ou égale, mais sans pouvoir le prouver) de la valeur optimale de la fonction objectif du problème. Les meilleures méthodes incomplètes sont celles qui vont retourner une "bonne" solution le plus rapidement possible. Parmi les méthodes *heuristiques*, on trouve les approches *par construction* et *gloutonnes* qui construisent une solution au fur et à mesure, puis les approches de *recherche locale* qui améliorent une solution initiale par transformations successives jusqu'à l'obtention d'une solution meilleure (selon la fonction d'objectif) : la nouvelle solution obtenue après chaque itération représente une génération et appartient au voisinage (des solutions très

proches, c'est à dire présentant peu de différences) de la solution de la génération précédente. Leurs constructions étant basées la plupart du temps sur la structure du problème à résoudre (plus précisément une structure de voisinage bien définie), ces *heuristiques* présentent malheureusement l'inconvénient de retourner généralement une solution correspondant à un optimum local de la fonction d'objectif (pour ce voisinage) et non un optimum global (minimums locaux et globaux dans le cadre d'une minimisation). Comme illustré à la figure 2.8, la fonction d'objectif est souvent représentée par un paysage (plan) où les abscisses représentent les instances de solution s et les ordonnées les valeurs de la fonction d'objectif pour ces solutions $F(s)$. La courbe liant les valeurs des fonctions d'objectif des différentes solutions forme des "vallées" et des "cols". L'optimum local (minimum local dans notre cas) pour un voisinage de solutions est la solution pour laquelle la fonction d'objectif est minimale dans la "vallée" formée par ces solutions.

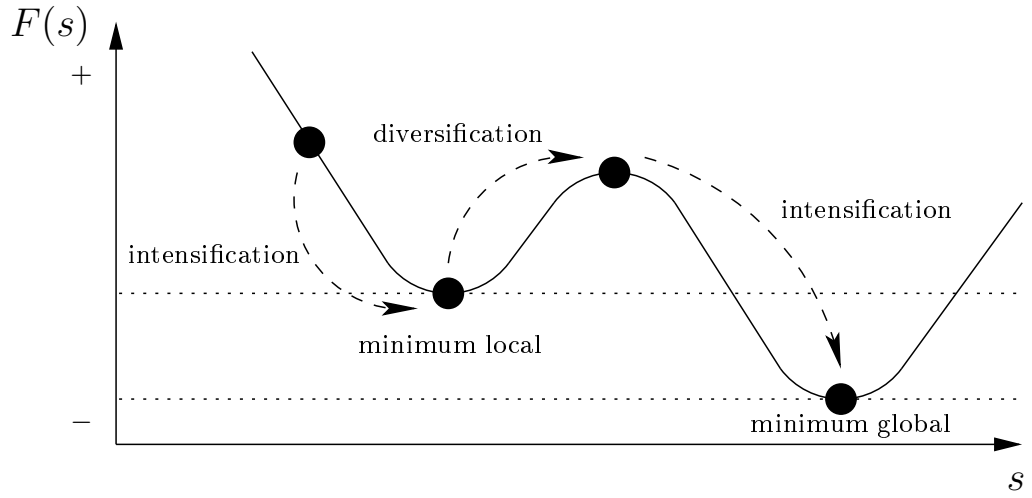


FIG. 2.8 – Diversifier pour s'échapper d'un minimum local, intensifier pour atteindre le minimum global.

Pour pallier cet inconvénient, des méthodes plus générales, appelées *méta-heuristiques*, ont été proposées afin de proposer des modèles pour ne plus bloquer sur ces optimums locaux et parvenir ainsi à atteindre un optimum global (du moins s'en approcher) grâce à l'utilisation de différentes techniques paramétrables. Nous distinguons deux grandes familles de *méta-heuristiques* : les *méta-heuristiques* de *recherche locale* et les *méta-heuristiques évolutionnaires*. Ces familles, basées sur la transformation de solutions admissibles, diffèrent dans le sens où lors de chaque itération la recherche locale considère la transformation d'une unique solution tandis que l'approche évolutionnaire considère la transformation d'une population de solutions. Cependant, l'efficacité de ces deux familles de *méta-heuristiques* résulte dans la mise en place de deux mécanismes : l'*intensification* et la *diversification* (figure 2.8). L'*intensification* consiste à effectuer les transformations qui améliorent la fonction d'objectif de la (ou des) solution(s) courante(s) : il s'agit en fait de "s'enfoncer" le plus possible dans une "vallée" jusqu'à atteindre un optimum local (qui est peut être un optimum global). La *diversification* consiste à accepter de dégrader la fonction d'objectif de la (ou des) solution(s) courante(s) afin de s'échapper d'un minimum local : il s'agit de "s'échapper" d'une vallée en passant par un "col", représentant une dégradation (plus ou moins forte selon la hauteur du "col" escaladé) de la fonction d'objectif, afin de "s'enfoncer" ensuite dans une "vallée" qui contiendra peut être soit un optimum global, soit un optimum local meilleur (voir aussi un optimum local moins bon...). Parmi les *méta-heuristiques* de recherche locale on peut citer le *recuit simulé* [Kirkpatrick *et al.*, 1983] et la *recherche avec tabous* [Glover, 1989] et pour les *méta-heuristiques évolutionnaires* il y a les algorithmes *génétiques* [Holland, 1975, Goldberg, 1989] et

ceux basés sur les *colonies de fourmis* [Dorigo *et al.*, 1996]. L'inconvénient des *heuristiques* et des *méta-heuristiques* provient du fait qu'elles nécessitent un travail important de paramétrage qui est spécifique à chaque problème : en effet, une de ces méthodes peut très bien fonctionner avec certains problèmes et moins bien avec d'autres. Ce paramétrage consiste à définir les types de transformations à effectuer et les critères de sélection des meilleurs individus d'une génération de solution(s) à une autre, la manière d'alterner les phases d'*intensification* et de *diversification*, les critères d'arrêt : intensifier durant combien d'itérations ? diversifier pendant combien d'itérations ? combien d'itérations pour trouver l'optimum global ? Il est intéressant de noter également que les méthodes approximatives peuvent être employées conjointement avec une approche complète. En effet, la solution trouvée par la méthode incomplète, supposée de bonne qualité, peut être utile dans une approche de *séparation et évaluation* : elle peut servir de borne supérieure et peut permettre de détecter des nœuds sondés. On parle alors d'approches *hybrides*. Bien évidemment, ces méthodes approximatives que nous venons de présenter peuvent également être exploitées dans le cadre WCSP, une description étant proposée dans la section 2.5.

Les solveurs de contraintes, exploitant les méthodes de résolution complètes expliquées précédemment, résolvent donc les WCSP en alternant des phases de recherche (séparation) et des phases d'inférence (évaluation). Nous présenterons donc tout d'abord ces deux phases et leurs particularités dans les sections 2.3 et 2.4. Ensuite, nous présenterons également quelques méthodes de résolution incomplètes adaptées au cadre WCSP dans la section 2.5.

2.3 Inférence

Des cohérences locales (souples) ont également été proposées dans le cadre WCSP et permettent comme nous le verrons une résolution plus efficace. Au delà de simplifier un réseau en supprimant des valeurs des domaines qui ne peuvent participer à une solution, comme c'est le cas dans le cadre des CSP, ces cohérences sont basées sur le calcul d'un minorant représentant une borne inférieure du coût de n'importe quelle solution pouvant être obtenue dans ce réseau et supprimer ainsi les branches de recherche ne permettant pas d'obtenir de meilleure solution que la meilleure solution trouvée jusqu'à présent, c'est à dire la borne supérieure. De plus, le calcul de ces bornes peut également être utilisé pour obtenir des informations sur les variables et les valeurs et permettre ainsi de renforcer les heuristiques utilisées pendant la résolution en déterminant des bons candidats, basés sur le critère de minimalité des coûts des solutions auxquelles elles peuvent appartenir. Il est inutile de préciser que plus une borne inférieure est de qualité, plus une recherche arborescente par séparation et évaluation est efficace. Nous distinguons deux manières de calculer cette borne inférieure. D'une part, la borne peut donc être obtenue par l'établissement de cohérences grâce à des transformations préservant l'équivalence des réseaux. En fonction d'une cohérence plus ou moins forte établie, nous verrons que la borne obtenue sera plus ou moins de bonne qualité. D'autre part, nous verrons qu'une borne peut également être obtenue sans modification du réseau par une estimation par calcul de cette borne : il s'agit d'une sous-estimation du coût de n'importe quelle solution pouvant être obtenue dans ce réseau. Nous abordons donc ces deux types d'approches dans les sections suivantes.

2.3.1 Transformations préservant l'équivalence

Les cohérences locales souples présentées dans les sections 2.3.2 et 2.3.3 vont être établies grâce à des transformations préservant l'équivalence d'un réseau (notées souvent *EPT* pour Equivalence Preserving Transformation) et introduites par la définition 37. Ces opérations sont appelées aussi *transferts de coûts*

car elles consistent à transférer des coûts entiers[9] entre les différentes contraintes du réseau. La nature d'une *EPT* est caractérisée par le type de contrainte à la source du transfert et le type de contrainte à la destination du transfert : la contrainte 0-aire $c_\emptyset$, une contrainte unaire ou une contrainte binaire/n-aire. Toute séquence d'*EPT* transforme donc un WCSP en WCSP équivalent. Nous présentons ici les trois *EPT* de base nécessaires pour établir les cohérences présentées dans la section suivante : il s'agit de la *projection*, de l'*extension* et de la *projection unaire* [Schiex, 2000, Larrosa, 2002].

La projection

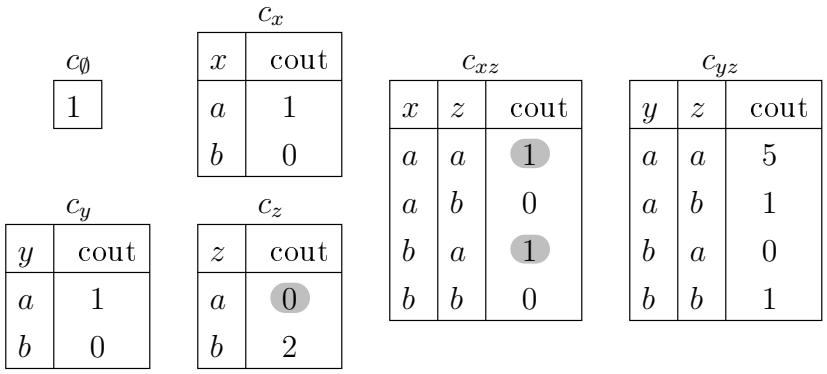
L'opération de *projection* consiste à transférer des coûts depuis une contrainte binaire ou n-aire jusqu'à une contrainte unaire. Considérons l'algorithme 12 : il s'agit de projeter à partir de la contrainte souple binaire ou n-aire c_S un coût α sur la contrainte souple unaire c_x, et plus précisément sur le coût affecté par la contrainte c_x à la valeur a, c'est à dire $c_x(a)$. Autrement dit, à la manière d'une factorisation, le coût α est ajouté au coût représenté par $c_x(a)$, puis il est soustrait de tous les tuples appartenant à la contrainte c_S et contenant la valeur a pour la variable x de la portée de c_S. Le coût α est positif et il est inférieur ou égal au coût minimal des coûts des tuples contenant la valeur (x, a) dans la contrainte c_S. L'objectif direct de cette opération est de déplacer les coûts du réseau au niveau des variables et plus précisément au niveau des contraintes unaires portant sur ces variables. Nous verrons dans la suite que cela va permettre de déceler et de supprimer plus efficacement les valeurs incohérentes dans les domaines.

Algorithm 12: projection(c_S : contrainte souple, x : variable, a : valeur, α : entier)

1 **Prérequis:** $x \in S \wedge a \in dom(x) \wedge 0 \leq \alpha \leq min\{c_S(\tau) \mid \tau \in l(S) \wedge \tau[x] = a\}$
$c_x(a) \leftarrow c_x(a) \oplus \alpha$;
2 **pour chaque** $\tau \in l(S) \mid \tau[x] = a$ **faire**
3 $\quad c_S(\tau) \leftarrow c_S(\tau) \ominus \alpha$;

Exemple 11 *Considérons le réseau introduit dans l'exemple 10 : nous souhaitons projeter un coût de* 1 *depuis la contrainte binaire* c_{xz} *vers la contrainte unaire* $c_z(a)$*, c'est à dire réaliser l'opération projection(*c_{xz}*,*z*,*a*,*1*) dans ce réseau. La figure 2.9(a) représente la répartition des coûts avant cette opération avec notamment* $c_z(a) = 0$*,* $c_{xz}((a, a)) = 1$ *et* $c_{xz}((b, a)) = 1$ *qui sont grisés car directement impactés. Dans un premier temps, un coût de* 1 *est ajouté au coût affecté par la contrainte unaire* c_z *à la valeur* a*, ce qui est représenté par l'action* $\oplus 1$ *sur* $c_z(a)$ *dans la figure 2.9(b). Dans un deuxième temps, un coût de* 1 *est retiré aux coûts affectés par la contrainte binaire* c_{xz} *aux tuples contenant la valeur* a *pour la variable* z*, ce qui est représenté par les actions* $\ominus 1$ *sur* $c_{xz}((a, a))$ *et* $c_{xz}((b, a))$ *dans la figure 2.9(b). La figure 2.9(c) représente la répartition des coûts après cette opération : les coûts* $c_z(a) = 1$*,* $c_{xz}((a, a)) = 0$ *et* $c_{xz}((b, a)) = 0$ *ont évolué. De plus, nous constatons bien dans cet exemple que le réseau obtenu après l'opération de projection est équivalent au réseau d'origine. En effet, considérons par exemple l'instanciation complète* $I = \{(x, a), (y, b), (z, a)\}$ *: dans le réseau représenté dans la figure 2.9(a), nous avons* $\mathcal{V}(I) = c_\emptyset \oplus c_x(I) \oplus c_y(I) \oplus c_z(I) \oplus c_{xz}(I) \oplus c_{yz}(I) = 1 \oplus c_x(a) \oplus c_y(b) \oplus c_z(a) \oplus c_{xz}((a, a)) \oplus c_{yz}((b, a)) = 1 \oplus 1 \oplus 0 \oplus 0 \oplus 1 \oplus 0 = 3$*; dans le réseau représenté dans la figure 2.9(c), nous avons toujours* $\mathcal{V}(I) = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 0 = 3$*.*

[9] Transférer un coût nul est possible, mais cela ne sert bien évidemment pas à grand chose car le réseau reste inchangé



$c_\emptyset$

1

c_x

x	cout
a	1
b	0

c_{xz}

x	z	cout
a	a	1
a	b	0
b	a	1
b	b	0

c_{yz}

y	z	cout
a	a	5
a	b	1
b	a	0
b	b	1

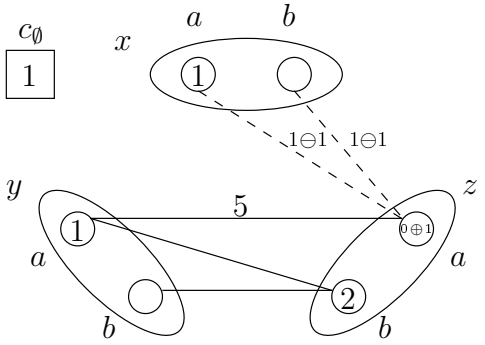
c_y

y	cout
a	1
b	0

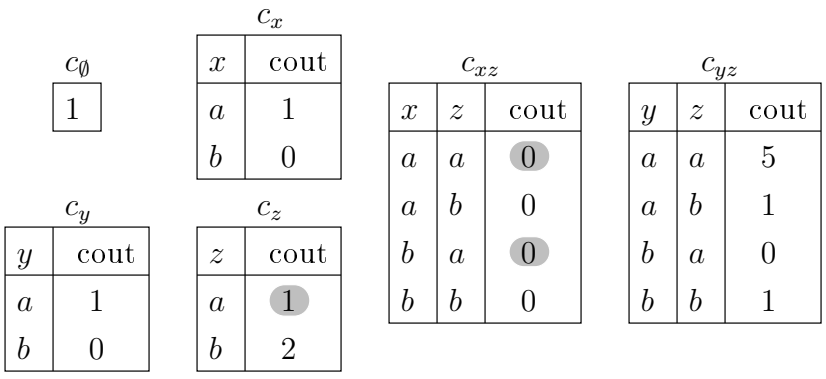
c_z

z	cout
a	0
b	2

(a) Réseau initial



(b) projection(c_{xz},z,a,1) dans le réseau



$c_\emptyset$

1

c_x

x	cout
a	1
b	0

c_{xz}

x	z	cout
a	a	0
a	b	0
b	a	0
b	b	0

c_{yz}

y	z	cout
a	a	5
a	b	1
b	a	0
b	b	1

c_y

y	cout
a	1
b	0

c_z

z	cout
a	1
b	2

(c) Réseau équivalent obtenu après projection(c_{xz},z,a,1)

FIG. 2.9 – L'opération de projection.

L'extension

À l'inverse de la projection, l'opération *extension* consiste à transférer des coûts depuis une contrainte unaire jusqu'à une contrainte binaire ou n-aire. Considérons l'algorithme 13 : il s'agit de transférer à partir de la contrainte souple unaire $c_x(a)$ un coût α sur la contrainte souple binaire ou n-aire c_S. Cette fois-ci, dans l'idée d'un développement, le coût α est ajouté à tous les tuples appartenant à la contrainte c_S et contenant la valeur a pour la variable x de la portée de c_S, puis il est soustrait de la contrainte unaire $c_x(a)$. Le coût α est positif et il est inférieur ou égal au coût $c_x(a)$. L'objectif direct de cette opération est de déplacer les coûts du réseau au niveau des tuples. Nous verrons dans la suite que cela va permettre de déceler et de supprimer des tuples incohérents, mais également de transférer des coûts d'une contrainte unaire à une autre.

Algorithm 13: extension(c_S : contrainte souple, x : variable, a : valeur, α : entier)

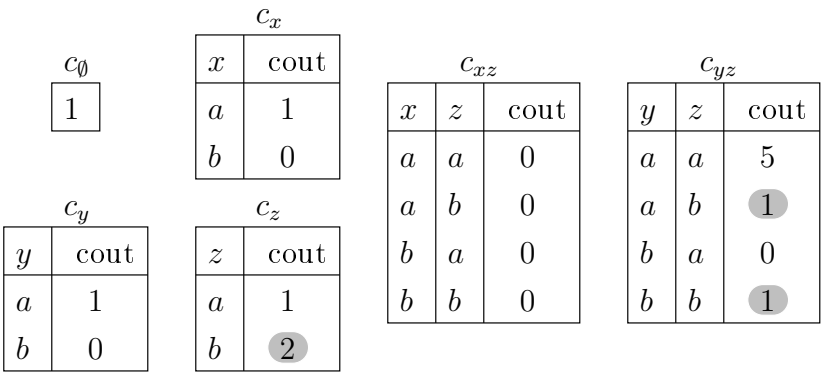
1 **Prérequis:** $x \in S \wedge a \in dom(x) \wedge 0 \leq \alpha \leq c_x(a)$
2 **pour chaque** $\tau \in l(S) \mid \tau[x] = a$ **faire**
3 $\quad c_S(\tau) \leftarrow c_S(\tau) \oplus \alpha$;
4 $c_x(a) \leftarrow c_x(a) \ominus \alpha$;

Exemple 12 *Considérons le réseau illustré à la figure 2.10(a) : nous souhaitons étendre un coût de* 1 *depuis la contrainte unaire* $c_z(b)$ *vers la contrainte binaire* c_{yz}*, c'est à dire réaliser l'opération extension(*c_{yz}*,*z*,*b*,*1*) dans ce réseau. La figure 2.10(a) représente la répartition des coûts avant cette opération avec notamment* $c_z(b) = 2$*,* $c_{yz}((a,b)) = 1$ *et* $c_{yz}((b,b)) = 1$*. Dans un premier temps, un coût de* 1 *est ajouté aux coûts affectés par la contrainte binaire* c_{yz} *aux tuples contenant la valeur* b *pour la variable* z*, ce qui est représenté par les actions* $\oplus 1$ *sur* $c_{yz}((a,b))$ *et* $c_{yz}((b,b))$ *dans la figure 2.10(b). Dans un deuxième temps, un coût de* 1 *est retiré au coût affecté par la contrainte unaire* c_z *à la valeur* b*, ce qui est représenté par l'action* $\ominus 1$ *sur* $c_z(b)$ *dans la figure 2.10(b). La figure 2.10(c) représente la répartition des coûts après cette opération : les coûts* $c_z(b) = 1$*,* $c_{xz}((a,a)) = 2$ *et* $c_{xz}((b,a)) = 2$ *ont évolué. De plus, nous constatons là aussi dans cet exemple que le réseau obtenu après l'opération d'extension est équivalent au réseau d'origine. En effet, considérons par exemple l'instanciation complète* $I = \{(x,a),(y,b),(z,b)\}$ *: dans le réseau représenté dans la figure 2.10(a), nous avons* $\mathcal{V}(I) = c_\emptyset \oplus c_x(I) \oplus c_y(I) \oplus c_z(I) \oplus c_{xz}(I) \oplus c_{yz}(I) = 1 \oplus c_x(a) \oplus c_y(b) \oplus c_z(b) \oplus c_{xz}((a,b)) \oplus c_{yz}((b,b)) = 1 \oplus 1 \oplus 0 \oplus 2 \oplus 0 \oplus 1 = 5$*; dans le réseau représenté dans la figure 2.10(c), nous avons toujours* $\mathcal{V}(I) = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 2 = 5$.

Comme nous l'avons dit, l'*extension* correspond à l'opération inverse de la *projection*. Bien que les coûts transférés par les transformations préservant l'équivalence ne sont pas négatifs, nous admettrons par convention que l'extension correspond à une projection d'un coût négatif. D'une part, nous verrons que cette convention est très importante pour établir la cohérence d'arc souple optimale OSAC présentée dans la section 2.3.3. D'autre part, c'est cette convention que nous avons suivie pour notre implémentation.

La projection unaire

L'opération de projection unaire consiste à projeter des coûts depuis une contrainte unaire jusqu'à la contrainte 0-aire $c_\emptyset$. Considérons l'algorithme 14 : il s'agit de projeter à partir de la contrainte souple unaire c_x un coût α sur la contrainte $c_\emptyset$. Le coût α est soustrait depuis la contrainte unaire c_x pour chaque valeur a appartenant au domaine de x, puis il est ajouté à la contrainte $c_\emptyset$. Le coût α est positif et il est inférieur ou égal au coût minimal des coûts unaires des valeurs du domaine de la variable x. L'objectif



$c_\emptyset$

1

c_x

x	cout
a	1
b	0

c_{xz}

x	z	cout
a	a	0
a	b	0
b	a	0
b	b	0

c_{yz}

y	z	cout
a	a	5
a	b	1
b	a	0
b	b	1

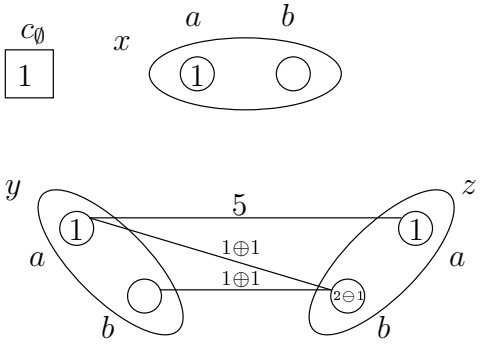
c_y

y	cout
a	1
b	0

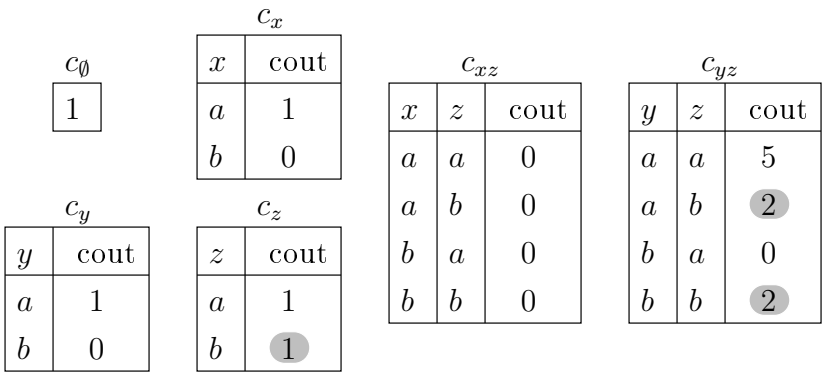
c_z

z	cout
a	1
b	2

(a) Réseau initial



(b) extension(c_{yz},z,b,1) dans le réseau



$c_\emptyset$

1

c_x

x	cout
a	1
b	0

c_{xz}

x	z	cout
a	a	0
a	b	0
b	a	0
b	b	0

c_{yz}

y	z	cout
a	a	5
a	b	2
b	a	0
b	b	2

c_y

y	cout
a	1
b	0

c_z

z	cout
a	1
b	1

(c) Réseau équivalent obtenu après extension(c_{yz},z,b,1)

FIG. 2.10 – L'opération d'extension.

direct de cette opération est de rassembler les coûts du réseau dans la contrainte $c_\emptyset$ et nous verrons dans la suite que cela va permettre de calculer une borne inférieure pour le réseau.

Algorithm 14: projectionUnaire(x : variable, α : entier)

1 **Prérequis:** $0 \leq \alpha \leq min\{c_x(a) \mid a \in dom(x)\}$
2 **pour chaque** *valeur* $a \in dom(x)$ **faire**
3 $\quad c_x(a) \leftarrow c_x(a) \ominus \alpha$;
4 $c_\emptyset \leftarrow c_\emptyset \oplus \alpha$;

Exemple 13 *Considérons le réseau illustré à la figure 2.11(a) : nous souhaitons réaliser une projection unaire d'un coût de* 1 *depuis la contrainte unaire* c_z *vers la contrainte binaire* $c_\emptyset$*, c'est à dire réaliser l'opération projectionUnaire(*z*,*1*) dans ce réseau. La figure 2.11(a) représente la répartition des coûts avant cette opération avec notamment* $c_\emptyset = 1$*,* $c_z(a) = 1$ *et* $c_z(b) = 1$*. Dans un premier temps, un coût de* 1 *est retiré aux coûts affectés par la contrainte unaire* c_z *aux valeurs* a *et* b*, ce qui est représenté par l'action* $\ominus 1$ *sur* $c_z(a)$ *et* $c_z(b)$ *dans la figure 2.11(b). Dans un deuxième temps, un coût de* 1 *est ajouté au coût de la contrainte* $c_\emptyset$*, ce qui est représenté par l'action* $\oplus 1$ *sur* $c_\emptyset$ *dans la figure 2.11(b). La figure 2.11(c) représente la répartition des coûts après cette opération : les coûts* $c_\emptyset = 2$*,* $c_z(a) = 0$ *et* $c_z(b) = 0$ *ont évolué. De plus, nous constatons là encore dans cet exemple que le réseau obtenu après l'opération de projection est équivalent au réseau d'origine. En effet, considérons par exemple l'instanciation complète* $I = \{(x, a), (y, b), (z, b)\}$ *: dans le réseau représenté dans la figure 2.11(a), nous avons* $\mathcal{V}(I) = c_\emptyset \oplus c_x(I) \oplus c_y(I) \oplus c_z(I) \oplus c_{xz}(I) \oplus c_{yz}(I) = 1 \oplus c_x(a) \oplus c_y(b) \oplus c_z(b) \oplus c_{xz}((a, b)) \oplus c_{yz}((b, b)) = 1 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 2 = 5$ *; dans le réseau représenté dans la figure 2.11(c), nous avons toujours* $\mathcal{V}(I) = 2 \oplus 1 \oplus 0 \oplus 0 \oplus 0 \oplus 2 = 5$*.*

Nous allons voir dans la section suivante que l'objectif de ces transferts de coût est de centraliser les différents coûts d'un réseau.

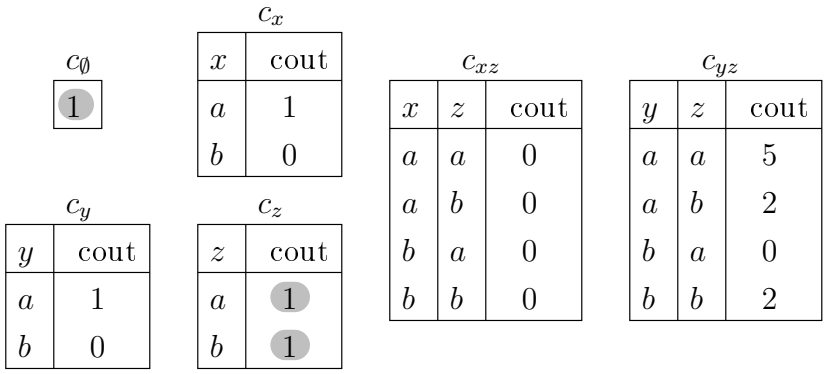
2.3.2 Cohérences locales souples

Dans un souci de clarté et de simplicité, nous présenterons ces cohérences locales souples dans le cadre de réseaux binaires. De plus, nous supposerons l'existence de contraintes unaires portant sur chacune des variables du réseau. Les réseaux utilisés dans les exemples sont basés sur la même structure que le réseau introduit dans la figure 2.9(a), c'est à dire avec les variables x, y, z et les contraintes $c_\emptyset$, c_{xz} et c_{yz}, seuls les coûts (unaires ou binaires) peuvent être différents. À noter que ces différentes cohérences ont été généralisées au cas n-aire dans [Cooper et Schiex, 2004, Sanchez *et al.*, 2008, Lee et Leung, 2009, Cooper *et al.*, 2010, Lee et Leung, 2010].

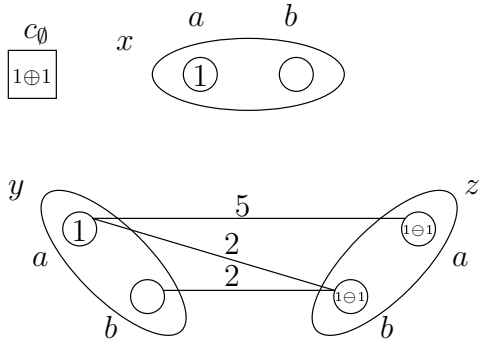
La cohérence de nœud NC* [Larrosa, 2002]

Définition 47 (Cohérence de nœud NC*) *Une valeur* (x, a) *est nœud-cohérente (NC*) si* $c_\emptyset \oplus c_x(a) < k$*. Une variable* x *est nœud-cohérente (NC*) si toutes les valeurs appartenant à* $dom(x)$ *sont nœud-cohérentes et s'il existe au moins une valeur* (x, a) *telle que* $c_x(a) = 0$*. La valeur* (x, a) *est alors appelée support de la variable* x*. Un réseau est nœud-cohérent (NC*) si toutes les variables appartenant à ce réseau sont nœud-cohérentes.*

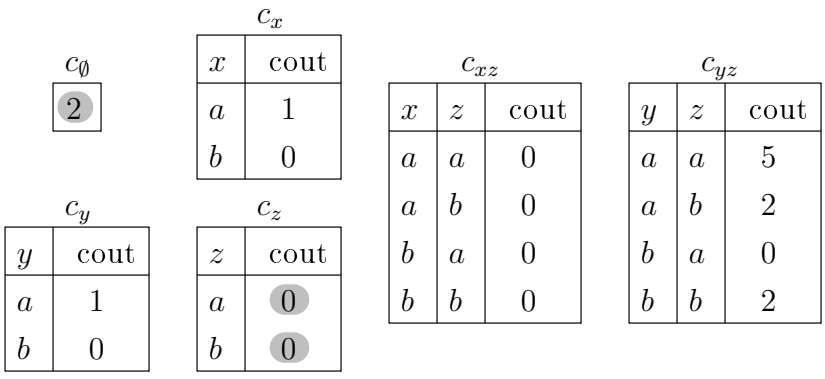
L'idée de cette propriété est que, au delà de supprimer les valeurs amenant à une solution de coût interdit, si les coûts unaires de toutes les valeurs pour une variable sont différents de 0, c'est qu'il est



(a) Réseau initial



(b) projectionUnaire(z,1) dans le réseau



(c) Réseau équivalent obtenu après projectionUnaire(z,1)

FIG. 2.11 – L'opération de projection unaire.

encore possible d'effectuer une projection unaire depuis cette variable et d'augmenter la contrainte $c_\emptyset$. La cohérence de nœud d'un réseau va être établie par la méthode *NC** (algorithme 15) en deux phases. Dans une première phase, la méthode *trouverSupport* (algorithme 16) obtient une valeur support pour chaque variable de ce réseau grâce à des projections unaires sur $c_\emptyset$ d'un coût α égal au coût minimal des coûts unaires des différentes valeurs de chacune de ces variables. Notons qu'il n'est pas encore utile pour l'instant de s'intéresser à la structure $Supp$ (ligne 1) qui est une optimisation utilisée dans les algorithmes pour les cohérences présentées par la suite. Il faut retenir ici que l'on sélectionne la valeur de la variable pour laquelle le coût unaire correspond au coût minimal[10]. Dans une deuxième phase, la méthode *filtrerDomaine* (algorithme 17) supprime dans les domaines les valeurs qui ne sont pas nœud-cohérentes. Si un domaine vide apparaît en établissant la cohérence de nœud du réseau, alors celui-ci est nœud-incohérent et faux est retourné, sinon vrai est retourné : le réseau obtenu est alors nœud-cohérent.

Algorithm 15: NC* ($P = (\mathcal{X}, \mathcal{C}, k)$: WCSP) : Booléen

```
1 pour chaque variable x ∈ 𝒳 faire
2     trouverSupport(x)
3 pour chaque variable x ∈ 𝒳 faire
4     si filtrerDomaine(x) alors
5         si dom(x) = ∅ alors
6             Retourner faux
7 Retourner vrai
```

Algorithm 16: trouverSupport (x : variable)

```
1 Supp(x) = argmin_{a∈dom(x)}{c_x(a)}
2 α = c_x(Supp(x))
3 projectionUnaire(x, α)
```

Algorithm 17: filtrerDomaine (x : variable) : Booléen

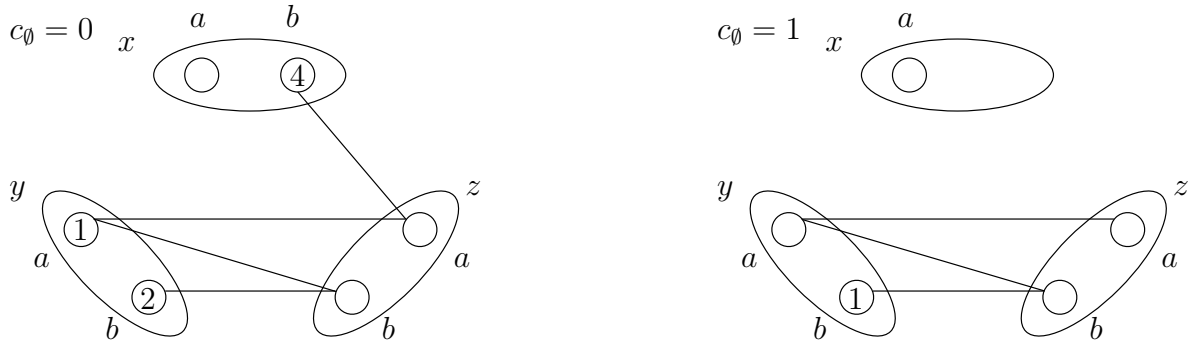
```
1 suppression ← faux
2 pour chaque valeur a ∈ dom(x) faire
3     si c_∅ ⊕ c_x(a) = k alors
4         dom(x) ← dom(x) \ {a}
5         suppression ← vrai
6 Retourner suppression
```

L'algorithme *NC** (algorithme 15) pour établir la cohérence de nœud NC* a une complexité en temps $O(nd)$ où n correspond au nombre de variables et d la taille maximale parmi les domaines des variables, et une complexité en espace $O(nd)$.

Exemple 14 *Le réseau WCSP (avec $k = 4$) illustré dans la figure 2.12(a) n'est pas NC* parce que les variables x et y ne sont pas NC* : il existe une valeur dans le domaine de la variable x qui n'est pas NC*,*

[10] $argmin_{a \in dom(x)}\{c_x(a)\}$, ou argument minimum de la fonction de coût c_x, représente la valeur du domaine de la variable x de coût minimal

en l'occurrence (x, b) car $c_x(b) = 4 = k$, puis la variable y ne possède pas de support. Pour établir NC sur ce réseau, une projection unaire d'un coût égal à 1 est tout d'abord effectuée à partir de la contrainte c_y ($c_\emptyset$ est incrémentée de 1) afin d'obtenir une valeur support pour la variable y, en l'occurrence (y, a). Ensuite, la valeur nœud-incohérente (x, b) est supprimée du domaine de x. Le réseau obtenu à la figure 2.12(b) est NC*.*



(a) Réseau non NC*

(b) Réseau NC*

FIG. 2.12 – Deux réseaux WCSP équivalents (avec $k = 4$).

Dans cet exemple, nous avons montré qu'établir la cohérence de nœud NC* sur ce réseau nous a permis d'augmenter (et donc d'améliorer) la borne inférieure $c_\emptyset$ correspondante. La cohérence de nœud NC* représente la cohérence locale souple basique, et nous allons voir que $c_\emptyset$ peut encore être améliorée en utilisant une cohérence locale plus forte que NC*, notamment la cohérence d'arc AC*.

La cohérence d'arc AC* [Larrosa, 2002, Larrosa et Schiex, 2003]

Définition 48 (Cohérence d'arc AC*) *Une valeur (x, a) est arc-cohérente (AC) pour la contrainte c_{xy} s'il existe une valeur (y, b) telle que $c_{xy}(a, b) = 0$. La valeur (y, b) est alors appelée support (simple) de la valeur (x, a) dans c_{xy}. La variable x est arc-cohérente si toutes les valeurs appartenant à $dom(x)$ sont arc-cohérentes dans toutes les contraintes binaires couvrant x. Un réseau est arc-cohérent (AC*) si toutes les variables appartenant à ce réseau sont arc-cohérentes (AC) et nœud-cohérentes (NC*).*

L'idée de cette propriété est que s'il n'existe aucun support pour une valeur dans une contrainte, c'est qu'il est possible de ramener des coûts depuis cette contrainte sur le coût unaire de cette valeur et potentiellement transférer des coûts depuis la contrainte unaire correspondante sur la contrainte $c_\emptyset$.

La cohérence d'arc (AC*) d'un réseau peut être établie efficacement par l'algorithme *AC*2001* (algorithme 18), proposé par [Larrosa, 2002] et inspiré de l'algorithme proposé par [Bessière et Régin, 2001] dans le cadre CSP (abordé dans la section 1.2.2 du chapitre 1). En ce sens, deux structures particulières sont utilisées : la structure $Supp(x, a, y)$ pour stocker la dernière valeur support trouvée pour la valeur (x, a) dans le domaine de la variable y pour la contrainte c_{xy}, puis la structure $Supp(x)$ pour stocker la valeur support courante pour la variable x. Avant d'établir la cohérence d'arc sur un réseau, celui-ci doit être nœud-cohérent (NC*). À partir d'un réseau NC*, une queue de propagation Q contenant initialement toutes les variables du réseau est alors parcourue. Tant que cette queue n'est pas vide, une variable est prélevée. Dans une première phase, la méthode *trouverSupportSimple* (algorithme 19) obtient un support dans toutes les contraintes du réseau couvrant cette variable pour chaque valeur appartenant à son domaine. Il n'est pas utile pour l'instant de s'intéresser à la structure R (ligne 7) exploitée dans les algorithmes présentés pour les cohérences suivantes. Ensuite, pour chaque variable du réseau, on supprime

les valeurs qui ne sont pas NC* via la méthode *filtrerDomaine* (les transferts de coûts sur des valeurs ont pu les faire devenir nœud-incohérentes) : si un domaine vide apparaît le processus est arrêté (ligne 10), sinon toute variable dont au moins une valeur a été supprimée du domaine est ajoutée dans la queue de propagation Q.

Algorithm 18: AC*2001 ($P = (\mathcal{X}, \mathcal{C}, k)$: WCSP) : Booléen

```
1  si NC*(P) alors
2      Q ← X
3      tant que Q ≠ ∅ faire
4          Choisir et éliminer y de Q
5          pour chaque contrainte c_xy faire
6              si trouverSupportSimple(x, y) alors
7                  R ← R ∪ {x}
8          pour chaque variable x ∈ X faire
9              si filtrerDomaine(x) alors
10                 si dom(x) = ∅ alors Retourner faux
11                 Q ← Q ∪ {x}
12     Retourner vrai
13 sinon
14     Retourner faux
```

La méthode *trouverSupportSimple* permet de trouver des supports pour les valeurs de la variable x dans le domaine de la variable y pour une contrainte c_{xy}. Pour toutes les valeurs (x, a) qui le nécessitent, c'est à dire celles dont le dernier support trouvé et stocké dans $Supp(x, a, y)$ n'appartient plus au domaine courant de y, on cherche dans le domaine de la variable y la valeur b pour laquelle le coût $c_{xy}(a, b)$ correspond au coût minimal[11] dans la contrainte c_{xy}. Cette valeur (y, b) trouvée devient alors la nouvelle valeur $Supp(x, a, y)$. Ensuite, une projection d'un coût égal à $c_{xy}(a, Supp(x, a, y))$ est effectuée sur $c_x(a)$. Une fois toutes ces variables (x, a) parcourues, on cherche une valeur support pour la cohérence de nœud NC* de la variable x. La valeur booléenne *transfertEffectue* précise si une projection a été réalisée et si la variable doit être ajoutée dans la queue R dont l'utilité sera détaillée dans les prochaines cohérences locales.

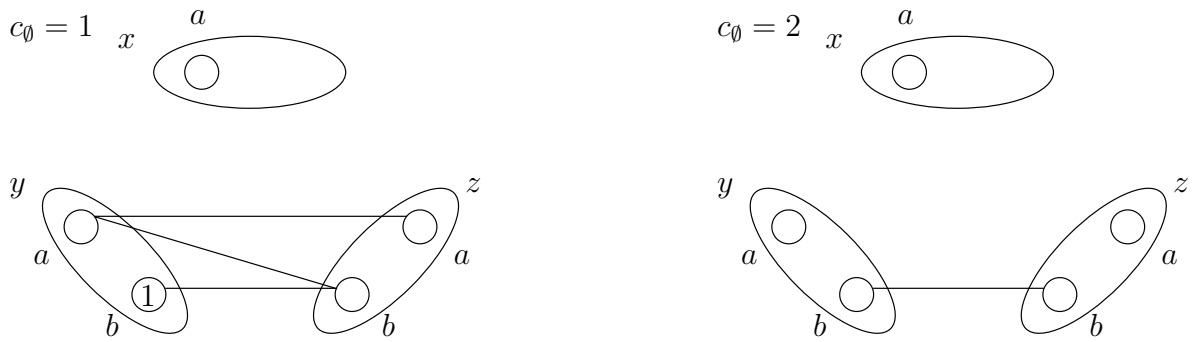
Algorithm 19: trouverSupportSimple (x : variable, y : variable) : Booléen

```
1  transfertEffectue ← faux
2  pour chaque valeur a ∈ dom(x) telle que Supp(x, a, y) ∉ dom(y) faire
3      Supp(x, a, y) = argmin_{b∈dom(y)}{c_xy(a, b)}
4      α = c_xy(a, Supp(x, a, y))
5      si c_x(a) = 0 ∧ α > 0 alors transfertEffectue ← vrai
6      projection(c_xy,x,a,α)
7  trouverSupport(x)
8  Retourner transfertEffectue
```

[11] $argmin_{b \in dom(y)}\{c_{xy}(a, b)\}$, ou argument minimum de la fonction de coût c_{xy}, représente la valeur b du domaine de la variable y pour laquelle (a, b) est de coût minimal dans c_{xy}

Dans le cadre d'un réseau binaire, l'algorithme *AC*2001* (algorithme 18) pour établir la cohérence d'arc AC* a une complexité en temps $O(n^2d^3)$ et une complexité en espace $O(ed)$ où n correspond au nombre de variables, e au nombre de contraintes et d est la taille maximale parmi les domaines des variables.

Exemple 15 *Le réseau WCSP (avec $k = 4$) illustré dans la figure 2.13(a) est NC* mais il n'est pas AC* parce que les variables y et z ne sont pas AC* : la valeur (y, a) (resp. la valeur (z, b)) ne possède pas de valeur support dans le domaine de la variable z (resp. y) pour la contrainte c_{yz}. Pour établir AC* sur ce réseau, une projection d'un coût égal à 1 est tout d'abord effectuée à partir de la contrainte c_{yz} sur $c_y(a)$ afin d'obtenir une valeur support dans le domaine de la variable z pour la valeur (y, a). Cette opération a pour conséquence de faire apparaître également une valeur support pour la valeur (z, b), en l'occurrence la valeur (y, a). Ensuite, pour respecter la cohérence de nœud NC*, une projection unaire d'un coût égal à 1 est effectuée depuis c_y. Le réseau obtenu à la figure 2.13(b) est AC*.*



(a) Réseau NC* mais non AC* (b) Réseau AC* après projection(c_{yz},y,a,1)

FIG. 2.13 – Deux réseaux WCSP équivalents (avec $k = 4$).

Dans cet exemple, nous avons montré qu'établir la cohérence d'arc AC* sur ce réseau nous a permis d'incrémenter encore la borne inférieure $c_\emptyset$ obtenue précédemment en établissant uniquement la cohérence de nœud NC*. Il est important également de savoir que la fermeture d'arc-cohérence AC* n'est pas unique contrairement à la fermeture d'arc-cohérence AC proposée dans le cadre CSP, ce que nous pouvons constater avec la figure 2.14. Il s'agit du réseau qui aurait été obtenu si on avait effectué l'opération projection(c_{yz},z,b,1) au lieu de projection(c_{yz},y,a,1). Ce réseau est bien lui aussi AC*, cependant on s'aperçoit que dans ce cas là, la borne $c_\emptyset$ calculée pour le réseau d'origine vaut 1 au lieu de 2, ce qui est moins bon. On remarque donc qu'au delà de la propriété de cohérence d'arc AC*, la qualité de la borne obtenue par $c_\emptyset$ dépend surtout de la séquence de transformations (EPT) réalisée pour y parvenir. Malheureusement, trouver une fermeture d'arc-cohérence optimale, c'est à dire la séquence d'EPT établissant AC* et maximisant la borne $c_\emptyset$, a été prouvé être un problème `NP-Difficile` [Cooper et Schiex, 2004] si l'on considère des coûts entiers. Pour s'orienter vers la séquence d'EPT permettant d'atteindre cette optimalité, ou tout du moins s'en approcher, des variantes de complexité polynomiale de la cohérence d'arc AC* ont ainsi été proposées. En effet, nous allons voir que la borne $c_\emptyset$ peut encore être améliorée en utilisant des cohérences locales dites "plus fortes", reposant notamment cette fois-ci sur la recherche de supports dits *supports complets*.

La cohérence d'arc complète FAC* [Larrosa et Schiex, 2003, De Givry *et al.*, 2005]

Définition 49 (Cohérence d'arc complète FAC*) *Une valeur (x, a) est arc-cohérente complète (FAC) pour la contrainte c_{xy} s'il existe une valeur (y, b) telle que $c_{xy}(a, b) \oplus c_y(b) = 0$. La valeur (y, b) est alors*

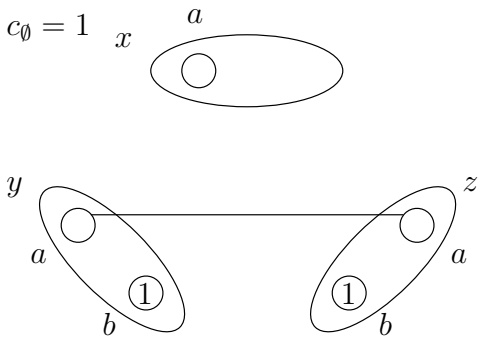


FIG. 2.14 – Réseau AC* équivalent au réseau de la figure 2.13(a) après projection(c_{yz},z,b,1).

appelée support complet de la valeur (x, a) dans c_{xy}. La variable x est arc-cohérente complète (FAC) si toutes les valeurs appartenant à $dom(x)$ sont arc-cohérentes complètes (FAC) dans toutes les contraintes binaires couvrant x. Un réseau est arc-cohérent complet (FAC) si toutes les variables appartenant à ce réseau sont arc-cohérentes complètes (FAC) et nœud-cohérentes (NC*).*

Jusqu'à présent nous avons vu que les cohérences locales souples NC* et AC*, afin de transférer le coût le plus grand possible sur la borne $c_\emptyset$, proposent de ramener les coûts des contraintes binaires vers les contraintes unaires en trouvant des supports simples pour chaque valeur des variables dans un objectif de centralisation des coûts. Au delà des supports simples, la recherche de supports complets va permettre de ramener sur une contrainte unaire des coûts supplémentaires qui vont être obtenus au delà des contraintes binaires, c'est à dire depuis les coûts unaires des variables voisines à cette variable pour cette contrainte, et bien évidemment de transférer encore plus de coûts sur la borne $c_\emptyset$.

La méthode *trouverSupportComplet* (algorithme 20) permet de trouver des supports complets dans le domaine de la variable y pour les valeurs de la variable x pour la contrainte c_{xy}. Elle s'appuie sur deux structures $P[a]$ et $E[b]$ qui vont stocker pour les valeurs (x, a) et (y, b) respectivement le coût projetable (pour projection) depuis c_{xy} sur $c_x(a)$ et le coût extensible (pour extension) depuis $c_y(b)$ sur c_{xy}. Des supports complets (y, b) sont cherchés uniquement pour les valeurs (x, a) pour lesquelles la somme $c_{xy}(a, Supp(x, a, y)) \oplus c_y(Supp(x, a, y))$ est différente de 0. Intuitivement, trouver un support complet (y, b) pour (x, a) dans c_{xy} va consister à déterminer les coûts transférables depuis les coûts $c_{xy}(a, Supp(x, a, y))$ et $c_y(Supp(x, a, y))$ vers $c_x(a)$ afin d'obtenir $c_{xy}(a, Supp(x, a, y))$ et $c_y(Supp(x, a, y))$ égaux à 0. Pour cela, une première étape (lignes 2 à 5) consiste à déterminer pour chaque valeur (x, a) la valeur (y, b) pour laquelle $c_{xy}(a, b) \oplus c_y(b)$ est minimal[12] et de stocker dans $P[a]$ le coût correspondant. Ensuite, une deuxième étape (lignes 6 à 8) consiste à déterminer pour chaque valeur (y, b) la valeur (x, a) pour laquelle $P[a] \ominus c_{xy}(a, b)$ est maximal[13] et de stocker dans $E[b]$ le coût correspondant. Pour finir, on étend pour chaque valeur (y, b) le coût correspondant $E[b]$ (ligne 10) et on projette sur chaque valeur (x, a) le coût correspondant $P[a]$ (ligne 12). Une fois ces différentes opérations terminées, on cherche une valeur support pour assurer la cohérence de nœud de la variable x.

Malheureusement, la cohérence d'arc complète FAC* ne représente pas une propriété exploitable en pratique car l'algorithme proposé pour l'établir n'atteint pas nécessairement un point fixe[14]. En effet, si tout problème WCSP peut être transformé en un réseau équivalent arc-cohérent AC* (prouvé dans

[12] $argmin_{b \in dom(y)}\{c_{xy}(a, b) \oplus c_y(b)\}$, ou argument minimum de la fonction $c_{xy}(a, b) \oplus c_y(b)$, représente la valeur b du domaine de la variable y pour laquelle la somme $c_{xy}(a, b) \oplus c_y(b)$ est minimale

[13] $argmax_{a \in dom(x)}\{P[a] \ominus c_{xy}(a, b)\}$, ou argument maximum de la fonction $P[a] \ominus c_{xy}(a, b)$, représente la valeur a du domaine de la variable x pour laquelle la somme $P[a] \ominus c_{xy}(a, b)$ est maximale

[14] Il est cependant intéressant de présenter l'algorithme trouverSupportComplet car il est utilisé pour établir certaines des cohérences abordées dans la suite de ce manuscrit

Algorithm 20: trouverSupportComplet (x : variable, y : variable) : Booléen

```
1  transfertEffectue ← faux
2  pour chaque valeur a ∈ dom(x) telle que c_xy(a, Supp(x,a,y)) ⊕ c_y(Supp(x,a,y)) > 0 faire
3      Supp(x,a,y) = argmin_{b∈dom(y)}{c_xy(a,b) ⊕ c_y(b)}
4      P[a] = c_xy(a, Supp(x,a,y)) ⊕ c_y(Supp(x,a,y))
5      si P[a] > 0 ∧ c_x(a) = 0 alors transfertEffectue ← vrai
6  pour chaque valeur b ∈ dom(y) faire
7      Supp(y,b,x) = argmax_{a∈dom(x)}{P[a] ⊖ c_xy(a,b)}
8      E[b] = P[Supp(y,b,x)] ⊖ c_xy(Supp(y,b,x), b)
9  pour chaque valeur b ∈ dom(y) faire
10     extension(c_xy,y,b,E[b])
11 pour chaque valeur a ∈ dom(x) faire
12     projection(c_xy,x,a,P[a])
13 trouverSupport(x)
14 Retourner transfertEffectue
```

[Larrosa et Schiex, 2004]), on ne peut pas toujours obtenir un réseau équivalent qui soit FAC* comme le montre l'exemple suivant.

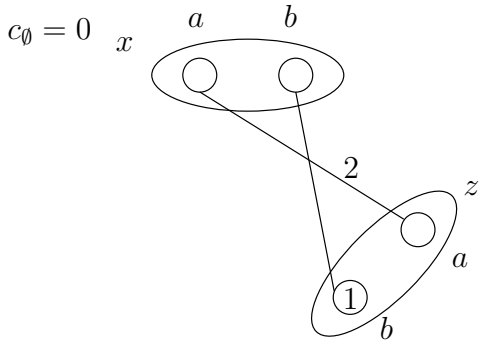
Exemple 16 *Le réseau WCSP (avec $k = 4$) illustré dans la figure 2.15(a) est AC* mais n'est pas FAC* parce que la valeur (x, a) ne possède pas de valeur support complète dans le domaine de la variable z pour la contrainte c_{xz}. Pour établir FAC* sur ce réseau, on cherche donc un support complet pour la valeur (x, a) selon la méthode trouverSupportComplet. L'état des structures obtenues durant l'appel à la méthode trouverSupportComplet est détaillé dans la figure 2.15(b). À partir des valeurs contenues dans ces structures, la figure 2.15(c) illustre les opérations de transfert de coûts réalisées : une extension d'un coût égal à 1 ($E[b]$) à partir de la contrainte $c_z(b)$ vers la contrainte c_{xz}, suivie de la projection d'un coût de 1 ($P[a]$) depuis la contrainte c_{xz} vers la contrainte $c_x(a)$. Nous obtenons le réseau illustré à la figure 2.15(d). Ce réseau équivalent n'est toujours pas FAC* parce que la valeur (z, b) ne possède pas de valeur support complète dans le domaine de la variable x pour la contrainte c_{xz}. Si on réalise cette fois-ci une recherche de support complet pour la valeur (z, b), on s'aperçoit que l'on va obtenir à nouveau le réseau dans l'état illustré à la figure 2.15(a). Il est clair que ce réseau n'a donc pas d'équivalent vérifiant la propriété FAC*.*

Pour pallier cet inconvénient, une propriété allégée de la cohérence d'arc complète FAC* a été proposée : il s'agit de la cohérence d'arc directionnelle DAC*.

La cohérence d'arc directionnelle DAC* [Larrosa et Schiex, 2003, De Givry *et al.*, 2005]

La propriété de cohérence d'arc directionnelle DAC* représente une version allégée de la cohérence d'arc complète FAC* dans le sens où les recherches de supports complets vont être effectuées sur les contraintes selon un ordre fixé des variables noté $<$.

Définition 50 (Cohérence d'arc directionnelle DAC*) *Une valeur (x, a) est arc-cohérente directionnelle (DAC) par rapport à un ordre $<$ sur les variables si elle possède un support complet dans toute contrainte c_{xy} telle que $x < y$. La variable x est arc-cohérente directionnelle (DAC) si toutes les valeurs*



(a) Réseau AC* mais non FAC*

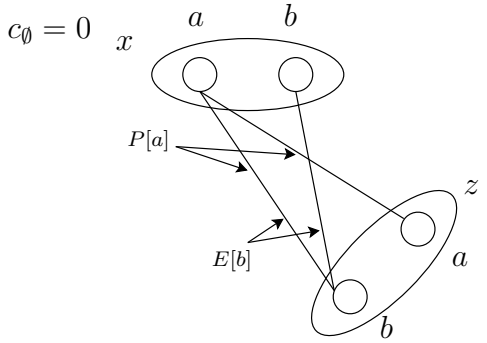
$Supp(x, a, z) = b$
$P[a] = c_{xz}(a, b) \oplus c_z(b) = 1$

$Supp(x, b, z) = a$
$P[b] = c_{xz}(b, a) \oplus c_z(a) = 0$

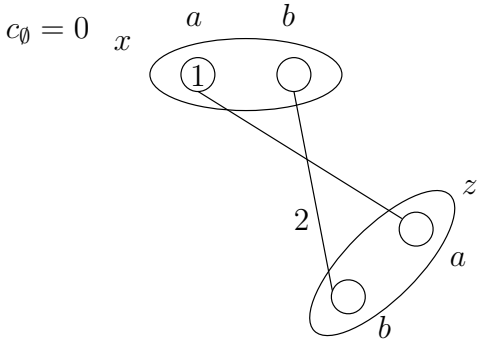
$Supp(z, a, x) = b$
$E[a] = P[b] \ominus c_{xz}(b, a) = 0$

$Supp(z, b, x) = a$
$E[b] = P[a] \ominus c_{xz}(a, b) = 1$

(b) État des structures après appel à trouverSupportComplet(x, z) (lignes 2 à 8)



(c) Opérations d'extension et de projection effectuées par trouverSupportComplet(x, z) (lignes 9 à 12)



(d) Réseau obtenu toujours non FAC*

FIG. 2.15 – Deux réseaux WCSP équivalents (avec $k = 4$).

appartenant à $dom(x)$ sont arc-cohérentes directionnelles (DAC). Un réseau est arc-cohérent directionnel (DAC) si toutes les variables appartenant à ce réseau sont arc-cohérentes directionnelles (DAC) et nœud-cohérentes (NC*).*

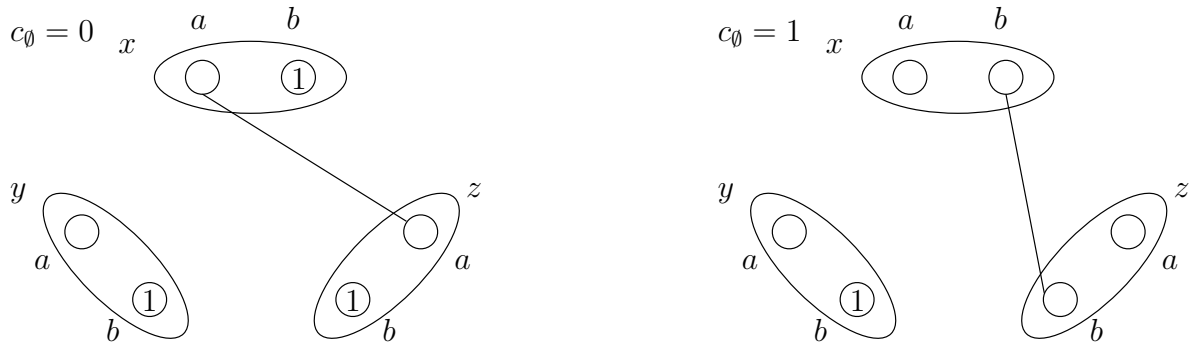
Nous étendons ici le terme de variable voisine. Nous appelons x une variable voisine inférieure (resp. variable voisine supérieure) d'une variable y si x et y sont voisines et si $x < y$ (resp. $x > y$) selon l'ordre des variables défini dans un réseau. Des supports complets vont devoir être trouvés pour les valeurs d'une variable uniquement pour les contraintes dans lesquelles cette variable correspond à la variable inférieure et elles devront être trouvées parmi les valeurs de sa variable voisine supérieure pour cette contrainte (unique car nous considérons ici le cas binaire).

À partir d'un réseau NC*, la méthode *DAC** (algorithme 21) parcourt une queue de priorité R, initialisée à $\mathcal{X}$, et qui va contenir les variables possédant une valeur pour laquelle le coût unaire a augmenté. En effet, cette variation de coût a peut être entraîné une perte de leur support complet pour les valeurs des variables qui sont leurs voisines inférieures au sein d'une contrainte. Dans un premier temps, une vérification de la cohérence de nœud est effectuée sur la variable sélectionnée car les variables présentes dans R font suite à des projections et les coûts unaires et le coût de la contrainte 0-aire ayant pu augmenter, certaines valeurs ne sont peut être plus nœud-cohérentes. Ces valeurs sont donc supprimées (ligne 5) et le cas échéant soit la variable est ajoutée dans une queue Q (ligne 6) qui a la même utilité que dans la méthode *AC*2001* (algorithme 18), soit le processus est arrêté car un domaine vide est apparu. Ensuite, des supports complets sont cherchés via un appel (ligne 8) à la méthode *trouverSupportComplet* (algorithme 20) pour les valeurs de toutes les variables voisines inférieures à cette variable dans les contraintes correspondantes. Finalement, un test de nœud-cohérence (ligne 11) est à nouveau effectué sur toutes les variables du réseau car les recherches de supports complets ont pu faire grimper la borne inférieure et ainsi fait apparaître des valeurs nœud-incohérentes pour des variables qui sont alors ajoutées le cas échéant dans Q.

Algorithm 21: DAC* ($P = (\mathcal{X}, \mathcal{C}, k)$: WCSP) : Booléen

```
1  si NC*(P) alors
2      Q ← X, R ← X
3      tant que R ≠ ∅ faire
4          Choisir et éliminer y de R
5          si filtrerDomaine(y) alors
6              si dom(y) = ∅ alors Retourner faux
7              Q ← Q ∪ {y}
8          pour chaque contrainte c_xy telle que x < y faire
9              si trouverSupportComplet(x,y) alors
10                 R ← R ∪ {x}
11     pour chaque x ∈ X faire
12         si filtrerDomaine(x) alors
13             si dom(x) = ∅ alors Retourner faux
14             Q ← Q ∪ {x}
15     Retourner vrai
16 sinon
17     Retourner faux
```

Exemple 17 *Le réseau WCSP (avec $k = 4$) illustré dans la figure 2.16(a), constitué des contraintes c_{xz} et c_{yz} et dans lequel les variables sont ordonnées par $x < y < z$, est AC* mais n'est pas DAC* parce que la valeur (x, a) ne possède pas de valeur support complète dans le domaine de la variable z pour la contrainte c_{xz}. Pour établir DAC* sur ce réseau, on cherche donc un support complet pour la valeur (x, a) selon la méthode trouverSupportComplet. Cette recherche amène à effectuer une extension d'un coût égal à 1 à partir de la contrainte $c_z(b)$ vers la contrainte c_{xz}, suivie de la projection d'un coût de 1 depuis la contrainte c_{xz} vers la contrainte $c_x(a)$. Nous obtenons le réseau illustré à la figure 2.16(b). Ce réseau équivalent n'est toujours pas FAC*, par contre il est bien DAC* car toutes les valeurs d'une variable possèdent bien un support complet pour chaque contrainte dans laquelle cette variable est inférieure.*



(a) Réseau AC* mais non DAC* (b) Réseau DAC* ($x < y < z$)

FIG. 2.16 – Deux réseaux WCSP équivalents (avec $k = 4$).

L'algorithme *DAC** (algorithme 21) pour établir la cohérence d'arc directionnelle DAC* a une complexité en temps $O(ed^2)$ et une complexité en espace $O(ed)$ où e correspond au nombre de contraintes et d la taille maximale parmi les domaines des variables.

La cohérence d'arc directionnelle complète FDAC* [Larrosa et Schiex, 2003, De Givry *et al.*, 2005]

Définition 51 (Cohérence d'arc directionnelle complète FDAC*) *Une valeur (x, a) est arc-cohérente directionnelle complète (FDAC) si elle est arc-cohérente directionnelle (DAC) et si en plus elle possède un support (simple) dans toute contrainte c_{xy} telle que $y < x$. La variable x est arc-cohérente directionnelle complète (FDAC) si toutes les valeurs appartenant à $dom(x)$ sont arc-cohérentes directionnelles complètes (FDAC). Un réseau est arc-cohérent directionnel complet (FDAC*) si toutes les variables appartenant à ce réseau sont arc-cohérentes directionnelles complètes (FDAC) et nœud-cohérentes (NC*).*

L'idée de cette propriété est de profiter à la fois des intérêts de la cohérence d'arc AC* et des intérêts de la cohérence directionnelle DAC*. En ce sens, toutes les valeurs appartenant au domaine d'une variable doivent posséder un support complet dans toutes les contraintes où cette variable est inférieure (support correspondant à une valeur du domaine d'une variable voisine qui est supérieure) et doivent posséder un support simple dans toutes les contraintes où cette variable est supérieure (support correspondant à une valeur du domaine d'une variable voisine qui est inférieure). En d'autres termes, ces valeurs doivent donc être supportées complètement dans un sens et supportées simplement dans l'autre. L'objectif de ce "double support" est bien évidemment de centraliser sur le coût unaire d'une valeur les coûts pouvant être obtenus depuis les deux directions.

En fait, comme cela apparaît dans l'algorithme 22, la méthode *FDAC** consiste à établir à la fois la cohérence d'arc AC* (appel de la méthode *AC*2001*, algorithme 18) et la cohérence d'arc directionnelle DAC* (appel de la méthode *DAC**, algorithme 21) sur le réseau traité. Sur la même idée que les algorithmes présentés précédemment, deux queues de priorité sont exploitées. Pour rappel, la queue Q contient les variables dont le domaine a été modifié et la queue R contient les variables pour lesquelles le coût unaire d'au moins une valeur a augmenté. Dès lors qu'une valeur est supprimée du domaine d'une variable lorsque les cohérences AC* ou DAC* sont établies, comme nous l'avons vu elle est ajoutée dans la queue Q car potentiellement une valeur vient d'être privée de son support simple et donc la cohérence AC* doit être vérifiée. Pour établir la cohérence AC*, c'est donc la queue Q qui est parcourue. Par contre, lors de la recherche de supports simples pour établir AC* ou complets pour établir DAC*, les coûts unaires des valeurs du domaine d'une variable peuvent augmenter. Ces variables sont donc ajoutées dans la queue R car potentiellement une valeur appartenant à une variable voisine inférieure de cette variable vient d'être privée de son support complet et donc la cohérence DAC* doit être vérifiée. Pour établir la cohérence DAC*, c'est donc la queue R qui est parcourue. La cohérence d'arc directionnelle complète FDAC* est donc établie une fois que les queues Q et R sont simultanément vides et qu'aucun domaine vide n'est apparu durant le processus.

Algorithm 22: FDAC* ($P = (\mathcal{X}, \mathcal{C}, k)$: WCSP)

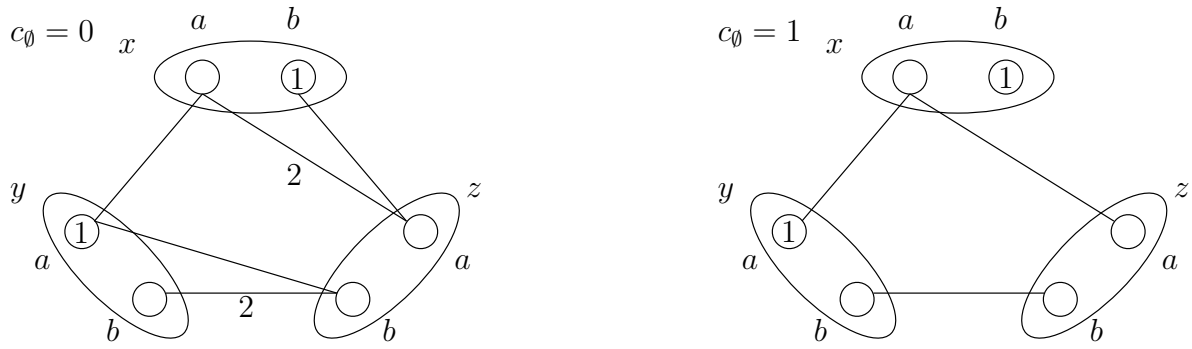
```
1 Q ← X, R ← X
2 tant que Q ≠ ∅ ∨ R ≠ ∅ faire
3 |   AC*(P)
4 |   DAC*(P)
```

Exemple 18 *Nous considérons l'existence d'une contrainte supplémentaire c_{xy} dans le réseau WCSP (avec $k = 4$) illustré dans la figure 2.17(a) basé sur l'ordre des variables $x < y < z$. Ce réseau est DAC* mais n'est pas FDAC* parce que les valeurs (z, a) et (z, b) ne sont pas AC* : la valeur (z, a) (resp. la valeur (z, b)) ne possède pas de valeur support simple dans le domaine de la variable x (resp. y) pour la contrainte c_{xz} avec $x < z$ (resp. la contrainte c_{yz} avec $y < z$). Pour établir AC* sur ce réseau et plus précisément sur ces deux contraintes, une projection d'un coût égal à 1 est tout d'abord effectuée à partir de la contrainte c_{xz} sur $c_z(a)$ afin d'obtenir une valeur support simple dans le domaine de la variable x pour la valeur (z, a). Ensuite, une projection d'un coût égal à 1 est effectuée à partir de la contrainte c_{yz} sur $c_z(b)$ afin d'obtenir une valeur support simple dans le domaine de la variable y pour la valeur (z, b). Ensuite, pour respecter la cohérence de nœud NC*, une projection unaire d'un coût égal à 1 est effectuée depuis c_z sur $c_\emptyset$. Le réseau obtenu à la figure 2.17(b) est FDAC*.*

L'algorithme *FDAC** (algorithme 22) pour établir la cohérence d'arc directionnelle complète FDAC* a une complexité en temps $O(end^3)$ et une complexité en espace $O(ed)$ où e correspond au nombre de contraintes, n le nombre de variables et d la taille maximale parmi les domaines des variables.

La cohérence d'arc directionnelle existentielle EDAC* [De Givry *et al.*, 2005]

Définition 52 (Cohérence d'arc existentielle EAC*) *Une variable x est arc-cohérente existentielle si et seulement si il existe au moins une valeur (x, a) telle que $c_x(a) = 0$ et possédant un support complet dans toutes les contraintes binaires couvrant x. Un réseau est arc-cohérent existentiel (EAC*) si toutes les variables appartenant à ce réseau sont arc-cohérentes existentielles et nœud-cohérentes (NC*).*



(a) Réseau DAC* mais non FDAC*

(b) Réseau FDAC*

FIG. 2.17 – Deux réseaux WCSP équivalents (avec $k = 4$).

La cohérence d'arc existentielle EAC* impose qu'il existe dans le domaine de chaque variable une valeur spéciale appelée *valeur complètement supportée*, c'est à dire une valeur complètement supportée dans toutes les contraintes, quelle que soit la direction. Le constat à l'origine de cette propriété est que si une telle valeur n'existe pas dans le domaine d'une variable x, cela veut dire qu'il va être possible de projeter des coûts sur toute valeur a telle que $c_x(a) = 0$ via la recherche d'un support complet afin de pouvoir ensuite effectuer des projections unaires pour établir NC* et améliorer ainsi la valeur de $c_\emptyset$.

Définition 53 (Cohérence d'arc existentielle directionnelle complète EDAC*) *Un réseau est arc-cohérent existentiel directionnel complet (EDAC*) s'il est arc-cohérent directionnel complet (FDAC*) et arc-cohérent existentiel (EAC*).*

La propriété EDAC* complète la propriété FDAC* dans le sens où toutes les valeurs de chaque variable du réseau doivent être complètement supportées dans une direction et simplement supportées dans l'autre direction (propriété FDAC*), mais aussi que chaque variable doit contenir une valeur complètement supportée dans les deux sens (propriété EAC*). Comme nous le verrons, établir la cohérence EDAC* sur un réseau n'offre pas la garantie d'obtenir une valeur optimale $c_\emptyset$. En effet, cela dépend de la séquence de transformations suivie et d'autres cohérences locales souples, décrites dans la section 2.3.3, ont été proposées dans ce sens. Cependant, une fois la cohérence EDAC* établie, on sait qu'un point fixe est atteint et que plus aucun transfert de coût n'est possible pour faire varier $c_\emptyset$ et donc l'améliorer. En ce sens, la cohérence EDAC* correspond réellement à une version aboutie des différentes cohérences locales souples présentées jusqu'ici dans ce manuscrit.

La méthode *EDAC** (algorithme 23) permet d'établir la cohérence d'arc existentielle directionnelle complète sur un réseau. Elle est basée sur l'utilisation de trois queues de priorité de propagation Q, R et K contenant au début l'ensemble des variables du problème. Ces différentes queues sont mises à jour durant l'exécution suite aux évènements nécessitant de vérifier le maintien des cohérences locales. Les queues Q et R proposent les mêmes caractéristiques que lors de leur utilisation dans les cohérences présentées précédemment. La queue K va contenir les variables pour lesquelles au moins une valeur du domaine de l'une de ses variables voisines inférieures a vu son coût unaire augmenter et lui a fait potentiellement perdre le support complet pour sa valeur complètement supportée. À noter que la queue S utilisée par les auteurs sert à construire efficacement la queue K, nous ne nous attarderons pas plus longtemps sur cette mécanique, de plus amples informations pouvant être obtenues dans [De Givry *et al.*, 2005]. La boucle principale de cet algorithme s'exécute tant que les queues Q, R et S ne sont pas simultanément vides. Les différentes boucles débutant aux lignes 5, 11 et 17 correspondent respectivement à établir les cohérences EAC*, DAC* et AC* (pour rappel, DAC* associé à AC* repré-

sente FDAC*) et la boucle débutant à la ligne 23 permet d'établir la cohérence de nœud avec mise à jour de la queue Q le cas échéant. Pour établir EAC*, la recherche de support complet pour obtenir une valeur complètement supportée est effectuée à la ligne 7 par la méthode *trouverSupportExistentiel* (algorithme 24). Des supports complets vont être cherchés pour les variables uniquement dans les contraintes les liant avec des variables voisines inférieures (ligne 4), la recherche dans l'autre direction étant déjà effectuée pour établir la propriété *FDAC** (ligne 13 de la méthode *EDAC**). Cette recherche peut amener à augmenter le coût unaire de valeurs dans les domaines de ces variables et de ce fait la propriété DAC* doit être vérifiée pour les variables voisines inférieures (ajout dans la queue R, ligne 8 de la méthode *EDAC**), ainsi que la propriété EAC* pour les variables voisines supérieures (ajout dans la queue K, ligne 10 de la méthode *EDAC**).

Algorithm 23: EDAC* ($P = (\mathcal{X}, \mathcal{C}, k)$: WCSP)

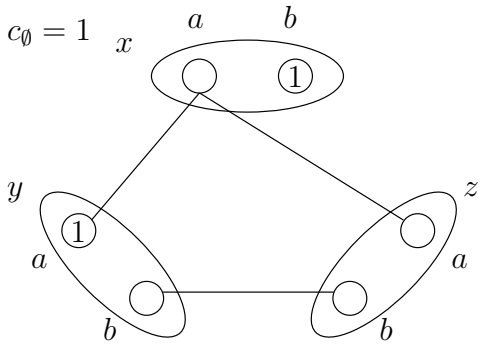
```
1  Q ← X, R ← X, S ← X
2  tant que Q ≠ ∅ ∨ R ≠ ∅ ∨ S ≠ ∅ faire
3      K = {y | x ∈ S, y > x, c_xy ∈ C} ∪ S
4      S = ∅
5      tant que K ≠ ∅ faire
6          Choisir et éliminer x minimal de R
7          si trouverSupportExistentiel(x) alors
8              R ← R ∪ {x}
9              pour chaque contrainte c_xy ∈ C telle que y > x faire
10                 K ← K ∪ {y}
11     tant que R ≠ ∅ faire
12         Choisir et éliminer y maximal de R
13         pour chaque contrainte c_xy ∈ C telle que x < y faire
14             si trouverSupportComplet(x,y) alors
15                 R ← R ∪ {x}
16                 S ← S ∪ {x}
17     tant que Q ≠ ∅ faire
18         Choisir et éliminer y maximal de Q
19         pour chaque contrainte c_xy ∈ C telle que x > y faire
20             si trouverSupportSimple(x,y) alors
21                 R ← R ∪ {x}
22                 S ← S ∪ {x}
23     pour chaque variable x ∈ X faire
24         si filtrerDomaine(x) alors
25             Q ← Q ∪ {x}
```

Exemple 19 *Le réseau WCSP (avec $k = 4$) illustré dans la figure 2.18(a), dans lequel les variables sont ordonnées par $x < y < z$, est FDAC* mais n'est pas EDAC*. En effet, la variable z ne possède pas de valeur complètement supportée. Certes les valeurs (z, a) et (z, b) possèdent des supports simples dans les contraintes c_{xz} et c_{yz} (respect de FDAC*) et $c_z(a) = c_z(b) = 0$, mais aucune de ces deux valeurs ne possède de support complet à la fois dans les contraintes c_{xz} et c_{yz} (non respect de EAC*).*

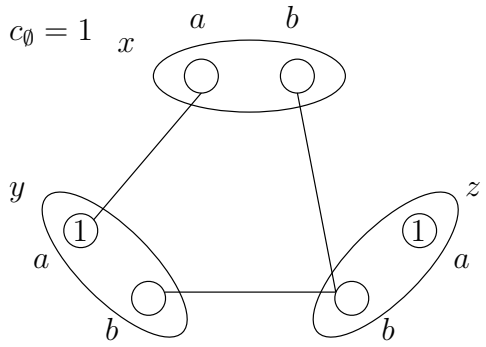
Algorithm 24: trouverSupportExistentiel (x : variable) : Booléen

```
1 transfertEffectue ← faux
2 α = min_{a∈dom(x)}{c_x(a) ⊕_{c_xy∈C,y<x} min_{b∈dom(y)}{c_xy(a,b) ⊕ c_y(b)}}
3 si α > 0 alors
4     pour chaque contrainte c_xy telle que y < x faire
5         transfertEffectue = transfertEffectue ∨ trouverSupportComplet (x,y)
6 Retourner transfertEffectue
```

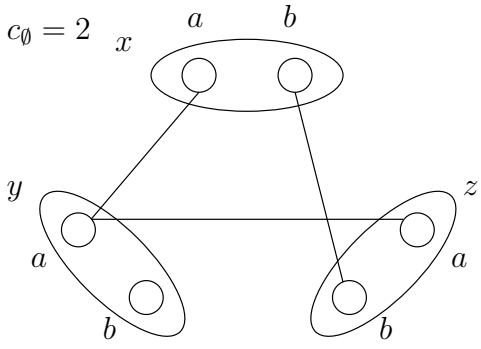
Pour établir EDAC sur ce réseau, une recherche de support complet est réalisée dans un premier temps pour la valeur (z, a) dans c_{xz} : une extension d'un coût égal à 1 est réalisée depuis la contrainte $c_x(b)$ vers la contrainte c_{xz}, puis une projection d'un coût égal à 1 est réalisée depuis la contrainte c_{xz} vers la contrainte $c_z(a)$. Le réseau obtenu et illustré dans la figure 2.18(b) n'est toujours pas EDAC* : certes la valeur $c_z(a)$ est complètement supportée, cependant $c_z(a) \neq 0$ (non respect de EAC*). Une recherche de support complet est alors réalisée pour la valeur (z, b) dans c_{yz} : une extension d'un coût égal à 1 est effectuée depuis la contrainte $c_y(a)$ vers la contrainte c_{yz}, puis une projection d'un coût égal à 1 est réalisée depuis la contrainte c_{yz} vers la contrainte $c_z(b)$. Finalement, pour établir la cohérence de nœud NC*, une projection unaire d'un coût égal à 1 est effectuée depuis c_z et la valeur de $c_\emptyset$ est incrémentée de 1. Le réseau obtenu à la figure 2.18(c) est EDAC* : la variable z possède au moins une valeur de coût unaire égal à 0 et complètement supportée dans les contraintes c_{xz} et c_{yz}.*



(a) Réseau FDAC* mais non EDAC*



(b) Réseau FDAC* mais non EDAC*



(c) Réseau EDAC*

FIG. 2.18 – Trois réseaux WCSP équivalents (avec $k = 4$).

L'algorithme *EDAC** (algorithme 23) pour établir la cohérence d'arc existentielle complète EDAC* a une complexité en temps $O(ed^2max\{nd, k\})$ et une complexité en espace $O(ed)$ où e correspond au nombre de contraintes, n le nombre de variables, d la taille maximale parmi les domaines des variables et k le coût interdit pour le réseau.

2.3.3 À la recherche de la meilleure cohérence locale souple

Deux cohérences locales souples ont été proposées dans le but d'améliorer encore la borne inférieure $c_\emptyset$ jusqu'à obtenir une fermeture d'arc-cohérence au plus proche de l'optimale, voire une fermeture qui est optimale : il s'agit de la cohérence d'arc souple optimale OSAC et de la cohérence d'arc virtuelle VAC que nous présentons dans les deux sections suivantes. Dans le cadre de nos contributions, aucune de nos méthodes et algorithmes n'a été comparée avec ces deux cohérences locales souples, et cela pour deux raisons : leur mise en œuvre s'avère lourde et difficile et, sachant que l'utilisation de ces cohérences durant la recherche est déjà très coûteuse dans le cadre de réseaux binaires (comme expliqué dans les sections suivantes), on peut légitimement présumer de leur inefficacité face à des problèmes avec contraintes de grande arité qui nous ont intéressés dans la plus grande partie de notre travail. Cependant, il est intéressant de les aborder dans ce manuscrit. D'une part, la cohérence d'arc souple optimale OSAC exploite de manière intéressante une technique issue de la recherche opérationnelle et surclasse, en terme de qualité de borne minorante, les différentes cohérences locales souples présentées ci-dessus. D'autre part, la cohérence d'arc virtuelle VAC représente un premier pas dans l'intégration de techniques CSP pour la résolution de problèmes WCSP et qu'une partie de la méthodologie de VAC a servi de point de départ à notre seconde contribution.

La cohérence d'arc souple optimale OSAC [Cooper *et al.*, 2007, Cooper *et al.*, 2010]

La cohérence d'arc souple optimale OSAC a été proposée afin de calculer des fermetures d'arc-cohérence optimales. Comme énoncé dans la section 2.3.2, trouver une fermeture d'arc-cohérence qui permet d'optimiser la valeur de la borne $c_\emptyset$ est un problème `NP-Difficile` si l'on se restreint à des coûts entiers. Pour pallier cette difficulté, le principe pour établir la cohérence d'arc souple optimale OSAC en temps polynomial est d'une part de relâcher la condition sur les coûts traités et projetés en autorisant des coûts rationnels : la structure de valuation considérée ici est alors $(\mathbb{Q} \cup \{+\infty\}, +, <)$ et non plus $(\mathbb{N} \cup \{+\infty\}, +, <)$. D'autre part, des transferts de coûts simultanés entre les différentes contraintes souples du réseau sont autorisés. Ces deux relâchements permettent d'établir en temps polynomial la cohérence OSAC qui représente une fermeture d'arc-cohérence optimale.

Pour établir la cohérence d'arc souple optimale OSAC, l'idée consiste à considérer le réseau WCSP à traiter comme un programme linéaire en variables continues, c'est à dire un problème dans lequel il s'agit d'optimiser (maximiser ou minimiser) une fonction linéaire de variables continues devant satisfaire un ensemble de contraintes linéaires d'égalités et/ou d'inégalités. Ce qui nous intéresse dans le cadre de OSAC, c'est de trouver l'ensemble des EPT qui, appliquées simultanément, vont permettre de maximiser l'augmentation de la contrainte $c_\emptyset$. L'un des algorithmes les plus connus pour résoudre ce type de problème est l'algorithme du *simplexe* [Dantzig, 1982]. Cependant, bien qu'il soit très souvent efficace en pratique, l'algorithme du *simplexe* n'est pas polynomial (exponentiel dans le pire des cas). L'algorithme de *Karmakar* [Karmarkar, 1984], qui est une "méthode de points intérieurs", est polynomial et compétitif en pratique par rapport à l'algorithme du *simplexe*. Ainsi, l'algorithme de *Karmakar* peut être exploité pour établir la cohérence d'arc souple optimale OSAC et ainsi obtenir une borne inférieure optimale sur les réels. Nous notons $p_{c_x}^{c_\emptyset}$ le coût transféré de la contrainte c_x vers la contrainte $c_\emptyset$ (par projection unaire) et $p_{c_S}^{c_x(a)}$ qui est soit le coût transféré de la contrainte c_S vers la contrainte $c_x(a)$ s'il est positif (par projection), soit le coût transféré de la contrainte $c_x(a)$ vers la contrainte c_S s'il est négatif (par extension).

Ainsi, la fonction d'objectif à maximiser (2.10) se traduit comme la somme des coûts transférés depuis toutes les contraintes unaires portant sur les variables du réseau vers la contrainte $c_\emptyset$. L'équation (2.11) impose que les coûts unaires finaux doivent rester positifs. En d'autres termes, le coût total transféré sur $c_\emptyset$ depuis un coût unaire ne doit pas être supérieur à la valeur initiale de ce coût unaire additionnée à la somme des coûts transférés sur ce coût unaire depuis les contraintes binaires. L'équation (2.12) impose que le total des coûts transférés depuis un tuple vers les coûts unaires des valeurs qu'il contient ne doit pas dépasser le coût de ce tuple lui-même. Ces deux conditions ont pour but d'éviter les coûts négatifs une fois la cohérence OSAC établie (ils sont cependant autorisés durant le traitement, mais pendant une durée nulle car les transferts sont simultanés) et d'obtenir ainsi un réseau WCSP valide et équivalent.

$$max \bigoplus_{x \in \mathcal{X}} p_{c_x}^{c_\emptyset} \tag{2.10}$$

$$\forall x \in \mathcal{X}, \forall a \in dom(x), c_x(a) \ominus p_{c_x}^{c_\emptyset} \oplus \bigoplus_{c_S \in \mathcal{C}, x \in S} p_{c_S}^{c_x(a)} \geq 0 \tag{2.11}$$

$$\forall c_S \in \mathcal{C}, |S| > 1, \forall \tau \in l(S), c_S(\tau) \ominus \bigoplus_{x \in S} p_{c_S}^{c_x(\tau[x])} \geq 0 \tag{2.12}$$

Il s'agit donc de résoudre le problème linéaire correspondant aux équations (2.10), (2.11) et (2.12) qui consiste à maximiser la fonction linéaire (2.10) tout en respectant les différentes contraintes linéaires d'inégalités (2.11) et (2.12). L'algorithme de *Karmakar* peut alors être utilisé pour résoudre ce problème linéaire : les valeurs calculées pour les différentes inconnues $p_{c_x}^{c_\emptyset}$ et $p_{c_S}^{c_x(a)}$ représentent ainsi les différentes projections à effectuer (une valeur négative pour $p_{c_S}^{c_x(a)}$ représentant une extension) et constituent la séquence d'EPT permettant de parvenir à une fermeture d'arc-cohérence optimale. Malheureusement trop coûteuse si utilisée durant la recherche, l'utilisation de la cohérence d'arc souple optimale OSAC est surtout efficace en pré-traitement pour des problèmes grands ou/et difficiles, où la borne inférieure obtenue peut être trois fois meilleure que la meilleure des approches standard, en l'occurrence la cohérence EDAC et plus généralement les approches effectuant uniquement des EPT classiques.

La cohérence d'arc virtuelle VAC [Cooper *et al.*, 2008, Cooper *et al.*, 2010]

La cohérence d'arc virtuelle VAC n'a pas la prétention de fournir une fermeture d'arc-cohérence optimale comme c'est le cas avec la cohérence OSAC. L'objectif est plutôt de parvenir à une fermeture d'arc-cohérence (plus ou moins proche de l'optimalité, voire l'optimalité elle-même) qui offre la garantie d'augmenter la valeur de $c_\emptyset$ quand c'est possible. Plus précisément, établir la cohérence d'arc virtuelle va permettre de déterminer une séquence d'EPT de coûts rationnels (comme OSAC) qui, appliquées séquentiellement, vont amener à cette fermeture d'arc-cohérence et augmenter $c_\emptyset$. La cohérence d'arc virtuelle s'appuie sur la transformation d'un WCN P en un CN, noté $Bool(P)$, défini par :

Définition 54 *Étant donné un WCN $P = (\mathcal{X}, \mathcal{W}, k)$, $Bool(P)$ est défini comme étant le CN $(\mathcal{X}, \mathcal{C})$ dans lequel $\exists c_S \in \mathcal{C}$ si et seulement si $\exists w_S \in \mathcal{W}$ avec $S \neq \emptyset$, et $\forall \tau \in l(S)$, τ est autorisé par c_S si et seulement si $w_S(\tau) = 0$.*

Autrement dit, $Bool(P)$ est un CN dans lequel un tuple est autorisé si son coût dans le WCN P est égal à 0 et qui est interdit sinon. Si $Bool(P)$ est arc-cohérent, alors P est arc-cohérent virtuel VAC. Le réseau $Bool(P)$ étant arc-cohérent, cela veut dire aussi que toute solution dans P a un coût égal à la valeur de la contrainte $c_\emptyset$ de P car tous les coûts présents dans $Bool(P)$ sont nuls. Par contre, si $Bool(P)$ n'est pas arc-cohérent (et donc P n'est pas arc-cohérent virtuel VAC), cela veut dire que toutes les solutions dans P ont un coût strictement supérieur à $c_\emptyset$ (les tuples de coût nul dans $Bool(P)$ ne sont

pas suffisants pour trouver une solution) et que donc il est possible d'augmenter $c_\emptyset$ d'une valeur que l'on peut noter λ. Établir la cohérence d'arc virtuelle VAC consiste donc à trouver la séquence d'EPT amenant à cette augmentation de λ de la contrainte $c_\emptyset$. Trois phases sont nécessaires :

- La première phase consiste donc à établir l'arc-cohérence AC sur le réseau $Bool(P)$: si le réseau $Bool(P)$ n'est pas arc-cohérent (apparition d'un domaine vide représenté par un coût non nul pour chacune des valeurs du domaine), c'est donc que le réseau P n'est pas arc-cohérent virtuel VAC et que donc $c_\emptyset$ peut être augmentée. À noter que durant cette phase, pour chaque valeur supprimée, l'identité de la variable responsable de la perte de support pour cette valeur est stockée pour la phase suivante. Si un domaine vide est apparu, c'est que toutes les valeurs de ce domaine sont interdites et qu'elles représentent alors un certain coût : le coût λ que l'on espère projeter sur $c_\emptyset$ peut donc être obtenu depuis ces valeurs.
- La deuxième phase consiste justement à déterminer dans le réseau $Bool(P)$ la provenance des coûts qui ont permis d'alimenter les coûts non nuls de ces valeurs. Guidée par les opérations ayant amené l'apparition et la suppression de valeurs sans support et stockées lors de la première phase, la recherche de ces coûts débute depuis ces valeurs et remonte plus loin dans les contraintes, binaires (récupération de coûts par projection) et unaires (récupération de coûts par extension), jusqu'à ce que des coûts non nuls, et donc pouvant fournir une valeur λ, soient rencontrés. Le nombre de demandes d'un coût λ à partir de chacun des coûts non nuls identifiés et le coût maximal disponible depuis chaque coût non nul est alors évalué et correspond au coût qui sera projeté sur $c_\emptyset$.
- La troisième phase consiste, une fois l'origine des coûts requis déterminée, à effectuer dans le réseau pondéré initial P les transferts de coûts d'une valeur égale à λ dans le sens inverse de la recherche des coûts de la phase 2, c'est à dire des contraintes les plus lointaines jusqu'aux contraintes unaires des valeurs du domaine devenu vide lors de la première phase.

Une fois la troisième phase achevée et les transferts de coûts effectués dans le réseau pondéré initial P, si le nouveau réseau durci $Bool(P)$ associé à P n'est pas arc-cohérent, cela veut dire que le réseau P n'est toujours pas VAC et qu'une nouvelle itération de VAC est nécessaire. D'ailleurs, on parle généralement de VAC$_\epsilon$ quand on s'intéresse à l'implémentation de la cohérence VAC. En effet, afin de borner le nombre d'itérations pour établir VAC, le processus (qui peut boucler indéfiniment avec des λ de plus en plus petits) est stoppé si l'amélioration de la valeur $c_\emptyset$ n'atteint pas un seuil noté ϵ. Le qualificatif *virtuelle* provient du fait que lorsque l'arc-cohérence AC est établie sur $Bool(P)$, des coûts virtuels sont simultanément transférés dans le réseau : les valeurs deviennent interdites et se voient donc attribuer un coût non nul qui n'est pas *réellement* projeté ou étendu depuis une autre contrainte. Cette propriété de virtualité est aussi valable pour qualifier certaines opérations simultanées effectuées durant la résolution des problèmes linéaires dans le cadre OSAC. À noter que la cohérence d'arc virtuelle VAC est plus légère que la cohérence d'arc souple optimale OSAC, cette dernière nécessitant pour être établie la résolution (généralement fastidieuse) d'un problème linéaire. De ce fait, certes la valeur $c_\emptyset$ obtenue n'est pas forcément aussi élevée mais le maintien de la cohérence VAC est envisageable durant la phase de recherche et permet une amélioration de l'efficacité pour résoudre des problèmes difficiles. Il est intéressant de savoir aussi que la cohérence VAC permet de résoudre directement et efficacement certains problèmes composés de fonctions de coût sous-modulaires [Cohen *et al.*, 2006]. À noter également qu'une approche plus efficace pour établir la cohérence d'arc virtuelle VAC a été proposée tout récemment dans [Nguyen *et al.*, 2013]. Il s'agit d'une variante consistant à exploiter pour les différents problèmes intermédiaires $Bool(P)$ traités les propriétés d'incrémentalité utilisées par les algorithmes de résolution dans le cadre des CSP dynamiques [Bessiere, 1991].

Pour conclure, il est intéressant de noter que l'idée ici a été d'intégrer des techniques issues du cadre CSP, à savoir la conversion d'un réseau WCSP en réseau CSP et l'utilisation de l'arc-cohérence AC, pour aider à la résolution de problèmes WCSP. L'intérêt évident est que l'efficacité de ces différentes

techniques a été prouvé depuis bon nombre d'années et que, contrairement aux arc-cohérences souples, l'arc-cohérence (dure) est confluante et donc bien moins complexe à mettre en œuvre. Comme nous venons de le voir, cette première intégration de techniques CSP dans le cadre WCSP est donc bénéfique et représente une motivation supplémentaire pour nos travaux, basés sur l'intégration de techniques CSP pour résoudre les WCSP, que nous avons réalisé dans nos deux contributions abordées dans les chapitres 3 et 4 de la partie II.

2.3.4 Comparaison et récapitulatif des cohérences locales souples

Afin de conclure sur les différentes cohérences locales souples présentées, nous proposons tout d'abord un comparatif de ces cohérences basé sur deux relations introduites dans [Dehani, 2014] : il s'agit des relations de *$c_\emptyset$-supériorité* et d'*implication* entre cohérences.

Définition 55 ($c_\emptyset$-supériorité) *Soient deux cohérences ϕ et ψ. On dit que ϕ est $c_\emptyset$-supérieure à ψ si et seulement si il n'existe pas un WCN P tel que $c_\emptyset(\psi \circ \phi(P)) \geq c_\emptyset(\phi(P))$, où $c_\emptyset(\theta(P))$ représente la borne inférieure maximale que l'on peut obtenir en établissant la propriété θ sur le WCN P. Si ϕ n'est pas $c_\emptyset$-supérieure à ψ et vice versa, alors les deux cohérences sont dites incomparables : on dit alors que ϕ est $c_\emptyset$-incomparable à ψ.*

En d'autres termes, une cohérence ϕ est $c_\emptyset$-supérieure à une cohérence ψ si et seulement si appliquer ψ après ϕ ne permet jamais d'augmenter la borne inférieure $c_\emptyset$.

Définition 56 (Implication de cohérences) *La cohérence ϕ implique la cohérence ψ si tout WCN P qui vérifie ϕ vérifie également ψ. Si ϕ implique ψ alors ϕ est $c_\emptyset$-supérieure à ψ*

La figure 2.19 illustre les relations de *$c_\emptyset$-supériorité* et d'*implication* existantes entre les cohérences présentées précédemment.

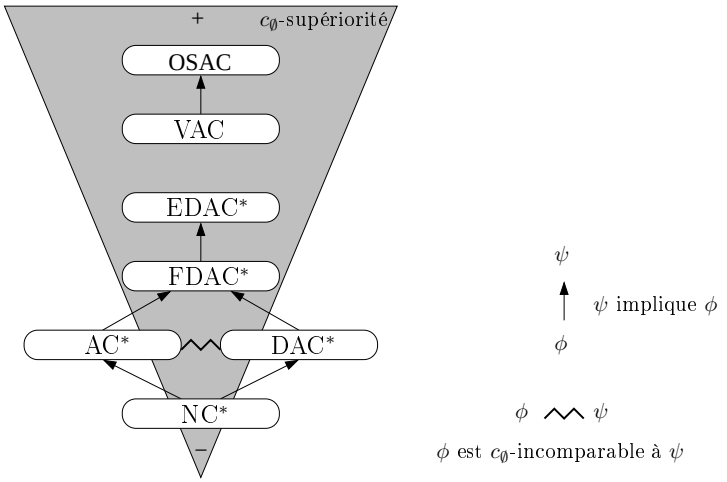


FIG. 2.19 – Relations de $c_\emptyset$-supériorité et d'implication entre les cohérences.

Sur cette figure, une cohérence ϕ implique une cohérence ψ si ϕ et ψ sont reliées par une flèche partant de ψ et menant à ϕ. D'après les définitions des cohérences AC* et DAC*, pour vérifier AC* ou DAC*, un WCN doit également vérifier NC* : les cohérences AC* et DAC* impliquent donc la cohérence NC* (définition 56) et sont alors $c_\emptyset$-supérieures à NC* (définition 55). Dans [Dehani, 2014], il a été montré d'une part que la cohérence AC* n'implique pas DAC* (un WCN vérifiant AC* ne vérifie

pas forcément DAC*) et d'autre part que la cohérence DAC* n'implique pas AC* (un WCN vérifiant DAC* ne vérifie pas forcément AC*). De plus, il a été constaté que la cohérence AC* n'est pas $c_\emptyset$-supérieure à DAC* (établir la cohérence DAC* sur un WCN vérifiant AC* peut permettre d'augmenter la valeur de $c_\emptyset$) et vice versa. Les cohérences AC* et DAC* sont donc $c_\emptyset$-incomparables (reliées par une ligne brisée sur la figure) et aucune de ces deux cohérences n'implique l'autre. Nous avons vu aussi précédemment qu'un WCN doit vérifier les cohérences AC* et DAC* pour vérifier FDAC* : la cohérence FDAC* implique donc AC* et DAC* et leur est $c_\emptyset$-supérieure. Même constat pour la cohérence EDAC* impliquant FDAC* (et EAC*). Dans [Cooper *et al.*, 2010], il a été montré que la cohérence VAC est $c_\emptyset$-supérieure à EDAC*. De plus, un WCN vérifiant VAC ne vérifie pas forcément AC* [Dehani, 2014]. La cohérence VAC n'implique donc pas les cohérences EDAC* et FDAC* qui impliquent elles-mêmes AC*. Enfin, il a été également montré dans [Cooper *et al.*, 2010] que la cohérence OSAC est $c_\emptyset$-supérieure à la cohérence VAC et que si un WCN vérifie la cohérence OSAC, alors il vérifie également la cohérence VAC : la cohérence OSAC implique donc la cohérence VAC.

Au delà des relations entre ces cohérences, le tableau 2.1 récapitule les complexités en temps et en espace dans le pire des cas des algorithmes présentés précédemment pour les établir sur un réseau.

Algorithme	Complexité en temps	Complexité en espace
OSAC [Cooper *et al.*, 2007]	$poly(ed+n)$[15]	$poly(ed^2+nd)$
VAC$_\epsilon$ [Cooper *et al.*, 2008]	$O(ed^2k/\epsilon)$	$O(ed)$
EDAC* [De Givry *et al.*, 2005]	$O(ed^2max(nd,k))$	$O(ed)$
FDAC* [Larrosa et Schiex, 2003]	$O(end^3)$	$O(ed)$
DAC* [Larrosa et Schiex, 2003]	$O(ed^2)$	$O(ed)$
AC*2001 [Larrosa, 2002]	$O(n^2d^3)$	$O(ed)$
NC* [Larrosa, 2002]	$O(nd)$	$O(nd)$

TAB. 2.1 – Complexités dans le pire des cas des algorithmes pour établir la cohérence correspondante.

Enfin, en complément de cette comparaison théorique, le tableau 2.2 propose une comparaison en pratique de l'utilisation des différentes cohérences locales souples lors de la résolution de problèmes WCSP. Même si la phase de recherche n'est abordée qu'à partir de la section suivante, le premier constat pouvant être fait est que l'utilisation d'une cohérence locale souple (et donc sujette à fournir une meilleure valeur de borne $c_\emptyset$) permet de visiter moins de nœuds durant la recherche : une meilleure valeur de borne $c_\emptyset$ permet de mieux filtrer les domaines, et ainsi mieux couper pour éviter des parcours inutiles dans l'espace de recherche (comme nous le verrons par la suite), ce qui permet une résolution plus efficace comme le montrent les temps de résolution et le nombre de nœuds visités pour chacun des problèmes.

2.3.5 Autre approche de calcul d'une borne inférieure : PFC-MPRDAC

Nous avons vu dans les sections précédentes que des bornes inférieures, de plus ou moins bonne qualité, pouvaient être obtenues par transferts de coûts en établissant des cohérences locales souples sur un réseau. L'approche PFC-MRDAC [Larrosa *et al.*, 1999], pour *Partial Forward Checking - Maintaining Reversible Directionnal Arc Consistency*, est une méthode sensiblement identique dans l'objectif visé mais bien différente dans le fonctionnement. L'approche PFC-MRDAC va calculer une borne inférieure correspondant à une sous-estimation du coût de n'importe quelle solution ou instanciation complète pouvant être obtenue dans un réseau à partir d'une instanciation partielle, et donc plus précisément lors de la

[15] L'algorithme de Karmakar s'exécute en temps polynomial par rapport à l'entrée $ed+n$

Instances		AC*	FDAC*	EDAC*
CELAR7-SUB1	CPU	3.04	1.33	1.87
	nœuds	91,457	17,830	14,286
SPOT5-404	CPU	451	415	217
	nœuds	$34M$	$18M$	$8M$
GRAPH-05	CPU	316	55	0.23
	nœuds	$2M$	322,114	901

TAB. 2.2 – Temps CPU (depuis un ordinateur équipé de processeurs Intel(R) Core(TM) i7-2820QM CPU 2.30GHz) et nombre de nœuds visités pour résoudre chacune des instances en maintenant les cohérences locales souples AC*, FDAC* et EDAC* via le solveur $ToulBar2$ (version 0.9).

prise d'une décision à chacun des nœuds d'un arbre de recherche. C'est la même chose que dans le cadre des cohérences locales souples si l'on considère les réseaux traités comme résultant de différentes assignations et filtrages et correspondant donc à une instanciation partielle donnée. Cependant, elle diffère dans le sens où cette fois-ci une borne inférieure est calculée par le biais d'une formule mathématique et non par le biais de transferts de coûts.

Introduite dans le cadre Max-CSP avec notamment des travaux préliminaires sur des méthodes de calcul du nombre contraintes violées dans [Freuder et Wallace, 1992], l'approche PFC-MRDAC s'étend également naturellement au cadre WCSP. Nous présentons dans un premier temps l'approche PFC-DAC [Larrosa et Meseguer, 1996], pour *Partial Forward Checking - Directionnal Arc Consistency*, qui est à la base de l'approche PFC-MRDAC. Soit un réseau P et une variable x de P. Nous notons $part^{nc}(x)$ l'ensemble (ou partition) des contraintes de P qui ne sont pas couvertes et qui portent sur x. Sur la même idée, notons $part^{nc}(\not x)$ (resp. $part^{c}(\not x)$) la partition composée des contraintes de P non couvertes (resp. couvertes) ne portant pas sur x. De plus, nous introduisons les valeurs $minCost(c_S) = min\{c_S(\tau) \mid \tau \in l(S)\}$ et $minCost(c_S, x, a) = min\{c_S(\tau) \mid \tau \in l(S) \wedge \tau[x] = a\}$: le coût $minCost(c_S)$ correspond au coût minimal des tuples dans la contrainte c_S et le coût $minCost(c_S, x, a)$ correspond au coût minimal des tuples contenant (x, a) dans la contrainte c_S. La borne inférieure $lb(x, a)$ calculée par l'approche PFC-DAC pour une valeur (x, a) est égale à :

$$lb(x,a) = \bigoplus_{c_S \in part^{c}(\not x)} c_S(I) \oplus \bigoplus_{c_S \in part^{nc}(x)} minCost(c_S, x, a) \oplus \bigoplus_{c_S \in part^{nc}(\not x)} minCost(c_S) \quad (2.13)$$

La première partie de la formule (2.13) correspond au coût de l'instanciation partielle courante I. Appelée aussi *distance(P)*, elle représente la totalité des coûts affectés par les contraintes de P couvertes par l'instanciation partielle courante. La deuxième partie correspond à une sous-estimation des coûts des supports possibles pour la valeur (x, a) dans chacune des contraintes de P non couvertes et portant sur la variable x. Enfin, la troisième partie correspond à une sous-estimation des coûts des instanciations ou tuples possibles dans chacune des contraintes de P non couvertes et ne portant pas sur la variable x. La formule totale donne une borne inférieure du coût que représente l'assignation $x = a$. La propriété de *Partial Forward Checking* de cette approche provient du fait que l'impact de l'instanciation partielle courante (*Partial*) est calculé sur les variables futures (*Forward Checking*). La propriété de *Directionnal Arc Consistency* permet d'éviter de calculer des coûts redondants au niveau des contraintes non couvertes : en effet, les contraintes non couvertes et leurs coûts vont être considérés une unique fois selon l'ordre fixé pour les variables du réseau. Considérons par exemple deux variables x et y (non assignées) telles que $x < y$, ainsi que l'existence d'une contrainte (non couverte) c_{xy}. Lors du calcul de borne, afin d'éviter de compter deux fois les coûts induits par la contrainte c_{xy}, celle-ci appartiendra uniquement à

la partition de contraintes non couvertes associée à x car x représente la variable voisine inférieure au sein du scope de c_{xy}.

Considérons le réseau illustré à la figure 2.20. L'instanciation partielle courante correspond à $w = b$ et on souhaite calculer la borne inférieure associée à la décision $x = a$, c'est à dire $lb(x, a)$.

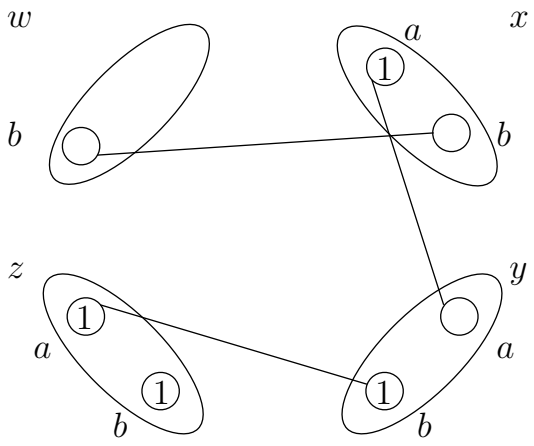


FIG. 2.20 – Réseau WCSP composé de quatre variables ordonnées $w < x < y < z$ avec $w = b$, quatre contraintes unaires c_w, c_x, c_y, c_z et trois contraintes binaires c_{wx}, c_{xy}, c_{yz}.

Selon la formule (2.13), la première étape consiste à calculer le coût de l'instanciation partielle courante, c'est à dire le coût de $\{(w, b)\}$, qui correspond en fait à la somme des coûts affectés par les contraintes couvertes appartenant à $part^c(\not{x})$ (figure 2.21). En l'occurrence, il s'agit de la contrainte c_w et le coût affecté est $c_w(b) = 0$ (car $w = b$). La deuxième étape consiste à calculer pour les variables futures le coût impacté par la décision $x = a$ au niveau des contraintes non couvertes portant sur la variable x appartenant à $part^{nc}(x)$ (Figure 2.21). En l'occurrence, il s'agit d'une part des contraintes c_{wx} et c_{xy} dans lesquelles on va chercher le coût minimal parmi les tuples contenant la valeur (x, a) : il s'agit du tuple (b, a) de coût égal à 0 pour la contrainte c_{wx} et du tuple (a, b) de coût égal à 0 pour la contrainte c_{xy}. D'autre part, il s'agit de la contrainte c_x pour laquelle le coût affecté est $c_x(a) = 1$. Le coût affecté par la deuxième étape est donc égal à $c_x(a) \oplus c_{wx}(b, a) \oplus c_{xy}(a, b) = 1 \oplus 0 \oplus 0 = 1$. Pour terminer, la troisième étape consiste à prospecter dans les variables futures les coûts minimaux possibles dans les contraintes selon l'ordre imposé par DAC, c'est à dire les contraintes non couvertes ne portant pas sur la variable x et appartenant donc à $part^{nc}(\not{x})$. En l'occurrence, il s'agit des contraintes c_y, c_z et c_{yz}. Dans la contrainte c_y le coût minimal correspond à la valeur (y, a) avec $c_y(a) = 0$, dans la contrainte c_z le coût minimal correspond à la valeur (z, a) avec $c_z(a) = 1$, puis dans la contrainte c_{yz} le coût minimal correspond au tuple (a, a) avec $c_{yz} = 0$: c'est à dire un coût total égal à $0 \oplus 1 \oplus 0 = 1$. Nous obtenons donc au final $lb(x, a) = 0 \oplus 1 \oplus 1 = 2$. Cela veut donc dire que l'instanciation partielle courante $\{(w, b), (x, a)\}$ conduira à une instanciation complète d'un coût au moins égal à 2.

L'approche PFC-MRDAC est basée sur le fait que l'ordre des variables utilisé va être dynamique au lieu d'être statique. Imaginons le graphe orienté dont les sommets représentent les variables et les arcs représentent les contraintes, et pour lesquels l'orientation dépend de l'ordre établi sur les variables dans le réseau. Les orientations de ces arcs (et donc l'ordre des variables) vont pouvoir être inversées (d'où le nom *reversible*) afin d'obtenir une meilleure borne, et cela durant la recherche (d'où le nom *maintaining*) et en considérant les informations du réseau courant. À noter que la recherche des inversions permettant d'obtenir le graphe amenant à la meilleure borne inférieure représente un problème `NP-Difficile`. Des travaux plus récents basés sur l'approche PFC-MRDAC, à savoir les approches PFC-PRDAC (resp. PFC-MPRDAC) pour *Partial Forward Checking - (resp. Maintaining) Partition Reversible Direction-*

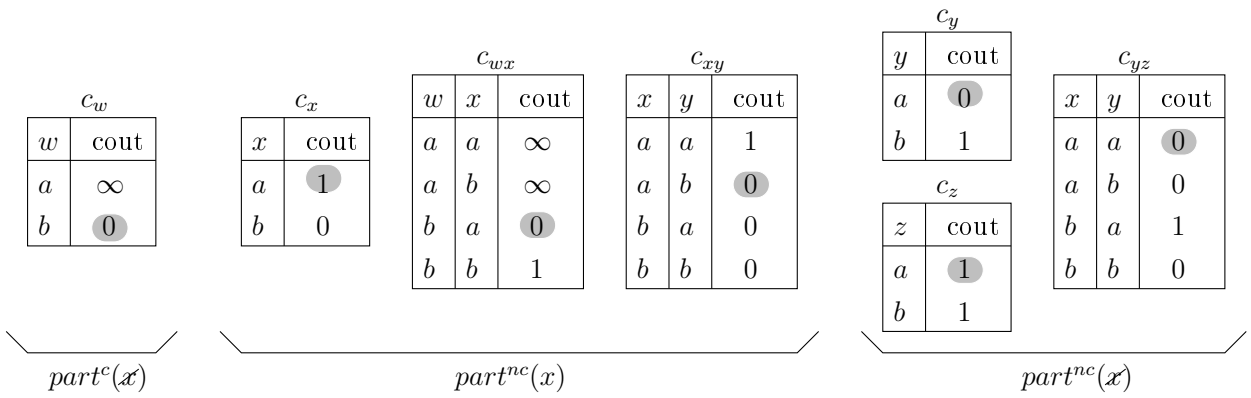


FIG. 2.21 – Les différents coûts associés aux contraintes du réseau avec en grisé les coûts utilisés pour le calcul de la borne $lb(x, a)$.

nal Arc Consistency, ont été proposés dans [Larrosa et Meseguer, 1999]. Elles consistent à calculer des bornes inférieures basées sur des partitions de variables futures, pouvant être composées de sous-ensembles plus ou moins grands de variables futures, agissant directement sur la qualité de la borne. De ce fait, ces bornes sont ainsi appelées *partition-based - lower bound*. L'idée est basée sur le fait de prendre en compte dans le calcul des bornes inférieures les coûts qui n'ont pas été pris en compte dans le calcul précédent mais qui sont obligatoires dans le calcul du coût de la meilleure solution et donc dans la solution la moins chère.

Tout comme dans le cadre CSP, les techniques d'inférence appliquées en pré-traitement ne sont malheureusement pas suffisantes dans le cadre WCSP. Une phase de recherche est généralement nécessaire et dans le cadre des problèmes de satisfaction de contraintes pondérées, et plus généralement les problèmes d'optimisations combinatoires, deux méthodes de recherche peuvent être utilisées : méthode de recherche complète et méthode de recherche incomplète. Nous présentons dans les sections suivantes ces deux modes de recherche proposés dans le cadre WCSP. À noter que tout comme dans le cadre CSP des approches hybrides ont été proposées pour le cadre WCSP (voir par exemple [Loudin et Boizumault, 2001]) mais elles ne seront pas plus détaillées dans ce document.

2.4 Stratégies de recherche complète

2.4.1 Approche arborescente de type séparation et évaluation DFBB

Les algorithmes de résolution complète de WCSP les plus efficaces correspondent à des approches arborescentes de type séparation et évaluation en profondeur d'abord notées *DFBB* (*Depth-First Branch-and-Bound*). Présentée dans un cadre général dans la section 2.2.3, ce type d'approche exploite deux bornes : une borne majorante appelée UB (Upper Bound) correspondant au coût de la meilleure solution trouvée et mise à jour durant toute la recherche, puis une borne minorante appelée LB (Lower Bound) obtenue pour chaque nœud durant la recherche par évaluation et correspondant au coût minimal de n'importe quelle solution du sous-problème associé. D'un point de vue général, ces approches sont basées sur un modèle SER (*Séparation Évaluation Retour-arrière*). Comme pour le modèle BPRA dans le cadre CSP, l'intérêt de ce modèle réside bien sûr dans la manière de combiner les techniques proposées par ces différents composants :

– Séparation (ou branchement) : les mêmes possibilités que dans le cadre CSP.

- Évaluation : la manière d'évaluer mathématiquement un minorant du coût de n'importe quelle solution pouvant être obtenue dans le sous-problème correspondant à chaque nœud considéré lors de la recherche (mécanismes d'inférence mis en œuvre : cohérences locales souples établies, approche PFC-MPRDAC).
- Retour-arrière : en cas d'échec, retour en arrière à la dernière décision prise (retour en arrière chronologique).

L'algorithme générique 25, basé sur le modèle SER, trouve une solution optimale et prouve son optimalité dans l'ensemble des solutions possibles pour le problème traité. Il commence par estimer (composant EVALUATION) le coût minimal que représente toute instanciation partielle dans le problème ou réseau considéré. Comme cela a été expliqué dans les sections précédentes, il y a deux manières de faire : soit établir une cohérence locale souple basée sur le calcul d'une borne inférieure $c_\emptyset$, soit calculer une borne inférieure par la méthode PFC-MPRDAC. Cette évaluation renvoie le coût ∞ (ligne 1) si un domaine vide apparaît (lors de l'établissement des cohérences locales souples) ou si la borne inférieure calculée ($c_\emptyset$ ou minorant PFC-MPRDAC) ne permet pas d'améliorer la meilleure solution trouvée car $LB >= UB$. À noter qu'au début d'une résolution d'un WCSP, la borne majorante est généralement initialisée par le coût interdit k défini pour le réseau (car objectif de minimisation). Dans ce cas, faux est retourné car c'est un échec et on ne doit pas continuer dans cette branche de l'arbre. Si au contraire le sous-réseau permet potentiellement d'améliorer la meilleure solution trouvée, on continue. De plus, si une valeur a été affectée à chacune des variables ou que chaque variable est singleton, alors une solution a été trouvée et la borne majorante UB est mise à jour (ligne 4) si le coût de cette solution trouvée l'améliore (c'est à dire si le coût est inférieur à UB dans notre objectif de minimisation). Dans le cas où l'instanciation courante n'est pas complète, on continue de séparer (brancher) en effectuant les prochaines décisions (composant SEPARATION). On applique chaque décision prise au réseau courant et on continue de chercher une solution optimale dans le réseau modifié. Si la suite de la résolution a conduit à un succès (ligne 11), on a trouvé une solution (optimale ou non) pour le réseau, sinon c'est que la décision prise a amené à un échec et qu'il faut effectuer un retour en arrière par rapport à cette décision (composant RETOUR-ARRIERE englobé dans l'appel récursif de l'algorithme).

Algorithm 25: solveWCSP ($P = (\mathcal{X}, \mathcal{C}, k)$: WCSP) : Booléen

```
1  si EVALUATION(P, UB) = ∞ alors Retourner faux
2  si ∀x ∈ X, |dom(x)| = 1 alors
3      si cout instanciation complete < UB alors
4          UB ← cout instanciation complete
5          Retourner vrai
6      sinon
7          Retourner faux
8  D ←SEPARATION(P)
9  pour chaque δ ∈ D faire
10     P' ← P |_δ
11     si solveWCSP(P') alors
12         Retourner vrai
13 Retourner faux
```

La figure 2.22 illustre le fonctionnement de l'algorithme 25. On y voit notamment l'instanciation partielle courante de la résolution et qui représente un parcours dans l'espace de recherche (représenté

par le triangle blanc) en profondeur d'abord. À cet instant donné, la meilleure solution de coût UB a été trouvée dans la partie "gauche" de l'espace de recherche. Cette solution améliore les meilleures solutions trouvées auparavant durant la recherche et représentées par UB_0 et UB_1. Au niveau du nœud courant, une évaluation est réalisée afin de savoir si le sous-problème correspondant (sous-espace de recherche grisé) peut potentiellement conduire à une solution améliorant la meilleure solution trouvée. La borne inférieure LB est soit calculée en établissant une cohérence locale souple, soit calculée par l'approche PFC-MPRDAC. Si LB≥UB, ça ne sert à rien de parcourir ce sous-espace de recherche et un retour en arrière avant le nœud courant (dernière décision prise) est alors effectué : l'espace de recherche sous ce nœud est alors dit coupé car non exploré. Pour conclure, il faut savoir que les méthodes les plus efficaces actuellement se composent d'un branchement binaire et d'une technique de retour en arrière chronologique (standard).

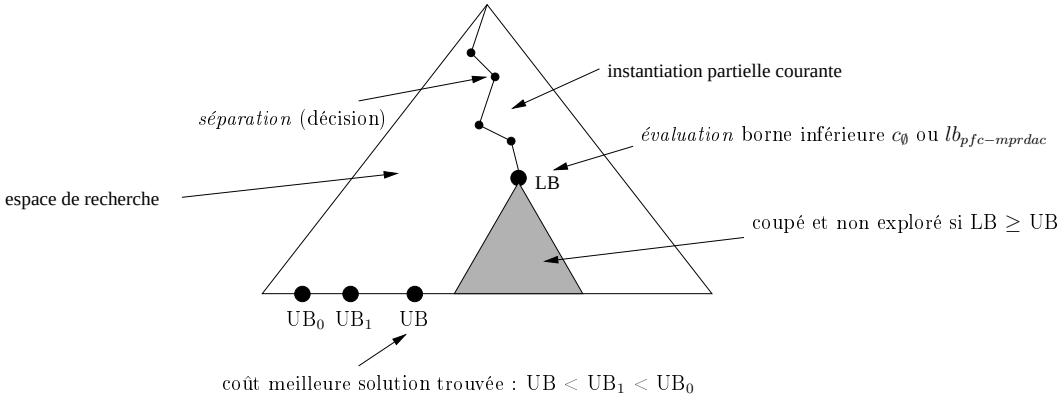


FIG. 2.22 – Principe de fonctionnement de la séparation et évaluation en profondeur d'abord dans le contexte WCSP.

2.4.2 DFBB exploitant une décomposition arborescente et des cohérences locales souples

Que ce soit dans le cadre CSP ou WCSP, l'idée générale de la décomposition arborescente (abordée dans [Robertson et Seymour, 1986]) est d'exploiter les propriétés structurelles caractérisant les problèmes à traiter. Plus précisément, ce qui la diffère des approches arborescentes classiques présentées dans la section précédente, c'est qu'elle vise à exploiter le découpage possible de certains problèmes en sous-problèmes structurellement indépendants plus faciles à résoudre que le problème original. Introduite notamment pour les cadres CSP [Dechter et Pearl, 1989] et Max-CSP [Jégou et Terrioux, 2004], et plus généralement pour le cadre VCSP [Verfaillie *et al.*, 1996], elle a également été exploitée pour le cadre WCSP : l'idée proposée dans [De Givry *et al.*, 2006] consiste à combiner une décomposition arborescente avec une recherche de type séparation et évaluation en profondeur d'abord avec utilisation des cohérences locales souples pour résoudre un problème WCSP.

Dans [De Givry *et al.*, 2006], une décomposition arborescente d'un réseau de contraintes pondérées correspond à un arbre dont les différents nœuds sont des regroupements de variables connectées, appelés aussi *clusters*. Les branches de cet arbre correspondent aux liaisons entre les *clusters*, à savoir les ensembles de variables communes entre ces *clusters* et appelés *séparateurs*. Bien évidemment, toutes les variables et les contraintes du réseau sont couvertes par cet arbre. L'idée exploitée dans [De Givry *et al.*, 2006] repose sur une des caractéristiques essentielles d'une décomposition arborescente : une fois que toutes les variables appartenant à un *séparateur* ont été assignées, les problèmes qui étaient liés par ce *séparateur* sont alors *déconnectés* et peuvent être résolus indépendamment. Le

problème ou réseau original peut ainsi être découpé en sous réseaux. Bien sûr, la décomposition arborescente ne peut être exploitée que lorsqu'un certain choix dans les variables est imposé. En effet, les variables englobées dans un *cluster* doivent être assignées après les variables englobées dans son *cluster* père[16], mais avant les variables englobées dans ses *clusters* fils.

La méthode la plus efficace proposée dans [De Givry *et al.*, 2006] exploite une décomposition arborescente de type DFBB (*Depth-First Branch-and-Bound*) avec l'exploitation de cohérences locales souples. De plus, l'utilisation de majorants améliorés pour chacun des sous-problèmes est incorporée dans cette approche : elle consiste à calculer à partir des valeurs du majorant et du minorant du problème père de ce sous-problème un majorant dit *amélioré* et de l'utiliser au lieu du majorant global du problème pour une coupe plus efficace lors de la résolution. Alors que l'utilisation dans un contexte classique d'une cohérence locale souple permet d'obtenir un minorant de l'optimum du sous-problème courant, la décomposition permet d'appliquer cette cohérence locale souple sur un sous-problème courant décomposé en sous-problèmes plus petits et ainsi obtenir, facilement, des minorants pour chacun de ces sous-problèmes. Cependant, les EPT utilisées pour établir une cohérence locale souple préservent certes le coût de toute instanciation complète dans le réseau, mais des coûts peuvent être transférés entre les différents sous réseaux (*clusters*) issues de la décomposition, et de ce fait les informations relatives à ces sous réseaux (minorant, coût optimal,...) peuvent devenir obsolètes. Pour pallier ces inconvénients, différentes techniques ont été utilisées : stocker les coûts transférés par projection et extension afin de ré-équilibrer et corriger les informations de coûts utiles et propres aux sous-problèmes, utilisation de minorants locaux $c_\emptyset^i$ en complément du minorant global $c_\emptyset$ pour y stocker les projections unaires effectuées dans un sous-problème i, etc. De plus, à partir de ces minorants locaux, une dernière idée proposée est d'effectuer des coupes dites *locales* (car coupe par rapport à un majorant local et non global) ou filtrage de valeurs si la somme du coût unaire d'une valeur additionnée au minorant local du sous-problème à résoudre atteint le majorant local de ce sous-problème. Les bornes théoriques obtenues pour cette approche, basées notamment sur le nombre de *clusters* ou sous-problèmes visités, sont moins bonnes qu'une approche de décomposition arborescente classique avec séparation et évaluation, mais présentent cependant une meilleure efficacité en pratique. Évidemment, cette approche, et plus généralement la décomposition arborescente, est particulièrement bien adaptée et envisageable pour des problèmes fortement structurés comme par exemple les problèmes de la série RLFAP (Radio Link Frequency Assignment Problem) [Cabon *et al.*, 1999]. Même constat pour la méthode proposée plus récemment dans [Sánchez *et al.*, 2009] où l'approche DFBB est remplacée par l'approche RDS (pour *Russian Doll Search*) au sein de la décomposition arborescente BTD (pour *Backtrack bounded by Tree Decomposition*) pour former la combinaison BTD-RDS.

2.4.3 Des heuristiques pour orienter les choix

Tout comme dans le cadre de la résolution des CSP, l'utilisation d'heuristiques est primordiale dans la résolution des WCSP par approche arborescente de type séparation et évaluation, ainsi que dans le contexte d'une recherche incomplète comme nous le verrons dans la section 2.5. Bien sûr, les heuristiques génériques du cadre CSP peuvent très bien être utilisées dans le cadre WCSP comme par exemple *deg*, *dom*, etc, les notions de degré et de taille de domaine étant toujours valables et donc exploitables au cadre WCSP. Cependant, des heuristiques génériques et plus adaptées au cadre WCSP, prenant en compte notamment la notion de coût, ont été proposées. Même si elles ne sont pas aussi nombreuses que pour les CSP, des heuristiques de choix de variable et de choix de valeur ont donc été étudiées spécifiquement pour les WCSP. Mises en évidence pour la plupart d'entre elles dans [Heras et Larrosa, 2006], les heuristiques de choix de variable pour le cadre WCSP peuvent être également répertoriées en trois

[16] Définitions de *père* et de *fils* de la théorie des graphes

catégories : statiques, dynamiques et adaptatives. De manière générale, la tendance derrière ces heuristiques est la même que dans le cadre des CSP : choisir d'abord les variables ayant le plus de chance de mener à un échec. Concernant les heuristiques statiques, le choix de variable peut se faire selon :

- *lexico* : les variables sont choisies par ordre lexicographique de leurs noms et son intérêt est le même que dans le cadre CSP (même constat pour l'heuristique *random*).
- *deg* (*maxdeg*) : les variables sont choisies par ordre décroissant de leur degré initial, c'est l'équivalent de l'heuristique deg (*maxdeg*) du cadre CSP.
- *2-side Jeroslow-Wang-like* : inspirée du cadre SAT [Jeroslow et Wang, 1990], cette heuristique présentée notamment dans [De Givry *et al.*, 2003] consiste à ordonner les variables selon un ratio entre la somme des moyennes des coûts dans les contraintes liées à cette variable et la taille de leur domaine. Pour favoriser l'apparition rapide des échecs, les variables possédant les ratios supérieurs sont choisies prioritairement pour mener aux solutions de coûts les plus élevés.

Concernant les heuristiques dynamiques, nous trouvons :

- *suc* : les variables sont choisies par ordre croissant de la somme des coûts unaires des valeurs présentes dans leur domaine courant.
- *dom* : les variables sont choisies par ordre croissant de la taille de leur domaine courant.

Finalement, nous trouvons des heuristiques adaptatives *RANK* et *dom/HQ* [Levasseur, 2008] basées sur un critère global nommé *H-quality*. À partir d'un historique des solutions trouvées depuis le début de la résolution (d'où le qualificatif adaptative), le critère *H-quality* représente l'aptitude des variables et des valeurs à amener à des solutions de qualité et ces heuristiques consistent alors à ordonner les variables en considérant prioritairement celles proposant une valeur *H-quality* minimale.

Évidemment, ces heuristiques peuvent être plus ou moins efficaces selon le type de problème ou le type de filtrage et les cohérences locales souples utilisés lors de la résolution. Par exemple, il a été montré dans [Heras et Larrosa, 2006] que les heuristiques *inv-deg* et *suc* sont les plus efficaces lors du maintien de la cohérence FDAC* alors que les résultats ne sont pas aussi bons pour la cohérence EDAC*. De même, des résultats intéressants ont été mis en évidence dans [De Givry *et al.*, 2003] en utilisant l'heuristique *2-side Jeroslow-Wang-like* pour maintenir FDAC* lors de la résolution de problèmes WCSP aléatoires. Il y a bien moins d'heuristiques de choix de valeur et la tendance est la même que dans le cadre des CSP : choisir la valeur qui devrait permettre de parvenir à une solution de qualité rapidement. Pour cela, plusieurs méthodes ont été proposées :

- *lexico* et *random* : comme pour l'heuristique de choix de variable, les valeurs sont choisies par ordre lexicographique de leurs noms et son intérêt est le même que dans le cadre CSP (même constat pour l'heuristique *random*).
- *min-ac* [Larrosa et Schiex, 2003] : les valeurs sont choisies par ordre croissant du nombre d'arc-incohérences résultant de l'affectation de cette valeur à la variable correspondante.
- *min-cout-unaire* : les valeurs d'une variable sont choisies par ordre croissant du coût unaire correspondant pour cette valeur dans la contrainte unaire définie pour cette variable. Quand aucune contrainte unaire n'est définie pour la variable où en cas d'égalité des coûts unaires, les valeurs sont choisies par ordre lexicographique ou de façon aléatoire.
- *HQOnce, HQAll* [Levasseur *et al.*, 2007] : les valeurs dans les domaines des variables sont choisies par ordre croissant de leur critère global *H-quality*. Pour une valeur, le critère *H-quality* correspond au coût de la solution de coût minimal présente dans l'historique des solutions rencontrées durant la résolution et contenant cette valeur.

Pour résumer brièvement, il a été montré dans [Levasseur *et al.*, 2007] une efficacité similaire des heuristiques de choix de valeur *min-ac*, *HQOnce* et *HQAll* pour des problèmes réels ou aléatoires, alors que cette efficacité tend à être améliorée avec *HQOnce* et *HQAll* pour ce genre de problèmes présentant une connectivité significative (nombre de variables voisines élevé pour chaque variable du réseau). Mais là encore, des informations plus précises pour comparer ces différentes heuristiques sont disponibles

dans les différents documents de référence.

2.5 Stratégies de recherche incomplète

Dans le cadre d'un problème d'optimisation, une recherche incomplète ou partielle permet d'obtenir une solution qui n'est pas forcément la solution optimale du problème à résoudre : on parle alors d'approche d'approximation et de solutions approchées de la solution optimale. Plus l'algorithme de recherche est efficace, plus la solution obtenue sera proche de la solution optimale, voire même une solution optimale dans certains cas. Lorsque les problèmes traités sont complexes au point que les méthodes complètes ne permettent pas d'obtenir une solution optimale dans un temps raisonnable, les méthodes incomplètes parviennent généralement à identifier une "bonne" solution. Présentées de façon générale dans la section 2.2.3, il faut savoir que ces méthodes incomplètes s'avèrent plutôt efficaces quand elles sont appliquées au cadre WCSP. Nous présentons ci-dessous leur adaptation dans ce cadre ainsi qu'une description légère de leur fonctionnement, tout en précisant que ces différentes techniques sont exploitées dans la bibliothèque INCOP proposée par le solveur $ToulBar2$, bibliothèque utilisée pour comparaison de techniques lors de nos expérimentations. La figure 2.23 représente un réseau WCSP simple qui va nous servir à expliquer les différentes stratégies de recherche incomplète présentées dans cette section. Ce problème WCSP porte sur cinq variables x_0,x_1,x_2,x_3, et x_4 avec $dom(x_0) = dom(x_1) = dom(x_2) = dom(x_3) = dom(x_4) = \{a, b, c, d\}$. Dans le cadre des recherches incomplètes, nous verrons que les techniques de transferts de coûts (et donc la contrainte $c_\emptyset$) ne sont pas utilisées.

c_{x_1}

x_1	cout
a	2
b	2
c	1
d	1

c_{x_3}

x_3	cout
a	4
b	3
c	2
d	3

$c_{x_1x_3}$ (cout par defaut=3)

x_1	x_3	cout
d	b	1

$c_{x_2x_3}$ (cout par defaut=3)

x_2	x_3	cout
b	b	7
b	c	0

FIG. 2.23 – Un réseau de contraintes pondérées composé de deux contraintes souples unaires c_{x_1} et c_{x_3}, deux contraintes souples binaires $c_{x_1x_3}$ et $c_{x_2x_3}$ et un coût interdit $k = 15$.

2.5.1 Méta-heuristiques de recherche locale

Généralement, les méthodes incomplètes sont basées sur une recherche locale qui consiste à modifier itérativement une instanciation complète initiale d'un problème dans le but de l'améliorer suivant l'objectif de ce problème. Dans le cadre du problème introduit dans 2.23, une instanciation complète initiale possible est représentée dans la figure 2.24. Il s'agit de l'instanciation complète $s = \{(x_0, b), (x_1, c), (x_2, a), (x_3, a), (x_4, c)\}$ dont le coût est égal à $\mathcal{V}(s) = 11$ et qui est de ce fait une solution au problème (car $\mathcal{V}(s) < k$), c'est à dire la somme de $c_{x_1}(c) \oplus c_{x_3}(a) \oplus c_{x_1x_3}((c, a)) \oplus c_{x_2x_3}((a, a))$. L'objectif de la recherche locale va être d'améliorer cette solution en trouvant une solution de coût strictement inférieur à $\mathcal{V}(s)$.

Généralement, l'instanciation initiale est obtenue aléatoirement mais il est indéniable que commencer la recherche locale depuis une bonne instanciation initiale, c'est à dire présentant de bonnes caractéristiques par rapport au problème traité et l'objectif, représente un avantage certain. À partir de cette

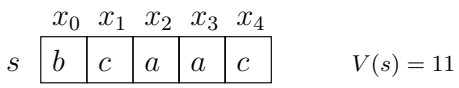


FIG. 2.24 – Une instanciation complète de coût égal à 11 dans le problème à la figure 2.23.

instanciation initiale, la recherche locale va se déplacer dans l'espace de recherche en sautant d'instanciation complète en instanciation complète jusqu'à atteindre un critère d'arrêt. Seule la dernière instanciation complète qui améliore la meilleure solution trouvée est acceptée et conservée, toutes les autres étant refusées. Ces mouvements sont définis par une *structure de voisinage* qui caractérise une stratégie de recherche locale. La figure 2.25 présente un exemple dans lequel c'est la variable x_3 qui a été choisie comme structure de voisinage pour notre problème. En d'autres termes, une instanciation voisine de l'instanciation courante va être obtenue à chaque itération en modifiant celle-ci au niveau de la variable x_3. La nature de cette modification est définie par une *méthode de transformation de voisinage* : dans notre cas elle consiste simplement à affecter une valeur à la variable x_3 autre que celle déjà choisie dans l'instanciation courante. On parle de recherche locale car le déplacement dans l'espace de recherche se fait plus précisément dans le *voisinage*, c'est à dire parmi les instanciations voisines présentes dans le voisinage de l'instanciation courante.

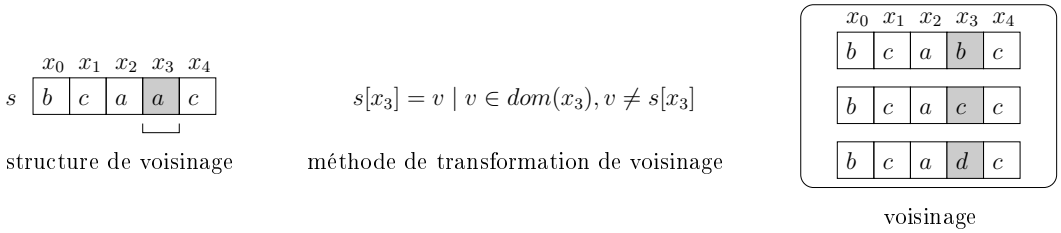


FIG. 2.25 – Exemple d'un voisinage pour le problème de la figure 2.23.

Considérant le problème introduit à la figure 2.23, la figure 2.26 présente un exemple de fonctionnement sur trois itérations d'une recherche locale standard basée sur le voisinage présenté dans la figure 2.25. Une première instanciation complète s_1 voisine de s_0 est obtenue lors de la première itération en remplaçant la valeur (x_3, a) de s_0 par (x_3, b). L'instanciation complète s_1 obtenue est une solution qui améliore s_0 car $\mathcal{V}(s_1) < \mathcal{V}(s_0)$ et elle est conservée comme meilleure solution trouvée. Comme dans une recherche complète de type DFBB, le coût interdit du problème, initialement égal à $k = 15$ correspond au fur et à mesure au coût de chaque nouvelle meilleure solution trouvée, c'est à dire ici $k = 10$. Ensuite, s_2 est obtenue à partir de s_1 en remplaçant la valeur (x_3, b) par (x_3, c). La nouvelle solution obtenue est meilleure car $\mathcal{V}(s_2) < \mathcal{V}(s_1)$: elle est donc conservée au dépend de s_1 qui devient ignorée. Lors de la troisième génération, la solution obtenue s_3 n'améliore pas la meilleure solution trouvée ($\mathcal{V}(s_3) > \mathcal{V}(s_2)$), elle est donc refusée et s_2 correspond à la meilleure solution trouvée après trois itérations.

Dans notre exemple présenté à la figure 2.26, la solution obtenue s_2 représente une solution optimale locale pour le voisinage défini dans la figure 2.25 mais ne représente pas forcément une solution optimale globale pour le problème complet. Rester dans ce voisinage ne suffit donc peut être pas si l'on souhaite atteindre une solution optimale au problème global. Comme cela a été dit dans la section 2.2.3, les méta-heuristiques permettent de sortir de ces *minimas locaux* en variant entre phase d'*intensification* et phase de *diversification*. Les différentes méta-heuristiques standard appliquées au cadre WCSP s'appuient donc sur une phase de recherche locale pour l'*intensification*. Ensuite, chacune d'entre elles possède sa propre technique pour la *diversification*. Certaines d'entre elles vont par

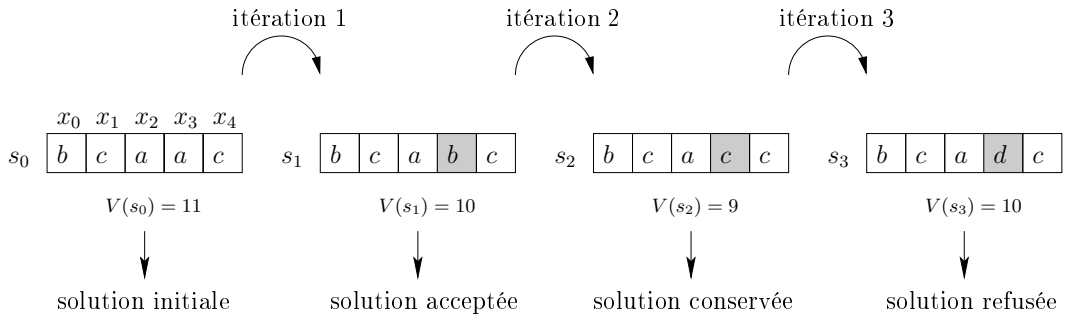


FIG. 2.26 – Recherche locale étalée sur 3 générations à partir de l'instanciation initiale de la figure 2.24 et selon la structure de voisinage proposée dans la figure 2.25.

exemple agir sur la structure de voisinage pour diversifier leur recherche, c'est le cas notamment pour les approches VNS (*Variable Neighborhood Search*) [Mladenović et Hansen, 1997] et DGVNS (*Decomposition Guided VNS*) [Fontaine *et al.*, 2013], LNS (*Large Neighborhood Search*) [Shaw, 1998], ainsi que la méthode hybride VNS/LDS+CP [Loudni et Boizumault, 2008] exploitant la recherche arborescente non systématique LDS (*Limited Discrepancy Search*) [Harvey et Ginsberg, 1995]. Plus généralement, il s'agit de toutes les méthodes de type *grand voisinage évolutif* abordées dans [Pisinger et Ropke, 2010] dans lesquelles les propriétés de structure des voisinages sont largement exploitées. Plus précisément, l'approche VNS consiste notamment à faire varier la structure de voisinage et ainsi étendre, ou en d'autres termes diversifier, le voisinage parcouru par le biais d'une structure de voisinage supplémentaire et de la transformation de voisinage correspondante.

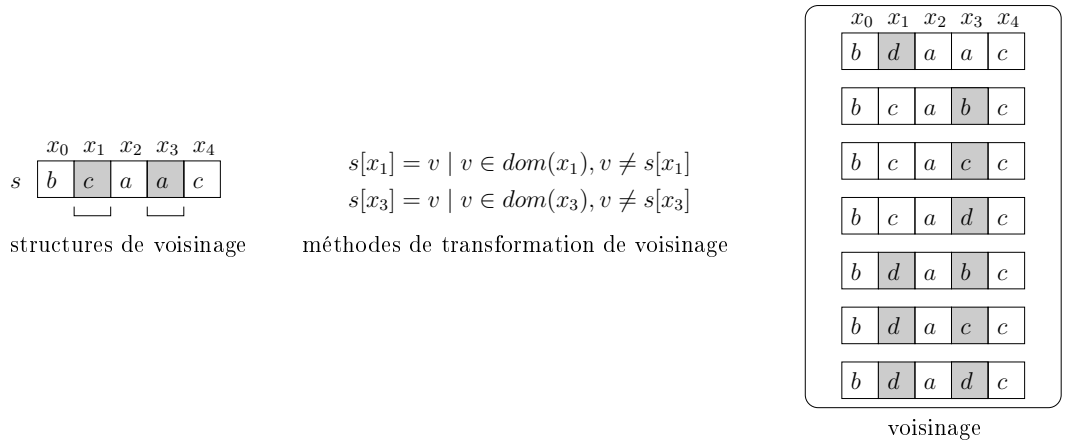


FIG. 2.27 – Exemple de deux voisinages pour une instanciation complète du problème de la figure 2.23.

Dans notre exemple présenté à la figure 2.28, la solution s_1 est obtenue à partir de la solution s_0 par rapport au voisinage basé sur la variable x_3, puis la solution s_2 est obtenue à partir de la solution s_1 par rapport au nouveau voisinage basé sur la variable x_1 et introduit dans la figure 2.27. Cette solution est conservée ($\mathcal{V}(s_3) > \mathcal{V}(s_2)$) tandis que la dernière solution obtenue à nouveau à partir du premier voisinage est refusée. On voit que varier dans les voisinages a permis d'obtenir une solution meilleure que celle obtenue par une recherche locale de base. Changer de voisinage (varier entre voisinage basé sur x_3 et voisinage basé sur x_1) a donc permis de sortir du minimum local rencontré dans l'exemple de la figure 2.26. D'autres méta-heuristiques comme la *recherche avec tabous* et le *recuit simulé* vont plutôt exploiter des structures particulières pour diversifier leur recherche, comme par exemple la notion

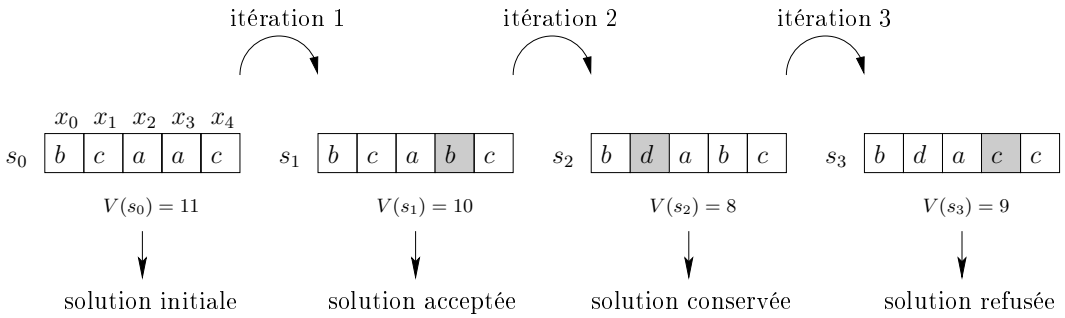


FIG. 2.28 – Recherche locale étalée sur 3 générations à partir de l'instanciation initiale de la figure 2.24 et selon la structure de voisinage proposée dans la figure 2.27.

de degré de température pour le recuit simulé. Derrière cette notion de température se cache une idée de seuil de tolérance qui permet à ces techniques de tolérer et d'accepter des solutions moins bonnes que la meilleure solution courante trouvée et cela durant un certain nombre d'itérations. Cette méthode leur permet de sortir des minimas locaux en diversifiant l'espace de recherche et plus précisément le voisinage. La solution est tolérée et donc acceptée quand elle ne dépasse pas le seuil. Nous considérons ici le parcours d'une troisième structure de voisinage, en l'occurrence via un changement de valeur pour la variable x_2.

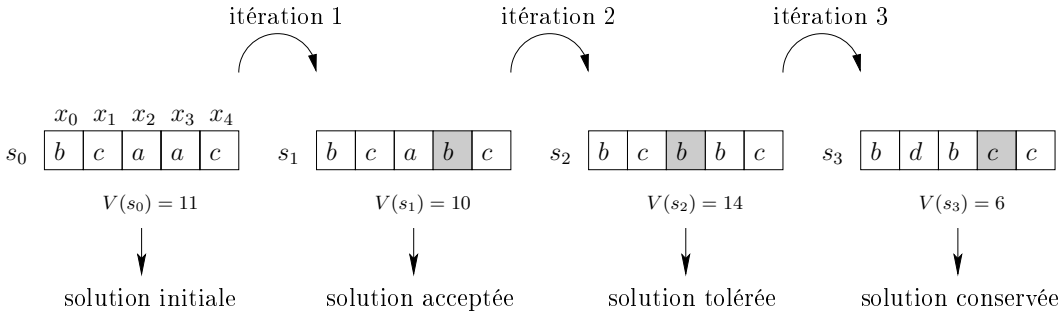


FIG. 2.29 – Recherche locale étalée sur 3 générations à partir de l'instanciation initiale de la figure 2.24 avec un seuil égal à 15.

Dans notre exemple présenté à la figure 2.29, la solution s_1 est obtenue à partir de la solution s_0 par rapport au voisinage basé sur la variable x_3, puis la solution s_2 est obtenue à partir de la solution s_1 par rapport au voisinage basé sur la variable x_2. Cette solution n'améliore pas la meilleure solution trouvée s_1 mais elle est tolérée car son coût ne dépasse pas le seuil autorisé égal à 15. La solution s_3 obtenue à partir de cette solution tolérée améliore quant à elle la meilleure solution trouvée. On s'aperçoit que grâce à un seuil de tolérance autorisé, on a obtenu une meilleure solution qu'avec les approches présentées ci-dessus. Évidemment toutes ces techniques sont très paramétrables, on peut les combiner entre elles, et il est très rare de trouver une configuration qui soit valable pour tout type de problème. Bien évidemment, même si les heuristiques présentées dans la section 2.4.3 sont destinées à une recherche (complète) de type DFBB, certaines d'entre elles peuvent être utilisées pour explorer efficacement l'espace de recherche dans le cadre d'une recherche incomplète. Les heuristiques (de choix de variable ou de valeur) spécifiques à des recherches incomplètes ne sont pas abordées dans ce manuscrit.

2.5.2 Méta-heuristiques évolutionnaires

Les méta-heuristiques évolutionnaires consistent, contrairement à la recherche locale, à considérer et modifier itérativement un ensemble initial d'instanciations complètes ou *individus*, appelé aussi *population d'individus initiale*, afin d'obtenir une nouvelle population contenant de meilleures solutions. À partir de cette population initiale là aussi aléatoire, chaque génération se décompose en trois étapes. La première étape consiste à évaluer par une fonction appelée *fonction de fitness* la qualité de chacune des solutions présentes dans la population courante et permet de classer les meilleures solutions pour l'objectif à atteindre (dans notre cas, les solutions proches de la solution optimale auront une meilleure valeur de *fitness*). Lors de la deuxième étape, les meilleurs individus sont sélectionnés selon leur valeur de fitness et participent à la génération des nouvelles solutions qui constitueront la prochaine population. À noter que selon la stratégie adoptée, des solutions moins bonnes pourront être sélectionnées pour permettre de garder une certaine diversité et ne pas converger trop rapidement vers un optimum local. Les nouvelles solutions peuvent être générées par croisement et/ou mutation. Le croisement de deux solutions donne généralement deux nouvelles solutions possédant une partie de chacune des deux solutions parentes. La mutation d'une solution consiste à effectuer une transformation locale (changement de la valeur affectée à une ou plusieurs variables, comme présentée dans la section précédente). Les nouvelles solutions obtenues composent la nouvelle population. La figure 2.30 présente un exemple de fonctionnement d'une méta-heuristique évolutionnaire de base sur notre problème de la figure 2.23. À partir de la population initiale composée des solutions s_0, s_1, s_2 et s_3, celles-ci sont évaluées (ici dans cet exemple la fonction *fitness* leur affecte une valeur correspondante à leur coût) et les trois meilleures solutions (s_0, s_1, s_2 car $\mathcal{V}(s_0) \leq \mathcal{V}(s_2) < \mathcal{V}(s_1) < \mathcal{V}(s_3)$) sont sélectionnées. Deux nouvelles solutions sont obtenues à partir des solutions s_0 et s_1 par croisement des valeurs pour les variables x_1 et x_3 et une nouvelle solution est obtenue par mutation en modifiant la valeur pour la variable x_3 dans la solution s_2. La population *descendante* obtenue est ainsi composée de deux solutions meilleures (s_4 et s_6) que les solutions de la population précédente, ainsi qu'une solution moins bonne (s_5).

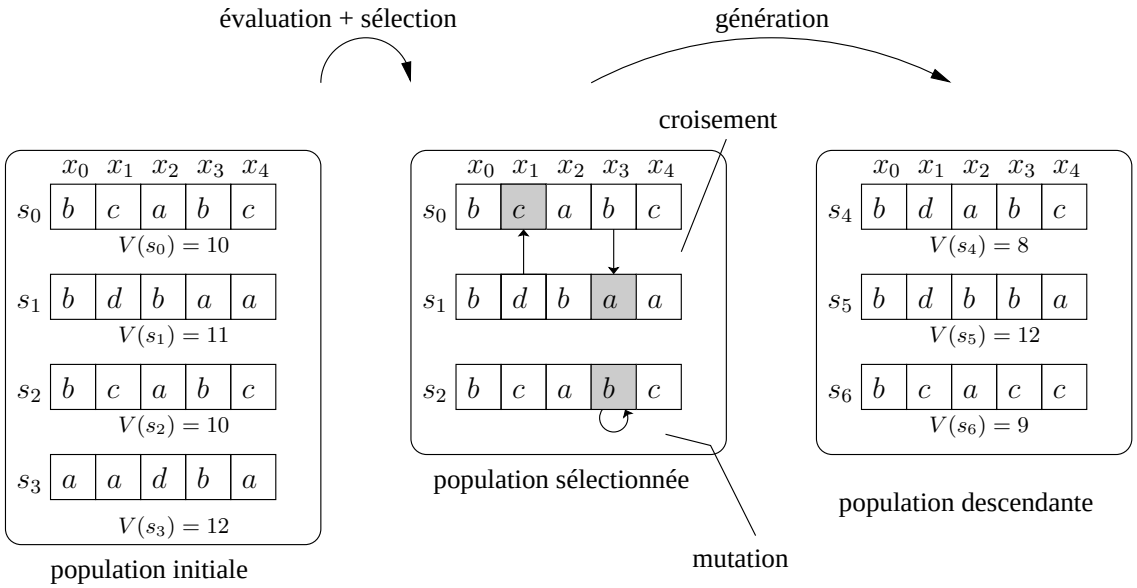


FIG. 2.30 – Exemple de la génération d'une nouvelle population pour le problème de la figure 2.23 par une approche évolutionnaire classique de type génétique.

L'idée proposée par les algorithmes génétiques, et plus particulièrement la considération simulta-

née de plusieurs solutions, est intéressante pour le cadre WCSP. Cependant, leur mise en œuvre et leur implémentation (ainsi que le paramétrage !) restent difficiles. De plus, comme cela a été soulevé dans [Neveu et Trombettoni, 2004], des problèmes très structurés comme les WCSP ne sont pas vraiment adaptés à des opérations de croisement qui peuvent également empêcher l'exploitation de propriétés incrémentales et donc impacter les performances. D'autres méta-heuristiques ont donc été proposées dans ce sens, plus faciles à mettre en œuvre et plus efficaces. Ces méthodes sont basées sur l'algorithme à populations *GWW* (*Go With the Winners*) [Aldous et Vazirani, 1994, Dimitriou et Impagliazzo, 1996]. C'est le cas notamment de la méthode d'hybridation *GWW-idw* (*Go With the Winners - Intensification and Diversification Walk*) [Neveu et Trombettoni, 2004] qui propose de remplacer la marche aléatoire introduite dans *GWW* par de la recherche locale par intensification et diversification. Cette méthode consiste à considérer une population d'individus et ne pas faire de croisements entre les meilleurs individus, mais leur faire faire une recherche locale. En fait, seules les mutations effectuées dans les algorithmes évolutionnaires classiques vont être effectuées. Considérant un seuil initial et après un certain nombre d'itérations de recherche locale pour chacun des individus, seuls les nouveaux individus générés ayant un coût inférieur au seuil (dans le cadre d'une fonction d'objectif de minimisation) sont conservés et les autres sont "redistribués" : ils reçoivent une copie de l'un des meilleurs individus choisi aléatoirement qui est ainsi représenté plusieurs fois (au moins deux fois) dans la nouvelle population. Afin de se diriger vers les meilleures solutions possibles, le seuil est décrémenté lors de chaque nouvelle itération afin de se rapprocher de la solution de coût optimal.

La figure 2.31 présente un exemple de fonctionnement de l'approche *GWW-idw* sur notre problème de la figure 2.23. À partir de la population initiale composée des solutions s_{00}, s_{10} et s_{20} et d'un seuil initial égal à 10, chacune de ces solutions va tenter d'être améliorée par de la recherche locale avec possibilité d'utiliser des techniques différentes pour chacune d'entre elles. Dans cet exemple, nous considérons une recherche locale avec seuil de tolérance à partir de la solution s_{00}, une approche *VNS* à partir de la solution s_{10} et une approche standard à partir de la solution s_{20}. Après une génération de l'approche *GWW-idw* (correspondant à trois itérations pour chacune des techniques de recherche locale utilisées), les meilleures solutions trouvées sont bien évidemment les mêmes que celles trouvées dans la section précédente pour des recherches locales réalisées indépendamment. Les solutions dépassant le seuil, c'est à dire avec un coût supérieur à 10 sont supprimées. Lors de la première génération aucune solution n'est supprimée. Cependant, lors de la deuxième génération et suite à la valeur du seuil décrémentée de 1, la solution s_{22} de coût égal à 9 atteint la valeur du seuil et elle est alors redistribuée : elle va recevoir le contenu de l'une des solutions encore acceptées, en l'occurrence ici la solution s_{22} va recevoir le contenu de la solution s_{12}.

Nous venons donc de présenter quelques méthodes incomplètes, à savoir des méta-heuristiques de recherche locale et des méta-heuristiques évolutionnaires, appliquées au cadre WCSP. Ces méthodes sont proposées par la bibliothèque INCOP (incorporée dans le solveur $ToulBar2$) que nous avons utilisée pour les comparaisons expérimentales dans notre seconde contribution.

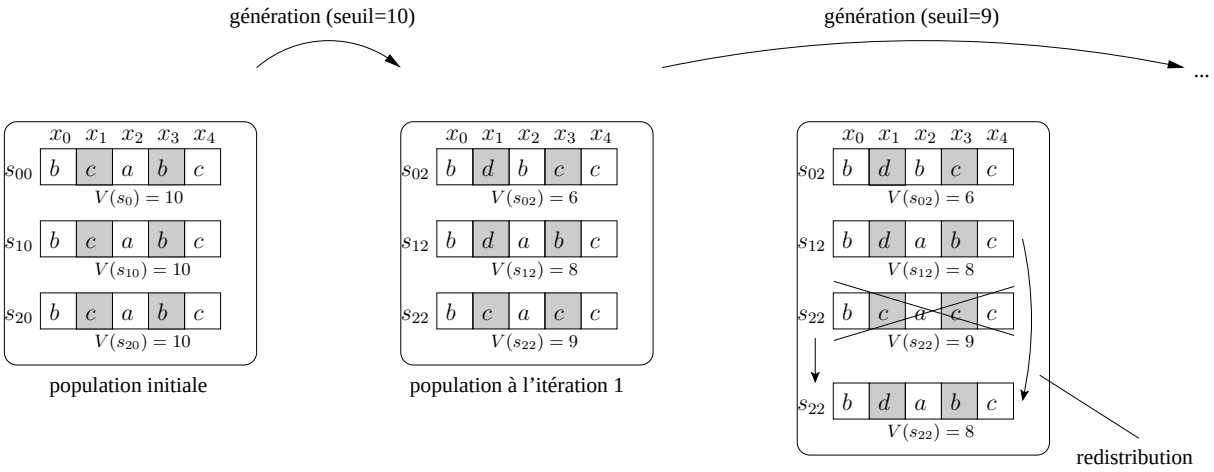


FIG. 2.31 – Exemple de la génération d'une nouvelle population pour le problème de la figure 2.23 par l'approche *GWW-idw*.

Deuxième partie

Contributions

Chapitre 3

Un algorithme de filtrage pour les contraintes tables souples de grande arité

Sommaire

Nous présentons dans cette section les travaux constituant notre première contribution. Ce travail a fait l'objet d'une publication dans [Lecoutre *et al.*, 2012] et d'une extension en cours de soumission pour le cas où le coût par défaut n'est ni 0, ni k.

3.1 Problématique

Comme nous l'avons vu lors du chapitre 2, le problème de satisfaction de contraintes pondérées (WCSP) représente un cadre d'optimisation. Dans un réseau de contraintes pondérées, une contrainte souple est définie par une fonction de coût qui associe un degré de violation, appelé coût, à toute instanciation de la portée de cette contrainte. En utilisant l'addition bornée $\oplus$, ces coûts peuvent être combinés afin d'obtenir le coût global de n'importe quelle instanciation. Trouver une instanciation complète de coût minimal est un problème `NP-Difficile`.

La propriété d'arc-cohérence AC, introduite à l'origine pour le problème de satisfaction de contraintes (CSP), a été étudiée plus tard dans le contexte des WCSP. Des développements de plus en plus sophistiqués dans le but de parvenir à la meilleure forme d'arc-cohérence souple ont naturellement été proposés au fil des années (EDAC*, VAC, OSAC, etc.). Le transfert de coûts, principe sur lequel sont basés les algorithmes établissant ces propriétés, préserve la sémantique des réseaux de contraintes souples tout en concentrant les coûts sur les valeurs des domaines (contraintes unaires) et sur un coût global (contrainte 0-aire). Les algorithmes à transferts de coûts s'avèrent particulièrement efficaces pour résoudre des instances de problèmes du monde réel, en particulier lorsque les contraintes souples sont binaires ou ternaires (le site `http://costfunction.org` propose de nombreux problèmes de ce type).

Cependant, pour les contraintes souples de grande arité, c'est à dire dont la portée contient bien au delà de trois variables, un problème d'ordre combinatoire peut se présenter lors de l'utilisation des transferts de coûts. En effet, pour effectuer ces opérations, un algorithme général qui n'a pas été optimisé pour le cas des contraintes tables souples et qui n'a pas été pensé pour gérer de manière efficace les tuples implicites lorsque le coût par défaut est différent de 0 et de k, va devoir nécessairement parcourir le produit cartésien des domaines pour les variables impliquées afin de déterminer les coûts minimaux. Ce nombre de tuples à parcourir est exponentiel en l'arité de la contrainte.

Dans la section 2.3.2, nous avions introduit la cohérence locale souple AC* pour les réseaux binaires. Pour les réseaux non binaires, l'arc-cohérence généralisée (GAC) du cadre CSP a également été adaptée au cadre WCSP : il s'agit de l'arc-cohérence généralisée pour le cadre VCSP [Cooper et Schiex, 2004, Cooper *et al.*, 2010].

Définition 57 (Cohérence d'arc généralisée pour le cadre VCSP [Cooper *et al.*, 2010]) *Un VCSP $(\mathcal{X},\mathcal{C},\mathcal{V})$ est arc-cohérent généralisé si pour toute contrainte $c_S \in \mathcal{C}$ telle que $|S| > 1$, nous avons :*
- $\forall \tau \in l(S), c_S(\tau) = k$ *si* $c_\emptyset \oplus \bigoplus_{x \in S} c_x(\tau[x]) \oplus c_S(\tau) = k$
- $\forall (x,a)$ *de* $c_S, \exists \tau \in l(S) \mid \tau[x] = a \wedge c_S(\tau) = 0$

La propriété de cohérence d'arc généralisée que nous exploitons dans cette contribution, appelée GAC*, est plus légère que celle proposée dans [Cooper et Schiex, 2004, Cooper *et al.*, 2010]. En effet, elle n'exige pas que les tuples τ, pour lesquels le coût étendu (définition 42) au sein d'une contrainte c_S est égal à k, soient directement interdits dans cette contrainte, c'est à dire $c_S(\tau) = k$.

Définition 58 (Cohérence d'arc généralisée GAC*) *Soit c_S une contrainte d'arité $|S| \geq 2$. Une valeur (x,a) de c_S est arc-cohérente généralisée (GAC*) ou GAC*-cohérente dans c_S si et seulement si $\exists \tau \in l(S) \mid \tau[x] = a \wedge c_S(\tau) = 0$. Le tuple τ est alors appelé support de la valeur (x,a) dans c_S. Une variable x de S est arc-cohérente généralisée (GAC*) dans c_S si et seulement si x est nœud-cohérente (NC*) et si pour toute valeur $a \in dom(x)$, (x,a) est arc-cohérente généralisée dans c_S. Une variable x est arc-cohérente généralisée (GAC*) si et seulement si elle est arc-cohérente généralisée (GAC*) dans toutes les contraintes portant sur x. Un réseau est arc-cohérent généralisé (GAC*) si toutes les variables appartenant à ce réseau sont arc-cohérentes généralisées (GAC*).*

Pour établir l'arc-cohérence généralisée GAC* sur les réseaux de contraintes souples n-aires, nous présentons l'algorithme 26 obtenu en étendant l'algorithme AC3* présenté pour le cadre binaire dans [Larrosa et Schiex, 2004]. Il est important de préciser qu'il s'agit d'une extension naturelle mais non publiée à notre connaissance. Cependant, notons qu'une version très similaire pour le cas n-aire a été proposée dans [Lee et Leung, 2009]. Par rapport à l'algorithme AC3* proposé pour le cadre binaire, les différences majeures dans GAC3* correspondent à la recherche de tuples dans le produit cartésien des domaines des variables appartenant au scope d'une contrainte, à savoir $l(S)$ avec S contenant plus que deux variables, dans l'algorithme trouverSupportSimple(), Algorithme 27, aux lignes 3 et 5 (dans le

corps de projection(), Algorithme 12), ainsi que le parcours des variables dans le scope de la contrainte à la ligne 7. À noter que les algorithmes projection(), projectionUnaire() et filtrerDomaine() utilisés correspondent respectivement aux algorithmes 12, 14 et 17 introduits dans la section 2.3.

Algorithm 26: GAC3* ($P = (\mathcal{X}, \mathcal{C}, k)$: WCSP) : Booléen

```
1  Q ← X
2  tant que Q ≠ ∅ faire
3      Choisir et éliminer y de Q
4      transfertEffectue ← faux
5      pour chaque contrainte c_S telle que |S| ≥ 2 ∧ y ∈ S faire
6          pour chaque variable x ∈ S | x ≠ y faire
7              transfertEffectue ← transfertEffectue ∨ trouverSupportSimple(c_S,x)
8              si filtrerDomaine(x) alors
9                  si dom(x) = ∅ alors Retourner faux
10                 Q ← Q ∪ {x}
11     si transfertEffectue alors
12         pour chaque variable x ∈ X faire
13             si filtrerDomaine(x) alors
14                 si dom(x) = ∅ alors Retourner faux
15                 Q ← Q ∪ {x}
16 Retourner vrai
```

Algorithm 27: trouverSupportSimple (c_S : contrainte, x : variable) : Booléen

```
1  transfertEffectue ← faux
2  pour chaque valeur a ∈ dom(x) faire
3      α = min{c_S(τ) | τ ∈ l(S) ∧ τ[x] = a}
4      si c_x(a) = 0 ∧ α > 0 alors transfertEffectue ← vrai
5      projection(c_S,x,a,α)
6  α = c_x(argmin_{a∈dom(x)}{c_x(a)})
7  projectionUnaire(x, α)
8  Retourner transfertEffectue
```

Il est intéressant de noter que la complexité temporelle de l'algorithme AC3* proposé pour le cas binaire dans [Larrosa et Schiex, 2004] est $O(n^2d^2 + ed^3)$ avec n le nombre de variables dans le réseau, e le nombre de contraintes et d la taille maximale parmi les domaines des variables.

Proposition 1 *Dans le cadre de réseaux n-aires, la complexité en temps de GAC3*, Algorithme 26, est $O(n^2d^2 + der(r(d^r + d)))$, avec n le nombre de variables dans le réseau, e le nombre de contraintes, d la taille maximale parmi les domaines des variables et r l'arité maximale parmi les contraintes.*

Preuve Dans le cadre de réseaux n-aires, les algorithmes projection() et trouverSupportSimple() ont respectivement des complexités en temps $O(d^{r-1})$ et $O(d^r)$. En effet, projection() peut nécessiter (pour une valeur (x, a)) de parcourir tous les tuples $\tau \in l(S)$ tels que $\tau[x] = a$ pour la contrainte c_S, c'est à

dire d^{r-1} tuples (une variable de S est fixée lors de la recherche donc uniquement $r-1$ variables à considérer). Dans l'algorithme trouverSupportSimple(), projection() et $min\{c_S(\tau) \mid \tau \in l(S) \wedge \tau[x] = a\}$ (en $O(d^{r-1})$ car même parcours de tuples que pour projection()) peuvent être exécutées d fois, ce qui donne une complexité en temps $O(d^r)$. La boucle principale *tant que* de GAC3* peut être exécutée nd fois (le domaine des n variables peut être modifié d fois). De ce fait, la complexité en temps des lignes 12 à 15 de GAC3* est $O(n^2d^2)$: en effet, les lignes 12 à 13 peuvent appeler n fois l'algorithme filtrerDomaine() (complexité $O(d)$) à chaque exécution. La complexité en temps des lignes 2 à 5 de GAC3* est $O(der)$ car les e contraintes impliquent au plus r variables dans leur scope (et non pas les n variables) et le domaine de ces r variables peut être modifié d fois (c'est pourquoi la ligne 5 est exécutée $\boldsymbol{erd}$ fois au lieu de $\boldsymbol{end}$). La complexité en temps de la ligne 7 (appel de trouverSupportSimple()) étant de $O(d^r)$, celle des lignes 6 à 10 de GAC3* est donc $O(r(d^r + d))$, nous obtenons une complexité en temps de $O(der(r(d^r + d)))$ pour les lignes 2 à 10. Par conséquent, la complexité totale de GAC3* est $O(n^2d^2 + der(r(d^r + d)))$[17]. □

Nous voyons donc bien que dans le cas n-aire, la complexité en temps donnée par la proposition 1 est problématique et correspond au problème d'ordre combinatoire annoncé précédemment. À noter que les versions généralisées des algorithmes établissant les cohérences locales souples FDAC* et EDAC* n'ont pas été présentées ici mais, comme précisé dans la section 2.3.4, ces deux cohérences impliquent la cohérence locale souple AC*. De ce fait, on en déduit logiquement que la complexité en temps pour établir GAC* représente un minorant pour les complexités en temps pour établir ces deux cohérences. Par analogie, on peut en déduire que la complexité en temps de GAC3*, à savoir $O(n^2d^2 + der(r(d^r + d)))$, représente un minorant de la complexité en temps des algorithmes (G)FDAC* et (G)EDAC*. Notre solveur $AbsCon$ implante l'algorithme GAC3*. Pour les contraintes de grande arité, on pourrait imaginer adapter au cadre des contraintes souples des techniques plus efficaces de parcours présentées dans la section 1.2.3, à savoir les algorithmes *GAC-allowed* et *GAC-valid+allowed*. Cependant, nous avons décidé de nous orienter directement vers l'un des algorithmes les plus efficaces pour établir l'arc-cohérence généralisée sur des contraintes tables positives non binaires, à savoir la technique de réduction tabulaire simple (STR pour Simple Tabular Reduction) présentée également dans la section 1.2.3.

Pour cette première contribution, nous nous intéressons aux contraintes tables souples et nous intégrons la réduction tabulaire simple STR, introduite dans le cadre CSP et adaptée au cadre WCSP, pour établir la cohérence GAC*. Fondamentalement, dès que des valeurs sont supprimées des domaines durant la propagation ou la recherche, tous les tuples qui deviennent invalides sont supprimés des contraintes tables. Cela nous permet alors d'identifier les valeurs qui ne sont plus GAC*-cohérentes. Ce qui est intéressant, c'est que parce tous les tuples valides des tables sont parcourus, il est facile et peu coûteux de calculer les coûts minimaux associés aux valeurs. Ceci est particulièrement utile pour effectuer de manière efficace les opérations de projection requises pour établir GAC*. De plus, nous proposons également dans cette contribution d'incorporer cette technique dans un contexte sans transfert de coûts, c'est à dire dans l'algorithme PFC-MPRDAC, pour calculer efficacement les coûts minimaux pour les valeurs et qui sont utilisés pour le calcul des bornes.

3.2 Méthodes de l'état de l'art

Une première méthode pour traiter les contraintes souples de grande arité, c'est à dire d'arité au moins égale à 4 ou 5, en exploitant les cohérences efficaces pour les contraintes binaires et ternaires est de retarder la propagation (des coûts) en attendant qu'un nombre suffisant de variables soient assignées. Ainsi, les contraintes n-aires se ramènent (d'une certaine manière) à des contraintes binaires ou ternaires et les transferts de coût redeviennent une technique efficace. Malheureusement, retarder la propagation

[17] Dans la section 3.8.2, cette complexité sera comparée avec la complexité en temps de l'approche que nous proposons

des coûts réduit considérablement la capacité de filtrage des algorithmes en début de recherche. Une seconde méthode consiste à concevoir des adaptations des algorithmes de cohérence d'arc souple pour certaines familles de contraintes (globales) souples. C'est l'approche proposée dans [Lee et Leung, 2009, Lee et Leung, 2010] où le concept de contraintes saines pour la projection est introduit. Une troisième approche [Favier *et al.*, 2011] consiste à décomposer les contraintes souples en contraintes souples de plus faible arité. Les décompositions de contraintes globales souples [Allouche *et al.*, 2012] sont également envisageables. Malheureusement, toutes les contraintes souples ne peuvent pas être décomposées. Enfin, il existe également une approche classique consistant à ne pas effectuer de transferts de coûts. Il s'agit de l'approche PFC-MPRDAC présentée dans la section 2.3.5 qui est un algorithme de séparation et évaluation permettant de calculer des bornes inférieures à chaque nœud de l'arbre de recherche.

3.3 Description simple de l'algorithme

Dans cette section, nous décrivons de manière simple les parties principales de notre algorithme GAC*-WSTR décrit plus formellement dans les sections 3.4 et 3.5. L'objectif ici est de permettre une compréhension globale de l'algorithme et faciliter ainsi la description plus détaillée dans les sections suivantes.

Pour rappel, une contrainte table souple contient une liste de tuples avec leurs coûts associés. Ces tuples sont appelés les tuples explicites. Chaque tuple qui n'est pas dans cette liste de tuples de la contrainte est associé à un coût par défaut (`defaultCost`) qui est défini pour la contrainte. Ces tuples sont appelés des tuples implicites. L'objectif principal de l'algorithme est de maintenir une liste courante de tuples explicites valides. Pour cela, nous utilisons la même technique que celle présentée dans STR dans la section 1.2.3. La liste des tuples explicites est alors séparée en deux parties. La première partie, qui s'étend de l'indice 1 à l'indice `currentLimit` contient les tuples qui sont couramment valides. La deuxième partie, qui s'étend de `currentLimit` $+ 1$ jusqu'à la fin de la liste contient les tuples qui sont couramment invalides. Initialement, tous les tuples sont valides et `currentLimit` correspond au nombre de tuples explicites. Considérant une contrainte, quand le domaine associé à une variable de la portée de cette contrainte est réduit, l'algorithme vérifie si tous les tuples présents entre les indices 1 et `currentLimit` sont toujours valides. Lorsqu'un tuple devient invalide, il est simplement échangé avec le dernier tuple valide présent à l'indice `currentLimit` et la valeur `currentLimit` est décrémentée. Dans le cadre d'un retour en arrière, il suffit de restaurer la valeur précédente de `currentLimit` pour considérer à nouveau ce tuple comme valide. Afin d'éviter des opérations de copies pouvant s'avérer coûteuses, les tuples ne sont pas réellement déplacés physiquement. Un tableau de références de tuples (appelé `position`) est introduit. Initialement, ce tableau contient les indices ordonnés allant de 1 au nombre de tuples explicites. Quand un tuple devient invalide, son indice est simplement échangé avec l'indice du dernier tuple valide dans `position`, soit une opération très peu coûteuse car en temps constant quelle que soit l'arité de la contrainte. La figure 3.1 illustre ce fonctionnement à la base de STR.

Le deuxième objectif de GAC*-WSTR est d'identifier les transferts de coûts qui vont pouvoir être effectués sur la contrainte table souple. Pour identifier les opérations de projection pouvant être effectuées sur une contrainte c_S, nous devons identifier pour chaque valeur (x, a), telle que $x \in S, a \in dom(x)$ et x non assignée, le coût minimal parmi les tuples (à la fois explicites et implicites) contenant (x, a). Quand ce coût minimal est 0, aucun transfert de coût n'est possible. Sinon, une opération de projection est effectuée. Calculer le coût minimal d'une valeur en considérant les tuples explicites couramment valides est facile à faire. En fait, le parcours de la table requis pour vérifier la validité des tuples va servir également à initialiser un tableau `minCosts` contenant le coût minimal de chaque valeur à travers les tuples valides. Prendre en compte également les tuples implicites pour calculer ce coût minimal peut

s'avérer difficile, en fonction de la valeur du coût par défaut.

Coût par défaut k : quand $\texttt{defaultCost} = k$, les tuples implicites ont le coût interdit k et ils n'ont donc pas d'influence sur le calcul du coût minimal car $\forall \alpha \in [0, \ldots, k[, min(k, \alpha) = \alpha$. Par conséquent, le coût minimal des tuples (explicites ou implicites) contenant (x, a) est juste égal au coût minimal parmi les tuples *explicites* contenant (x, a). Puisque aucune opération de projection ne peut être effectuée depuis des tuples implicites associés à $\texttt{defaultCost} = k$, nous n'avons pas besoin de garder un historique des opérations de projection réalisées antérieurement depuis les tuples implicites.

Coût par défaut 0 : quand $\texttt{defaultCost} = 0$, il est alors suffisant de déterminer s'il existe un tuple implicite valide contenant (x, a). Si un tel tuple implicite existe, son coût est alors égal à 0 et donc le coût minimal parmi les tuples contenant (x, a) est 0 et aucune projection de coût ne peut être effectuée. Si au contraire il n'existe pas de tel tuple implicite, alors le coût minimal des tuples (explicites ou implicites) contenant (x, a) est juste égal au coût minimal parmi les tuples *explicites* contenant (x, a). Déterminer s'il existe un tuple implicite valide contenant (x, a) est réalisé en comptant le nombre de tuples explicites valides contenant (x, a) ($\texttt{nbExplicitValidTuples}[x][a]$ dans l'algorithme) et en comparant ce nombre avec le nombre total de tuples valides contenant (x, a) ($\texttt{nbValidTuples}[x]$ dans l'algorithme). Si ces deux nombres sont différents, c'est qu'il existe un tuple implicite contenant (x, a). Puisque aucune opération de projection ne peut être effectuée depuis des tuples implicites associés à $\texttt{defaultCost} = 0$, nous n'avons pas besoin là non plus de garder un historique des opérations de projection réalisées antérieurement depuis les tuples implicites.

Coût par défaut intermédiaire : dans le cas intermédiaire, autrement dit quand $\texttt{defaultCost} \neq 0$ et $\texttt{defaultCost} \neq k$, nous devons garder une trace à la fois des opérations de projections antérieures qui ont modifié le coût des tuples implicites et aussi identifier un tuple implicite ayant le plus petit coût afin de déterminer le coût minimal parmi les tuples contenant (x, a). Il faut souligner que dans ce cas intermédiaire, les tuples implicites peuvent ne plus avoir tous le même coût. Par exemple, avec un coût par défaut égal à 10, une fois que la projection d'un coût de 3 sur (x, a) et la projection d'un coût de 2 sur (y, b) ont été effectuées (et aucune autre projection effectuée), les tuples implicites τ tels que $\tau[x] = a \wedge \tau[y] = b$ ont alors un coût de 5, les tuples implicites tels que $\tau[x] = a \wedge \tau[y] \neq b$ ont un coût de 7, les tuples implicites tels que $\tau[x] \neq a \wedge \tau[y] = b$ ont un coût de 8 et les autres tuples implicites ont toujours un coût égal à 10.

Une manière pour garder une trace des projections effectuées serait de transformer les tuples implicites en une forme de tuples explicites décrits par un produit de sous-domaines. Par exemple, après une projection d'un coût de 3 sur (x, a) pour une table avec un coût par défaut égal à 10, nous pourrions diviser l'ensemble des tuples implicites en deux parties : une partie avec $\tau[x] = a$ et un coût égal à 7, l'autre avec $\tau[x] \neq a$ et un coût égal à 10. Évidemment, ce principe génère un nombre exponentiel d'ensembles de tuples et ne peut pas être utilisé en pratique.

Au lieu de cela, nous avons développé une méthode polynomiale pour à la fois garder une trace des opérations de projections effectuées et identifier un tuple implicite de coût minimal contenant une valeur (x, a). Nous conservons une trace des projections en additionnant les coûts transférés sur une valeur (x, a) dans une variable $\texttt{deltas}[x][a]$. Par exemple, après une projection d'un coût 3 sur (x, a), une projection d'un coût 2 sur (y, b) et d'une autre projection d'un coût 1 sur (x, a), nous avons $\texttt{deltas}[x][a] = 4$ et $\texttt{deltas}[y][b] = 2$. Ces deltas nous évitent de modifier le coût des tuples de la contrainte, mais évidemment nous devons les prendre en compte en soustrayant les deltas à la fois du coût des tuples implicites et des tuples explicites. Par conséquent, le coût d'un tuple τ (qu'il soit explicite ou implicite) sur une portée S avec un coût initial c correspond à $c \ominus \bigoplus_{x \in S} \texttt{deltas}[x][\tau[x]]$.

Pour identifier le tuple implicite de coût minimal, nous énumérons tous les tuples valides possibles dans un ordre de coût *croissant* et on vérifie si chaque tuple énuméré est explicite ou implicite en le cherchant dans l'ensemble des tuples explicites. S'il n'est pas trouvé, alors il s'agit d'un tuple implicite de coût minimal, sinon nous générons le tuple suivant selon l'ordre de coût *croissant* et nous recommençons jusqu'à ce nous trouvions le tuple implicite de coût minimal. Puisque le nombre de tuples explicites est polynomial en fonction de la taille de la contrainte, tout comme la génération du tuple suivant selon un ordre *croissant* du coût et la vérification pour un tuple explicite, cette identification d'un tuple implicite de coût minimal est polynomial en fonction de la taille de la contrainte.

Pour énumérer tous les tuples valides possibles selon un ordre de coût *croissant*, nous commençons par trier les valeurs de chaque domaine selon un ordre de coût croissant (ou en d'autres termes, selon un ordre décroissant des deltas) et nous générons le tuple de coût minimal en choisissant la valeur de coût minimal dans chaque domaine. Pour générer le tuple suivant selon l'ordre de coût croissant, nous générons les r successeurs du tuple courant en remplaçant pour chaque variable $x \in S$ sa valeur courante par la valeur suivante dans le domaine de x trié par ordre de coût croissant. Ces r successeurs sont insérés dans une queue de priorité qui est triée par ordre de coût croissant et en cas d'égalité, par ordre lexicographique croissant des tuples. Le tuple suivant selon l'ordre de coût croissant correspond au tuple de coût minimal présent dans la queue de priorité. Le lemme 4 et la propriété 2 prouvent que cette méthode est correcte et polynomiale.

Une fois qu'un tuple implicite de coût minimal et qu'un tuple explicite de coût minimal contenant (x, a) ont été identifiés, il est alors facile d'obtenir le coût minimal parmi les tuples contenant (x, a) et d'effectuer les opérations de projections possibles.

Pour résumer, il est donc possible quelle que soit la valeur du coût par défaut d'identifier en temps polynomial les projections de coût possibles et de les réaliser. Une fois ces opérations effectuées, il est facile d'effectuer les opérations de projections unaires possibles.

Afin d'accélérer l'exécution de l'algorithme, nous nous concentrons sur les variables présentes dans un ensemble S^{sup} pour lesquelles une opération de projection est potentiellement possible. En d'autres termes, les variables dans S^{sup} sont celles pour lesquelles nous n'avons pas un support (d'où le nom S^{sup}) pour chaque valeur, un support pour une valeur correspondant à un tuple de coût 0 et contenant cette valeur. De la même façon, les tests de validité sont concentrés sur un ensemble de variables S^{val} (val pour validité) qui correspondent aux variables pour lesquelles le domaine a été modifié depuis le dernier appel de GAC*-WSTR.

3.4 Structures de données

Dans cette section, nous commençons tout d'abord par décrire les structures de données de base utilisées pour manipuler les contraintes tables souples. Ensuite, nous discutons de la gestion des tuples implicites (en considérant les trois cas généraux de coût par défaut). Pour finir, nous y décrivons une représentation orientée objet des contraintes table souples.

3.4.1 Structures de base pour les contraintes tables souples

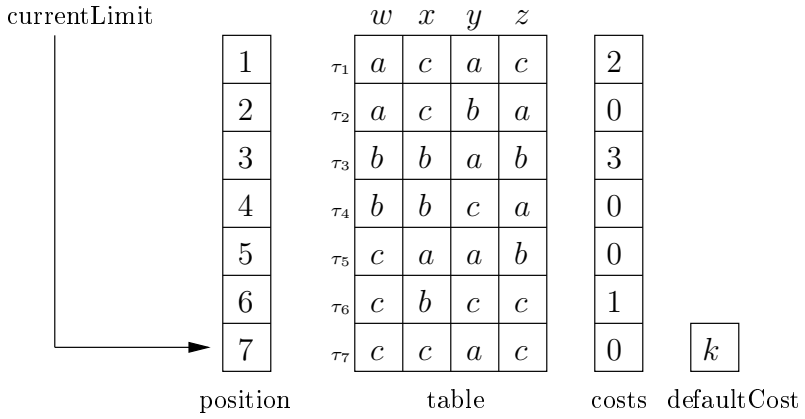
Pour rappel, une contrainte table souple c_S est une contrainte définie par un tableau c_S.`table` de t tuples (construits sur S), un tableau c_S.`costs` de t entiers, et un entier c_S.`defaultCost`. Le i$^{\text{ième}}$ tuple dans c_S.`table` (avec $i \in 1..t$) se voit associer comme coût la i$^{\text{ième}}$ valeur dans c_S.`costs`. Un tuple implicite, c'est à dire un tuple qui n'est pas présent dans c_S.`table`, se voit associer comme coût la valeur c_S.`defaultCost`.

Une caractéristique importante de STR (section 1.2.3) est de maintenir de manière efficace la liste

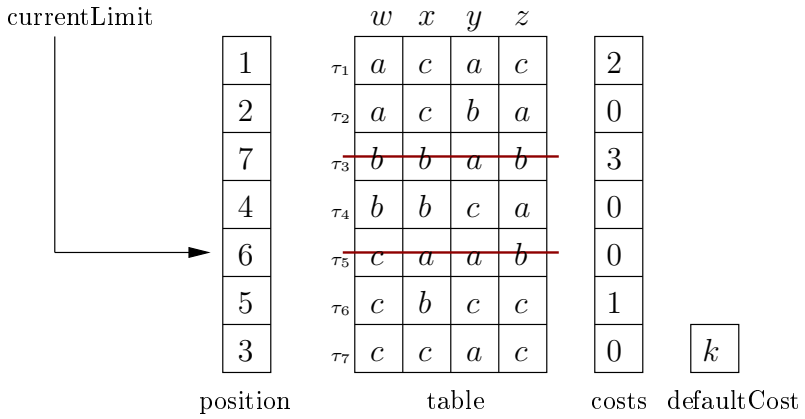
des tuples valides. Le principe est de diviser les tuples de chaque contrainte table c_S en deux ensembles. Le premier ensemble contient tous les tuples qui sont couramment valides : les tuples de cet ensemble constituent le contenu de la *table courante* de c_S. Un tuple présent dans la table courante d'une contrainte c_S est appelé un *tuple courant* de c_S. L'autre ensemble contient alors les tuples qui sont invalides et par conséquent supprimés aux différents niveaux de la recherche. Invalider un tuple et restaurer la liste des tuples valides lors d'un retour en arrière est particulièrement simple et efficace. Afin de simplifier la compréhension de notre algorithme, les structures de données utiles pour gérer les retours en arrière dans le cadre STR ne seront pas détaillées ici. Cependant, plus d'informations peuvent être obtenues dans la section 1.2.3 ou dans [Lecoutre, 2009, Lecoutre, 2011].

Pour une contrainte table (souple) c_S, les tableaux suivants permettent d'accéder aux différents ensembles disjoints de tuples valides et invalides dans c_S.`table` :

- c_S.`position` est un tableau de taille t qui offre un accès indirect aux tuples de c_S.`table`. À tout moment, les valeurs dans c_S.`position` représentent une permutation de $\{1, 2, \ldots, t\}$. Le i$^{\text{ième}}$ tuple de c_S est c_S.`table`$[c_S$.`position`$[i]]$, et son coût correspond à c_S.`costs`$[$`c`$_S$.`position`$[$`i`$]]$.
- c_S.`currentLimit` est la position du dernier tuple courant dans c_S.`table`. La table courante de `c`$_S$ est composée d'exactement c_S.`currentLimit` tuples. Les valeurs dans c_S.`position` aux indices allant de 1 à c_S.`currentLimit` correspondent aux positions des tuples courants de c_S.



(a) Contrainte table souple c_{wxyz}, avec initialement 7 tuples explicites valides τ_1, $\ldots$, τ_7.



(b) Contrainte table souple c_{wxyz} avec 5 tuples explicites valides (tuples τ_3 et τ_5 supprimés) après que la décision $z \neq b$ ait été prise.

FIG. 3.1 – Structures de base pour une contrainte table souple quaternaire c_{wxyz}.

La figure 3.1(a) illustre comment une contrainte table souple c_{wxyz} est représentée avec nos structures de données (nous supposons que $dom(w) = dom(x) = dom(y) = dom(z) = \{a, b, c\}$). Le tableau `table` est composé de 7 tuples (triés par ordre lexicographique de τ_1 à τ_7). Le coût associé à chaque tuple est donné par le tableau `costs`. Le tableau `position` fournit un accès indirect aux tuples. Le dernier tuple valide de la table est marqué par le pointeur `currentLimit`. Initialement, tous les tuples de la table sont valides et la table courante est composé d'exactement `currentLimit` $= 7$ tuples. Il est intéressant de noter aussi que chaque valeur possède un support dans c_{wxyz}. Supposons maintenant que la décision $z \neq b$ est prise, la contrainte c_{wxyz} doit donc être filtrée : la figure 3.1(b) représente l'état de `position` après que les tuples τ_3 et τ_5 aient été supprimés parce qu'ils sont devenus invalides. Nous pouvons aussi observer que la valeur (x, a) ne possède plus de support, et que tous les tuples contenant la valeur (x, a) sont implicites et ont un coût égal à $k : a$ devra donc être supprimée du domaine de x.

Comme présenté pour STR2 (section 1.2.3), nous introduisons également deux ensembles de variables, appelés S^{sup} et S^{val} (pour traiter respectivement les opérations liées aux supports et aux tests de validité). D'une part, dès que toutes les valeurs du domaine d'une variable ont été détectées GAC*-cohérentes, il est inutile de continuer de chercher des supports pour les valeurs de cette variable. Nous introduisons alors l'ensemble S^{sup} composé des variables non assignées (appartenant à la portée de la contrainte) pour lesquelles le domaine contient au moins une valeur pour laquelle un support n'a pas encore été trouvé. Toutes les opérations principales de notre algorithme travailleront uniquement avec les variables présentes dans S^{sup}. Pour mettre à jour S^{sup}, nous utilisons un tableau pour compter le nombre c_S.`nbGacValues`$[x]$ de valeurs GAC*-cohérentes identifiées pour chaque variable x. Dès que c_S.`nbGacValues`$[x] = |dom(x)|$, x est supprimé de S^{sup}.

D'autre part, à la fin de l'invocation de GAC*-WSTR pour une contrainte c_S, nous savons que pour toute variable $x \in S$, tout tuple τ tel que $\tau[x] \notin dom(x)$ a été supprimé de la table courante de c_S. Si aucun retour en arrière n'est effectué et que $dom(x)$ ne change pas entre cette invocation et la suivante, alors au moment de la prochaine invocation, il est indéniable que $\tau[x] \in dom(x)$ pour tout tuple τ dans la table courante de c_S. Dans ce cas, il n'y a pas besoin de tester si $\tau[x] \in dom(x)$; on gagne alors en efficacité en omettant cette vérification. Nous implémentons cette optimisation grâce à l'ensemble S^{val}, qui correspond à l'ensemble des variables non assignées pour lesquelles le domaine a été réduit depuis l'invocation précédente de GAC*-WSTR. Pour mettre en place S^{val}, nous avons besoin d'enregistrer la taille du domaine de chaque variable $x \in S$ juste après l'exécution de GAC*-WSTR sur c_S : cette valeur est enregistrée dans c_S.`lastSize`$[x]$. Au début du prochain filtrage, S^{val} est initialisé à l'ensemble des variables x telles que $|dom(x)| \neq c_S$.`lastSize`$[x]$ tandis qu'au début du programme, c_S.`lastSize`$[x]$ est initialisé à $|dom^{init}(x)|$.

Pour établir GAC* sur une contrainte c_S donnée, nous avons besoin de calculer les coûts minimaux des valeurs dans c_S. Ceci peut être effectué à moindre coût en traversant la table courante de c_S. Nous avons juste besoin d'un tableau c_S.`minCosts` pour stocker ces coûts minimaux ; c_S.`minCosts`$[x][a]$ correspondra au coût minimal de (x, a) dans c_S. Pour finir, quand le coût par défaut de c_S est différent de k, il est utile de compter le nombre c_S.`nbExplicitValidTuples`$[x][a]$ de tuples explicites valides contenant une valeur (x, a), afin de déterminer si un tuple implicite valide existe. S'il est vérifié que c_S.`nbExplicitValidTuples`$[x][a]$ est différent du produit cartésien des tailles des domaines courants de chaque variable exceptée x (c'est à dire $\prod_{y \in S; y \neq x} |dom(y)|$), alors c'est qu'il existe un tuple implicite contenant (x, a).

Pour conclure cette sous-section, discutons brièvement de la manière dont les transferts de coûts peuvent être implémentés. Afin de garder l'historique des transferts de coûts effectués à partir des tuples implicites sans modifier la liste des tuples explicites, nous avons eu besoin d'adopter la solution proposée dans [Cooper et Schiex, 2004, Larrosa et Schiex, 2004], qui présente une complexité raisonnable de $O(|S|d)$. Le principe consiste à conserver, pour chaque contrainte c_S telle que $|S| \geq 2$, les valeurs originales de c_S.`costs` tout en enregistrant dans une structure auxiliaire appelée c_S.`deltas` le coût cumulé de

toutes les projections sur chacune des valeurs ($c_S.\texttt{deltas}[x][a]$ est la somme de tous les coûts transférés sur (x, a)). Le coût courant pour un tuple explicite τ à la position $index$ (dans notre représentation) est alors calculé de la manière suivante :

$$c_S(\tau) = c_S.\texttt{costs}[index] \ominus \bigoplus_{x \in S} c_S.\texttt{deltas}[x][\tau[x]] \tag{3.1}$$

Pour un tuple implicite τ, le coût courant est :

$$c_S(\tau) = c_S.\texttt{defaultCost} \ominus \bigoplus_{x \in S} c_S.\texttt{deltas}[x][\tau[x]] \tag{3.2}$$

La procédure que nous utiliserons pour réaliser les projections est définie par la fonction projectDelta. Elle modifie la structure de données $c_S.\texttt{deltas}$ au lieu de modifier $c_S.\texttt{costs}$.

Function projectDelta(c_S : soft constraint, x : variable, a : value, α : integer)

Require: $|S| \geq 2 \wedge x \in S \wedge a \in dom(x) \wedge 0 < \alpha \leq min\{c_S(\tau) \mid \tau \in l(S) \wedge \tau[x] = a\}$

1 $c_x(a) \leftarrow c_x(a) \oplus \alpha$;
2 $c_S.\texttt{deltas}[x][a] \leftarrow c_S.\texttt{deltas}[x][a] \oplus \alpha$

3.4.2 Gestion des tuples implicites

Lorsque le coût par défaut $c_S.\texttt{defaultCost}$ d'une contrainte table souple c_S est égal à 0 ou k, nous savons que gérer l'historique des projections est inutile pour les tuples implicites de c_S. Plus précisément, si d'une part $c_S.\texttt{defaultCost} = 0$, alors il est évident qu'aucune projection ne peut être effectuée depuis les tuples implicites puisque les coûts négatifs ne sont pas autorisés. Dans l'équation 3.2, nous avons $c_S.\texttt{defaultCost} = 0$ qui implique $c_S.\texttt{deltas}[x][\tau[x]] = 0, \forall x \in S$ (aucun transfert possible). D'autre part, si $c_S.\texttt{defaultCost} = k$, le coût de n'importe quel tuple implicite reste inchangé, et ce quelles que soient les opérations de projections réalisées. Dans l'équation 3.2, nous avons $c_S.\texttt{defaultCost} = k$ qui implique $c_S(\tau) = k$ puisque $k \ominus \beta = k$ lorsque $0 \leq \beta \leq k$.

Toutefois, si $0 < c_S.\texttt{defaultCost} < k$, appelé aussi coût par défaut *intermédiaire* ci-après, il est envisageable d'effectuer des opérations de projection en considérant tous les tuples, incluant les tuples implicites, tout en faisant attention aux coûts déjà projetés sur des valeurs. Pour toute valeur (x, a) d'une contrainte table souple c_S avec un coût par défaut intermédiaire, les questions que nous avons à nous poser sont les suivantes :

- comment le coût minimal des tuples (explicites ou implicites) contenant la valeur (x, a) peut-il être identifié ?
- ce coût minimal peut-il être identifié en temps polynomial ?

L'approche que nous proposons, prouvée plus tard comme polynomiale, consiste à exploiter les coûts des tuples implicites, si nécessaire, après avoir exploité les tuples explicites. Il est important de rappeler que le coût courant d'un tuple implicite τ est calculé à partir de $c_S.\texttt{defaultCost}$ et en considérant toutes les projections déjà réalisées et enregistrées dans la structure $c_S.\texttt{deltas}$ (voir l'équation 3.2). Considérant une valeur (x, a), notre objectif est alors de trouver le coût minimal de n'importe quelle instanciation $I \in l(S) \setminus c_S.\texttt{table}$ contenant cette valeur (c'est à dire telle que $I[x] = a$) et défini de la manière suivante :

$$c_S.\texttt{defaultCost} \tag{3.3}$$

$$\ominus c_S.\texttt{deltas}[x][a] \tag{3.4}$$

$$\ominus \bigoplus_{y \in S \setminus \{x\}} c_S.\texttt{deltas}[y][I[y]] \tag{3.5}$$

Compte tenu que (3.3) et (3.4) sont invariants ($c_S.\texttt{defaultCost}$ et $c_S.\texttt{deltas}[x][a]$ sont fixés), il est alors évident que pour obtenir un coût minimal, (3.5) doit être maximisé. En d'autres termes, un tuple implicite avec un coût minimal pour la valeur (x, a) correspond à un tuple implicite tel que la somme des deltas correspondant aux valeurs pour toutes les variables dans S, excepté pour x, soit maximale. Afin de trouver ce tuple, nous introduisons un tableau $c_S.\texttt{sortedDomains}$ qui fournit pour chaque variable x de S son domaine courant avec toutes les valeurs triées par ordre décroissant de leurs valeurs de delta. Plus précisément, pour chaque variable $x \in S$, $c_S.\texttt{sortedDomains}[x].first()$ renvoie la première valeur (dans le domaine trié) et $c_S.\texttt{sortedDomains}[x].succ(v)$ renvoie la valeur juste après la valeur v, avec nil voulant dire qu'il n'y en a plus.

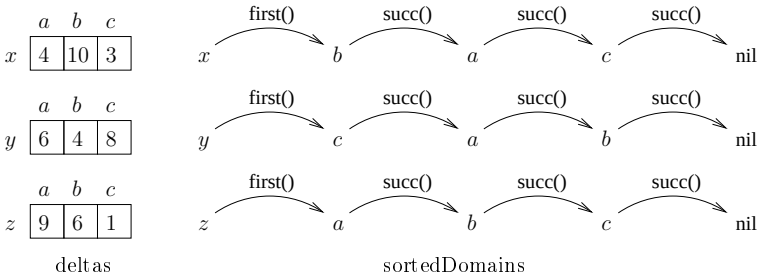


FIG. 3.2 – Les structures `deltas` et `sortedDomains` pour une contrainte souple c_{xyz}.

La figure 3.2 illustre cette structure de données pour une contrainte ternaire c_{xyz} impliquant les variables x, y, z avec les trois valeurs a, b, c dans leurs domaines respectifs. Sur la gauche, les tableaux contenant les valeurs de deltas sont affichés, et sur la droite les domaines triés sont présentés à travers l'utilisation des fonctions first() et succ(). Par exemple, on constate que la première valeur pour x dans $c_S.\texttt{sortedDomains}[x]$ est b, signifiant que $c_S.\texttt{sortedDomains}[x].first()$ renvoie b, la seconde valeur est a, signifiant que $c_S.\texttt{sortedDomains}[x].succ(b)$ renvoie a, et ainsi de suite. L'utilisation de cette structure pour traiter les tuples implicites sera expliqué dans la section 3.5.

3.4.3 Représentation orientée objet des contraintes tables souples

Dans nos algorithmes, les contraintes tables souples sont représentées par des objets de la classe SoftTableConstraint (illustrée par `Class SoftTableConstraint` dans ce manuscrit). Les attributs qui sont présents dans cette classe (la plupart d'entre eux ayant déjà été introduits précédemment) peuvent être classés comme suit :

- Les attributs basiques correspondant aux structures de données nécessaires pour représenter les contraintes table souples (portée, tuples avec coûts, coût par défaut, deltas) et pour enregistrer les coûts minimaux durant le processus d'inférence ;
- Les attributs STR correspondant aux structures de données utilisées par STR2 (section 1.2.3) ;
- Les attributs additionnels nécessaires pour traiter les tuples implicites lorsque le coût par défaut de la contrainte (objet) n'est pas égal à k.

La méthode principale dans la classe est findSupports() qui utilise des méthodes auxiliaires comme par exemple tableScan() lorsque la table courante doit être parcourue pour calculer les coûts minimaux.

Toutes ces méthodes seront décrites dans la section 3.6. Notons que du fait que nous utilisons une représentation orientée objet, nous utiliserons à certains endroits "this" pour référencer l'objet courant.

Class SoftTableConstraint

Fields :

/*Attributs basiques */
S : set of r variables ; // portée
table : array of t tuples ; // tuples explicites
costs : array of t integers ; // coût de chaque tuple explicite
defaultCost : integer ; // coût de chaque tuple implicite
deltas : array of $r \times d$ integers ; // coût cumulé des projections sur les valeurs
minCosts : array of $r \times d$ integers ; // coût minimal de chaque valeur

/*Attributs STR */
position : array of t integers ;
currentLimit : integer ;
lastSize : array of r integers ;
S^{val} : set of r variables ;
S^{sup} : set of r variables ;
nbGacValues : array of r integers ;

/*Attributs additionnels nécessaires pour traiter les tuples implicites */
nbValidTuples : array of r integers ;
nbExplicitValidTuples : array of $r \times d$ integers ;
sortedDomains : r sets of d values ; // selon valeurs décroissantes de deltas

Methods :

findSupports() ;
tableScan() ;
isValidTuple(tuple) ;
updateMinCost(variable,value,integer) ;
nbImplicitTuplesWith(variable,value) ;
handleZeroDefaultCost() ;
handleIntermediateDefaultCost() ;
searchImplicitTupleWithMinimalCost(variable,value) ;
buildSuccessors(variable,(tuple,integer)) ;

3.5 Algorithme GAC*-WSTR

Dans cette section, nous présentons une procédure de filtrage, nommée GAC*-WSTR, Algorithme 28, pour établir GAC* sur un réseau WCN P. Nous choisissons un schéma de propagation classique à gros grain, c'est-à-dire que la propagation des inférences est guidée par les variables qui sont stockées dans un ensemble Q. De base, lorsque le domaine d'une variable x est réduit, x est placée dans Q afin de pouvoir propager cet évènement aux contraintes impliquant x. L'algorithme 28 peut être utilisé pour établir GAC* durant une phase de pré-traitement, ou pour maintenir cette propriété au cours d'une recherche avec retour en arrière. Le premier cas implique d'appeler GAC*-WSTR avec le second paramètre Q cor-

respondant à $vars(P)$, tandis que le second cas implique d'appeler GAC*-WSTR après qu'une décision de recherche de la forme $x = a$ ou $x \neq a$ ait été prise, avec le second paramètre Q correspondant à $\{x\}$.

Dans notre approche, toutes les projections unaires et les suppressions de valeurs sont effectuées dans le corps de l'algorithme 28. Pour réaliser efficacement ces opérations, une structure de type "map" appelée `touchedVariables` est introduite pour stocker uniquement les paires (x, α) composées d'une part d'une variable x pour laquelle un coût unaire a été modifié, d'autre part du coût unaire minimal α de c_x, c'est à dire $\alpha = min\{c_x(a) \mid a \in dom(x)\}$. L'intérêt de cette "map" est d'itérer facilement sur les variables pour lesquelles la fonction de coût unaire a été modifiée, et obtenir facilement leurs coûts unaires minimaux sans les recalculer. Comme toute structure "map", `touchedVariables` supporte les opérations suivantes :

- `touchedVariables`.$clear()$ vide la "map".
- **foreach** $(x, \alpha) \in$ `touchedVariables` itère sur toutes les paires "clé-valeur" de la "map".
- `touchedVariables`.$put(x, \alpha)$ stocke la paire (x, α) en remplaçant une éventuelle paire (x, β) déjà présente.

Afin d'éviter des instructions inutiles (boucles), nous introduisons :

- `maxUCosts`, qui est un tableau qui donne pour chaque variable x le coût unaire maximal de c_x. Nous avons `maxUCosts`$[x] = max\{c_x(a) \mid a \in dom(x)\}$.
- `globalMaxUCost`, qui est une variable qui donne une borne supérieure du coût unaire maximal pour toutes les variables du WCN P. Nous avons `globalMaxUCost` $\geq max\{$`maxUCosts`$[x] \mid x \in vars(P)\}$. Maintenir une valeur exacte, au lieu d'une borne supérieure, serait plus coûteux puisque cela nécessiterait de recalculer $max\{$`maxUCosts`$[x] \mid x \in vars(P)\}$ pour des variables déjà considérées.

L'algorithme 28, que nous décrivons maintenant, exploite trois invariants principaux. Tout d'abord, les coûts maximaux (c'est-à-dire les valeurs de `maxUCosts` et `globalMaxUCost`) sont actualisés au moment où GAC*-WSTR est invoqué. Ensuite, c_S.findSupports() nous permet de calculer les valeurs de c_S.`minCosts`, c'est-à-dire pour chaque valeur (x, a) de c_S, après l'exécution de c_S.findSupports(), c_S.`minCosts`$[x][a]$ est égal à $min\{c_S(\tau) \mid \tau \in l(S) \wedge \tau[x] = a\}$. Enfin, lorsqu'une contrainte unaire c_y est modifiée, (c'est-à-dire qu'au moins un coût unaire a été modifié) durant un appel c_S.findSupports(), il y a une paire (y, α_y) dans `touchedVariables` avec α_y correspondant au coût unaire minimal de c_y.

À chaque itération de la boucle principale de l'algorithme 28, une variable x est extraite de Q. Nous devons alors propager la réduction du domaine de x (la raison pour laquelle x était dans Q) sur toutes les contraintes impliquant x. Pour chacune de ces contraintes c_S (lignes 4-5), nous exécutons c_S.findSupports() afin de mettre à jour c_S.`minCosts` ainsi que `touchedVariables` (voir nos hypothèses précédentes). À la ligne 6, nous pouvons exploiter `touchedVariables` pour à la fois effectuer toutes les projections unaires possibles en appelant unaryProject, algorithme 14, puis pour mettre à jour `maxUCosts` et `globalMaxUCost` en appelant la fonction updateMaxUCostOf. À la ligne 10, nous avons la garantie que `globalMaxUCost` est une borne supérieure des coûts unaires (parce que après avoir exécuté la fonction updateMaxUCostOf, la valeur de `globalMaxUCost` pourrait être sur-estimée) et `maxUCosts` est actualisé. Ceci nous permet d'éviter de considérer toutes les variables ou des variables individuelles aux lignes 10 et 13 quand le coût $c_\emptyset$ ajouté à un coût maximal calculé est strictement inférieur à k. En effet, nous savons dans ce cas qu'aucune valeur ne peut être supprimée. Enfin, toute valeur pour laquelle le coût a nécessairement atteint k est supprimé par la fonction pruneVar. Quand le domaine d'une variable y est réduit, nous vérifions qu'un domaine vide (échec) n'est pas apparu (ligne 15) et nous mettons à jour Q (ligne 17). Les autres instructions permettent de mettre à jour les structures `maxUCosts` et `globalMaxUCost`, ainsi lorsque l'algorithme 28 se termine, ces structures contiennent les valeurs correctes. Ceci prouve que notre première hypothèse est valide, sous la condition que `maxUCosts` et `globalMaxUCost` soient correctement initialisées lors de la construction, et correctement mises à jour au cours d'un retour en arrière (non détaillé ici). Il est utile de rappeler que unassigned(P) référence

l'ensemble des variables qui ne sont pas explicitement assignées par l'algorithme de recherche.

Algorithm 28: GAC*-WSTR(P : WCN, Q : set of variables)

```
/*Algorithme à exécuter pour établir GAC* sur le WCN P                    */
1  while Q ≠ ∅ do
2      pick and delete x from Q ;
       // Nous effectuons les projections possibles sur les
          contraintes impliquant x
3      touchedVariables.clear() ;
4      foreach constraint c_S ∈ cons(P) | x ∈ S ∧ |S| > 1 do
5          c_S.findSupports() // mise à jour de c_S.minCosts et touchedVariables
       // Nous effectuons les projections unaires possibles
6      foreach pair (y, α) ∈ touchedVariables do
7          if α > 0 then
8              unaryProject(y, α) ;
9          updateMaxUCostOf(y) ;
       // Nous effectuons les suppressions de valeurs possibles
10     if c_∅ ⊕ globalMaxUCost = k then
11         globalMaxUCost ← 0 ;
12         foreach variable y ∈ unassigned(P) do
13             if c_∅ ⊕ maxUCosts[y] = k then
14                 pruneVar(y) ;
15                 if dom(y) = ∅ then
16                     throw FAILURE ;
17                 Q ← Q ∪ {y} ;
18                 updateMaxUCostOf(y) ;
19             globalMaxUCost ← max(globalMaxUCost, maxUCosts[y]) ;
```

3.6 Trouver des supports dans les contraintes tables souples

Dans cette section, nous décrivons comment des supports sont établis au sein des contraintes tables souples (avec une arité supérieure ou égale à 2). Tandis que STR nécessite simplement un unique scan (parcours) de table dans le cadre de contraintes tables dures, pour les contraintes tables souples nous avons à traiter potentiellement plusieurs itérations sur les tuples de la table. Ces itérations supplémentaires sont dûes aux opérations de transfert de coût comme nous allons le voir maintenant avec la méthode findSupports(), Méthode 1, qui établit des supports sur toute contrainte table souple avec n'importe quel coût par défaut. C'est la méthode principale de notre classe, les autres méthodes étant simplement des méthodes auxiliaires. Nous décrivons dans un premier temps la méthode 1 sans détailler les méthodes auxiliaires.

Les lignes 1-13 de findSupports() correspondent à l'initialisation requise des ensembles S^{val} et S^{sup}, ainsi que des tableaux `nbGacValues`, `minCosts` et `nbExplicitValidTuples`. L'ensemble S^{val} est initialisé pour contenir les variables pour lesquelles les domaines ont étés modifiés depuis le dernier appel à l'algorithme (pour la même contrainte). L'ensemble S^{sup} contient uniquement les variables non assi-

Function updateMaxUCostOf(x : variable)

```
/*Exécutée pour mettre à jour maxUCosts[x] et globalMaxUCost */
1 m ← 0 ;
2 foreach value a ∈ dom(x) do
3     m ← max(m, c_x(a)) ;
4 maxUCosts[x] ← m ;
5 globalMaxUCost ← max(globalMaxUCost, m) ;
```

Function pruneVar(x : Variable)

```
/*Exécutée pour supprimer des valeurs de x */
1 foreach value a ∈ dom(x) do
2     if c_∅ ⊕ c_x(a) = k then
3         remove a from dom(x) ;
```

gnées, ce qui correspond à `unassigned`(P), c'est-à-dire l'ensemble des variables du WCN P qui ne sont pas explicitement instanciées par l'algorithme de recherche. Les tableaux `nbGacValues` et `minCosts` sont initialisés aux lignes 9 et 11 : initialement, aucune valeur n'a été prouvée comme possédant un support et `minCosts` est initialisé au plus grand coût possible k. Lorsque le coût par défaut de la contrainte est différent de k, nous devons également initialiser le tableau `nbExplicitValidTuples` : le nombre de tuples explicites valides dans la table est mis à 0 pour chaque valeur.

Les ensembles S^{val} et S^{sup} sont utilisés par tableScan(), Méthode 2, détaillée plus tard, pour effectuer de manière efficace un scan de la table courante. Ces scans nous permettent de calculer pour chaque valeur (x, a) de la contrainte le coût minimal pour un tuple contenant (x, a). Un premier scan est effectué à la ligne 14 de la méthode findSupports(), qui nous permet de mettre à jour S^{sup}, en conservant uniquement les variables pour lesquelles un support n'a pas encore été trouvé pour au moins l'une des valeurs de leur domaine. Tant que S^{sup} n'est pas vide, des projections de coût doivent être effectuées. La boucle principale de findSupports() vise à déceler un support pour chaque valeur de la contrainte courante. Une variable x est extraite à chaque tour de cette boucle, et trois actions successives sont accomplies. Tout d'abord, des projections concernant les valeurs dans le domaine de x sont effectuées (nous avons la garantie qu'il y en a au moins une qui est réalisée puisque x était dans S^{sup}). Ensuite, la paire (x, α) est placée dans `touchedVariables`, avec α correspondant au coût unaire minimal de c_x. Dans l'algorithme 28, ceci va nous permettre de mettre à jour les structures `maxUCosts` et `globalMaxUCost`, et de réaliser des projections unaires si α est strictement plus grand que 0. Si S^{sup} contient toujours au moins une variable, tableScan() est à nouveau appelé à la ligne 27 de la méthode findSupports(). Ce nouvel appel permet de mettre à jour les coûts minimaux ainsi que l'ensemble S^{sup}. Notons qu'après le premier appel, S^{val} est vide et nous avons la garantie que tous les tuples restants sont valides (les valeurs sont supprimées plus tard, hors contexte de la contrainte). Enfin, une incohérence peut être détectée précocement lorsque le coût $c_\emptyset$ ajouté au coût unaire minimal de c_x calculé est égal à k. Lorsque cela se produit, une exception est levée à la ligne 24.

Nous décrivons maintenant tableScan(), Méthode 2, qui est centrale dans notre algorithme. Les lignes 1 à 15 de cette méthode sont dédiées aux tuples explicites. La boucle démarrant à la ligne 2 parcourt successivement tous les tuples τ de la table courante. À la ligne 6, un test de validité est effectué pour le tuple τ. Ce test de validité est réalisé par isValidTuple(), Méthode 3, qui traite uniquement les variables présentes dans S^{val} (qui est l'ensemble vide après le premier scan de la table). Si le tuple courant τ est

valide, des structures sont potentiellement sujettes à des mises à jour. Plus spécifiquement, pour chaque variable $x \in S^{sup}$, si le coût γ de τ est inférieur au coût minimal courant de (x, a) enregistré dans $\texttt{minCosts}[x][a]$, avec $a = \tau[x]$, un appel à updateMinCost(), Méthode 4, est effectué. Cet appel met à jour la valeur de $\texttt{minCosts}[x][a]$, et de plus, quand $\gamma = 0$, nous savons que nous venons juste de trouver un support pour (x, a) dans la contrainte. Nous pouvons alors incrémenter $\texttt{nbGacValues}[x]$ et retirer x de S^{sup} dans le cas où (x, a) était la dernière valeur dans $dom(x)$ ne possédant pas encore de support. Pour les contraintes tables souples avec un coût par défaut différent de k, le comptage des tuples explicites valides est géré à la ligne 11. À la ligne 14, un tuple qui est invalide est supprimé en temps constant de la table courante : il est échangé avec le dernier tuple de la table courante avant que la valeur de `currentLimit` ne soit décrémentée.

Method 1: findSupports()

```
   /*Méthode à exécuter pour établir AC sur la contrainte          */
   // Initialisation
1  S^val ← ∅ ;
2  S^sup ← ∅ ;
3  foreach x ∈ S do
4      if lastSize[x] ≠ |dom(x)| then
5          S^val ← S^val ∪ {x} ;
6          lastSize[x] ← |dom(x)| ;
7      if x ∈ unassigned(P) then
8          S^sup ← S^sup ∪ {x} ;
9          nbGacValues[x] ← 0 ;
10         foreach value a ∈ dom(x) do
11             minCosts[x][a] ← k ;
12             if defaultCost ≠ k then
13                 nbExplicitValidTuples[x][a] ← 0 ;
   // Scans et projections
14 tableScan() ;                    // Premier scan de table ; S^sup mis à jour
15 S^val ← ∅ ;      // Plus de tests de validité après le premier scan de
   table
16 while S^sup ≠ ∅ do
17     pick and delete x from S^sup ;
18     α ← k ;
19     foreach a ∈ dom(x) do
20         if minCosts[x][a] > 0 then
21             projectDelta(this, x, a, minCosts[x][a]) ;
22         α ← min(α, c_x(a)) ;
23     if c_∅ ⊕ α = k then
24         throw FAILURE ;
25     touchedVariables.put(x, α) ;
26     if S^sup ≠ ∅ then
27         tableScan() ;                          // Scan de table additionnel
```

Method 2: tableScan()

```
  /*Scan de la table dans le but de supprimer les tuples invalides
    et de calculer les coûts minimaux des valeurs                    */
  // Trouver les coûts minimaux parmi les tuples explicites
1 i ← 1 ;
2 while i ≤ currentLimit do
3     index ← position[i] ;                          // index du i^ème tuple
4     τ ← table[index] ;                             // tuple courant τ
5     γ ← costs[index] ⊖ ⨁_{x∈S} deltas[x][τ[x]] ;   // coût de τ
6     if isValidTuple(τ) then
7         foreach variable x ∈ S^sup do
8             if γ < minCosts[x][τ[x]] then
9                 updateMinCost(x, τ[x], γ) ;
10            if defaultCost ≠ k then
11                nbExplicitValidTuples[x][τ[x]] + + ;
12        i + + ;
13    else
          // supprimer le tuple à la position i
14        swap(position[i], position[currentLimit]) ;
15        currentLimit − − ;
  // Mettre à jour les coûts minimaux à partir des tuples
     implicites
16 if defaultCost ≠ k then
17    nb ← |Π_{x∈S} dom(x)| ;
18    foreach variable x ∈ S^sup do
19        nbValidTuples[x] ← nb/|dom(x)| ;
20    if defaultCost = 0 then
21        handleZeroDefaultCost() ;
22    else
23        handleIntermediateDefaultCost() ;
```

Method 3: isValidTuple(τ : tuple) : Boolean

```
1 foreach variable x ∈ S^val do
2     if τ[x] ∉ dom(x) then
3         return false
4 return true
```

Method 4: updateMinCost(x : variable, a : value, γ : integer)

```
1 minCosts[x][a] ← γ ;
2 if γ = 0 then
3     nbGacValues[x] + + ;
4     if nbGacValues[x] = |dom(x)| then
5         S^sup ← S^sup \ {x};
```

Method 5: handleZeroDefaultCost()

```
1 foreach variable x ∈ S^sup do
2     foreach value a ∈ dom(x) do
3         if 0 < minCosts[x][a] ∧ nbImplicitTuplesWith(x, a) > 0 then
4             updateMinCost(x, a, 0) ;
```

Les lignes 16 à 23 de tableScan() traitent les tuples implicites. Tout d'abord, pour chaque variable x présente dans S^{sup}, nous calculons dans $\texttt{nbValidTuples}[x]$ le nombre de tuples valides pouvant être construits à partir des domaines courants des variables dans S et qui impliquent une valeur du domaine de x. Par exemple, si la contrainte est c_{xyz} et que $|dom(x)| = 3$, $|dom(y)| = 4$ et $|dom(z)| = 2$, alors $\texttt{nbValidTuples}[z] = |dom(x)| \times |dom(y)| = 3 \times 4 = 12$. Cela sera utile pour déterminer si des tuples implicites existent ou non pour une valeur donnée. Ensuite, nous avons deux cas à considérer : lorsque le coût par défaut est 0 et lorsqu'il est intermédiaire.

Supposons dans un premier temps que $\texttt{defaultCost} = 0$. Dans ce cas, handleZeroDefaultCost(), Méthode 5, est appelée à la ligne 21 de tableScan(). Spécifiquement, une fois que la table courante a été parcourue, nous avons besoin de vérifier l'existence d'un tuple implicite valide pour chaque valeur de S^{sup}. Le nombre de tuples implicites contenant (x, a) est obtenu à la ligne 3 de handleZeroDefaultCost() par un appel à nbImplicitTuplesWith(), Méthode 6 : il correspond au nombre de tuples valides possibles contenant (x, a) auquel on soustrait le nombre de tuples explicites valides dans `table` contenant (x, a). Si au moins un tuple implicite contenant (x, a) existe, $\texttt{minCosts}[x][a]$ est mis à jour à 0 (ligne 4).

À présent, supposons que $0 < \texttt{defaultCost} < k$. Dans ce cas, handleIntermediateDefaultCost(), Méthode 7, est appelée à la ligne 23 de tableScan(). Une fois que la table courante a été parcourue, nous avons besoin de trouver un tuple implicite valide de coût minimal pour chaque valeur (x, a) de S^{sup}. Considérons la valeur lb représentant le coût le plus faible qu'un tuple dans $l(S)$ pourrait avoir s'il était implicite (ligne 1). Ainsi, lb représente clairement une borne inférieure du coût minimal qui peut être obtenu depuis les tuples implicites. Nous entamons alors ces recherches en regroupant (lignes 3 à 6) dans un ensemble F toutes les valeurs (x, a) de S^{sup} telles que $\texttt{minCosts}[x][a] > lb$ et pour lesquelles il existe au moins un tuple implicite $\tau \in l(S)$ tel que $\tau[x] = a$. Si $\texttt{minCosts}[x][a]$ est plus grand que cette borne inférieure, cela signifie que $\texttt{minCosts}[x][a]$ peut être potentiellement amélioré en considérant l'ensemble des tuples implicites impliquant (x, a). Lorsque F n'est pas vide, tous les domaines sont triés par ordre décroissant des deltas associés à leurs valeurs, et donc selon un ordre de coût croissant (nous supposons ici que la fonction sort() appelé à la ligne 9 de la méthode handleIntermediateDefaultCost()

Method 6: nbImplicitTuplesWith(x :variable, a :value)

```
1 return nbValidTuples[x] − nbExplicitValidTuples[x][a]
```

Method 7: handleIntermediateDefaultCost()

```
   // Initialisation de F
1  lb ← defaultCost ⊖ ⨁_{x∈S} max{deltas[x][a] | a ∈ dom(x)} ;     // maximal sum of
   deltas
2  F ← ∅ ;
3  foreach variable x ∈ S^sup do
4      foreach value a ∈ dom(x) do
5          if nbImplicitTuplesWith(x, a) > 0 ∧ minCosts[x][a] > lb then
6              F ← F ∪ {(x, a)} ;
   // Tri des domaines selon les deltas
7  if F ≠ ∅ then
8      foreach variable x ∈ S do
9          sortedDomains[x] ← sort(dom(x), deltas[x]) ;        // trié dans l'ordre
           décroissant
   // Recherche des tuples implicites de coût minimal pour les
      valeurs dans F
10 while F ≠ ∅ do
11     pick and delete (x, a) from F ;
12     (τ, γ) ← searchImplicitTupleWithMinimalCost(x, a) ;
       // nous avons γ < minCosts[x][a], autrement nil est retourné
13     if (τ, γ) ≠ nil then
14         updateMinCost(x, a, γ) ;
15         foreach variable y ≠ x | (y, τ[y]) ∈ F do
16             if γ = 0 ∨ nbImplicitTuplesWith(y, τ[y]) = 1 then
                   // nous pouvons mettre à jour aussi le minCosts pour
                      d'autres valeurs
17                 if γ < minCosts[y][τ[y]] then
18                     updateMinCost(y, τ[y], γ) ;
19                 remove (y, τ[y]) from F ;
```

Method 8: searchImplicitTupleWithMinimalCost(x : variable, a : value)

```
/*Retourne une paire (τ,γ) où τ est un tuple implicite impliquant
  (x,a) de coût minimal γ < minCosts[x][a], ou nil s'il n'existe pas */

// Nous construisons un tuple τ contenant (x,a) avec le plus petit
   coût possible
1  τ[x] ← a ;
2  γ ← defaultCost ⊖ deltas[x][a] ;
3  foreach variable y ∈ S | y ≠ x do
4      τ[y] ← sortedDomains[y].first() ; // la première valeur a le plus petit
       coût
5      γ ← γ ⊖ deltas[y][τ[y]] ;
  /*R est un ensemble implémenté par une queue de priorité qui
    retourne d'abord les tuples de plus petit coût. Quand les coûts
    sont égaux, il retourne les tuples dans l'ordre
    lexicographique.                                              */
6  R ← {(τ,γ)} ;
  // Énumération des tuples contenant (x,a) par coût croissant
7  while R ≠ ∅ do
8      pick and delete the smallest pair (τ',γ') from R (tuple with least cost);
9      if minCosts[x][a] ≤ γ' then
10         return nil ;     // parce que les tuples restants ont un coût ≥ γ'
11     else if ¬ binarySearch(τ',table) then
12         return (τ',γ') ;  // Un tuple implicite de coût minimal est trouvé
13     else
14         R ← R ∪ buildSuccessors(x,(τ',γ')) ;
```

Method 9: buildSuccessors(x : variable, (τ : tuple, γ : integer)) : set of (τ_i, γ_i)

```
/*Retourne la liste des successeurs immédiats d'un tuple τ de
  coût γ tel que la valeur de x est préservée.                    */
1  sucessors ← ∅ ;
2  foreach variable y ∈ S | y ≠ x do
3      b ← sortedDomains[y].succ(τ[y]) ;   // la valeur suivante selon l'ordre
       de coût croissant
4      if b ≠ nil then
5          τ' ← τ_{y=b} ;          // τ' est une copie de τ avec y assignée à b
6          γ' ← γ ⊕ deltas[y][τ[y]] ⊖ deltas[y][τ'[y]] ;      // γ' est le coût de τ'
7          successors ← successors ∪ {(τ',γ')} ;
8  return successors ;
```

réalise cette opération). Cela s'avèrera utile pour générer les tuples implicites par ordre de coût croissant. Trier les domaines peut être effectué à ce niveau assez tôt de l'exécution parce que les deltas des valeurs ne sont jamais modifiés dans la boucle démarrant à la ligne 10, qui traite toutes les valeurs de F. Pour chacune de ces valeurs, searchImplicitTupleWithMinimalCost(), Méthode 8, est appelée à la ligne 12. Cette fonction retourne soit une paire (τ, γ) avec τ un tuple implicite valide contenant (x, a) de coût minimal γ strictement inférieur à $\texttt{minCosts}[x][a]$, soit nil si un tel tuple n'existe pas. Lorsque ce n'est pas nil qui est retourné, il est alors possible de mettre à jour $\texttt{minCosts}[x][a]$ à la ligne 14. Le tuple τ trouvé peut également être exploité pour les autres valeurs $(y, \tau[y])$ dans F sous réserve que $\gamma = 0$ ou que le nombre de tuples implicites valides contenant $(y, \tau[y])$ est égal à 1. De ce fait, $\texttt{minCosts}[y][\tau[y]]$ peut potentiellement être amélioré et $(y, \tau[y])$ peut alors être supprimée de F de façon sûre.

Intéressons nous maintenant à searchImplicitTupleWithMinimalCost(), Méthode 8. Le premier tuple valide τ impliquant (x, a) et de coût minimal γ est construit aux lignes 1 à 5 : il contient alors (x, a) et, pour chaque variable $y \neq x$, il contient la première valeur de son domaine trié. Ensuite, (τ, γ) est la première paire placée dans l'ensemble R, introduit pour enregistrer les candidats au rôle de tuple implicite valide de coût minimal pour (x, a). À chaque itération de la boucle démarrant à la ligne 7, une paire (τ', γ') de R, pour laquelle il n'existe aucune autre paire (τ'', γ'') dans R avec $\gamma'' < \gamma'$, est extraite (et donc supprimée) de R. Pour les tuples de coûts égaux, le tuple le plus petit selon l'ordre lexicographique est retourné. Un premier test est effectué à la ligne 9. Si $\texttt{minCosts}[x][a] \leq \gamma'$, cela signifie qu'il n'existe aucun tuple implicite valide impliquant (x, a) qui puisse avoir un coût strictement inférieur à $\texttt{minCosts}[x][a]$: par construction, ces tuples ont un coût qui est nécessairement supérieur ou égal à γ', et donc à $\texttt{minCosts}[x][a]$ par transitivité. De ce fait, nil est retourné. Un deuxième test est réalisé à la ligne 11. Si τ' n'est pas présent dans `table`, cela signifie que τ' est un tuple implicite. Ainsi, nous avons trouvé un tuple implicite valide de coût minimal pour (x, a), qui peut alors être retourné ; son coût γ' améliore nécessairement la valeur de $\texttt{minCosts}[x][a]$. Enfin, lorsque τ' appartient à `table`, τ' est explicite et donc nous devons continuer à chercher un tuple implicite pertinent : pour réaliser cela, nous ajoutons dans R toutes les paires correspondant aux successeurs de τ' basés sur la variable x. Si une même paire est déjà présente dans R, elle n'est pas insérée une seconde fois.

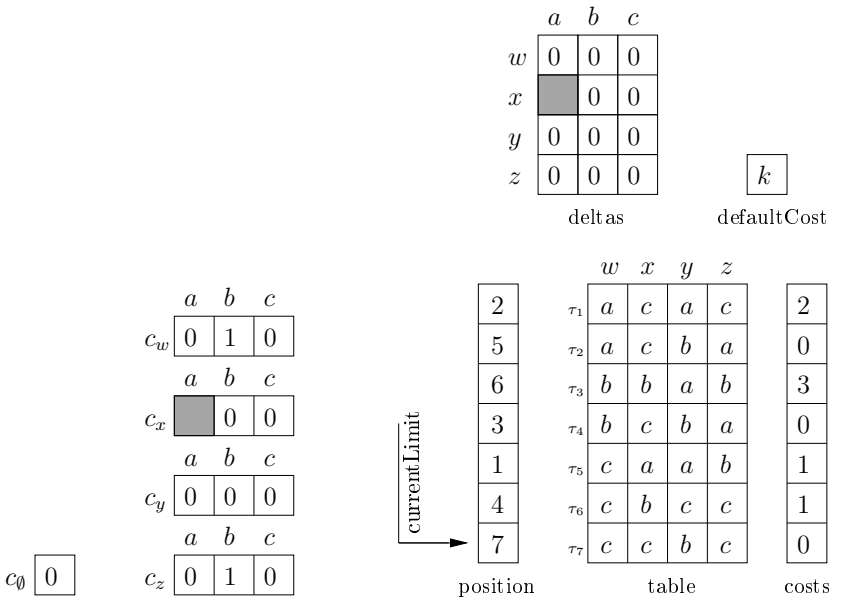
La méthode buildSuccessors(), Méthode 9, nous permet de construire ces paires successeurs. Plus précisément, nous construisons un ensemble `successors` composé d'au plus $|S| - 1$ nouvelles paires, potentiellement une pour chaque variable $y \neq x$ of S. Pour chaque variable $y \neq x$ dans S, nous considérons la valeur b qui vient juste après $\tau[y]$ dans $\texttt{sortedDomains}[y]$. Si cette valeur existe (c'est à dire que $\tau[x]$ n'est pas la dernière valeur dans le domaine trié de x et que donc nil n'est pas retourné), un nouveau tuple τ' est construit en clonant τ et en positionnant la valeur b à y dans τ'. La paire (τ', γ') est ajoutée dans `successors` avec γ' représentant le coût de τ'.

3.7 Illustration

Dans cette section, le fonctionnement de findSupports(), Méthode 1, est illustré. Nous nous intéressons aux quatre contraintes unaires c_w, c_x, c_y et c_z, représentées dans la figure 3.3(a), ainsi qu'à la contrainte quaternaire c_{wxyz} représentée dans la figure 3.3(b). Nous supposons que l'évènement $x \neq a$ a déclenché l'exécution de findSupports() à la ligne 5 de l'algorithme GAC*-WSTR pour la contrainte c_{wxyz}. Avant un premier appel à tableScan(), le tableau `minCosts` est rempli avec la valeur k et l'ensemble $\texttt{S}^{\texttt{sup}}$ contient toutes les variables du scope de la contrainte ; voir le haut de la figure 3.4(a). Alors, tous les tuples sont parcourus, en commençant par τ_2.[18] Puisque ce tuple est valide, la structure `minCosts` est mise à jour : k est remplacé par 0, le coût de τ_2, pour chaque valeur de τ_2. Ensuite τ_5 est considéré.

[18] Les tuples ne sont pas ordonnés comme dans la figure 3.1(a) pour les besoins de l'illustration.

En raison de $x \neq a$, τ_5 n'est plus valide, et par conséquent les positions de τ_5 et du dernier tuple valide courant sont échangées et `currentLimit` est décrémenté. Tous les autres tuples demeurant valides, les structures alors obtenues sont représentées dans les figures 3.4(a) (bas) et 3.4(b). En particulier, on constate que la variable w a pu être supprimée de $\mathtt{S}^{\mathtt{sup}}$ puisque, après avoir traité tous les tuples, toutes les valeurs de w ont un coût minimal égal à 0 (ce qui veut dire que toutes les valeurs possèdent au moins un support dans cette contrainte). Comme $\mathtt{S}^{\mathtt{sup}}$ n'est pas vide, nous entrons dans la boucle à la ligne 16 de findSupports(), Méthode 1. Ici, nous supposons que x est la première variable extraite de $\mathtt{S}^{\mathtt{sup}}$. Par conséquent, une projection est effectuée concernant (x, b) puisque son coût minimal dans `minCosts` est strictement positif. La figure 3.5(a) montre que le coût unaire de (x, b) est passé de 0 à 1 et que $\mathtt{S}^{\mathtt{sup}}$ est mis à jour, alors que la figure 3.5(b) montre les coûts mis à jour de certains tuples de la contrainte c_{wxyz} (en vert, nous présentons les sommes de delta par tuple, lorsqu'elle est différente de 0, ainsi que le coût courant des tuples). Puisque $\mathtt{S}^{\mathtt{sup}}$ n'est pas encore vide, un second appel à tableScan() est effectué. Ce second scan, qui est nécessaire dû à la projection qui vient juste d'être effectuée, nous permet de mettre à jour les coûts minimaux comme cela est montré dans la figure 3.6(a) ; la table associée à c_{wxyz} demeure inchangée comme le montre la figure 3.6(b). Finalement, nous supposons que la variable y est extraite de $\mathtt{S}^{\mathtt{sup}}$, qui nécessite une nouvelle projection concernant cette fois-ci (y, a). Le résultat est illustré dans la figure 3.7.

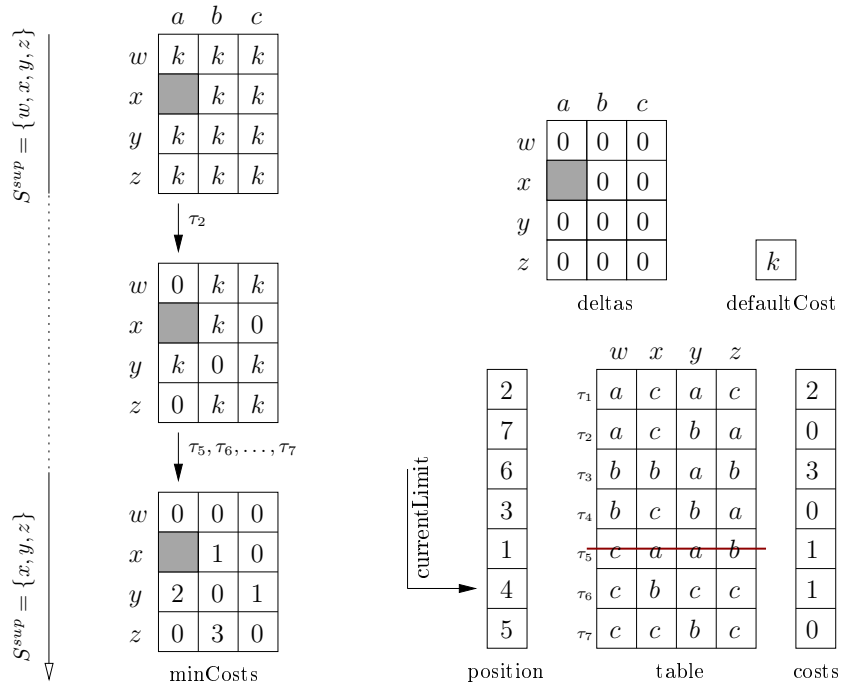


(a) Contraintes unaires et 0-aire.

(b) Une contrainte table souple c_{wxyz} avec un coût par défaut k.

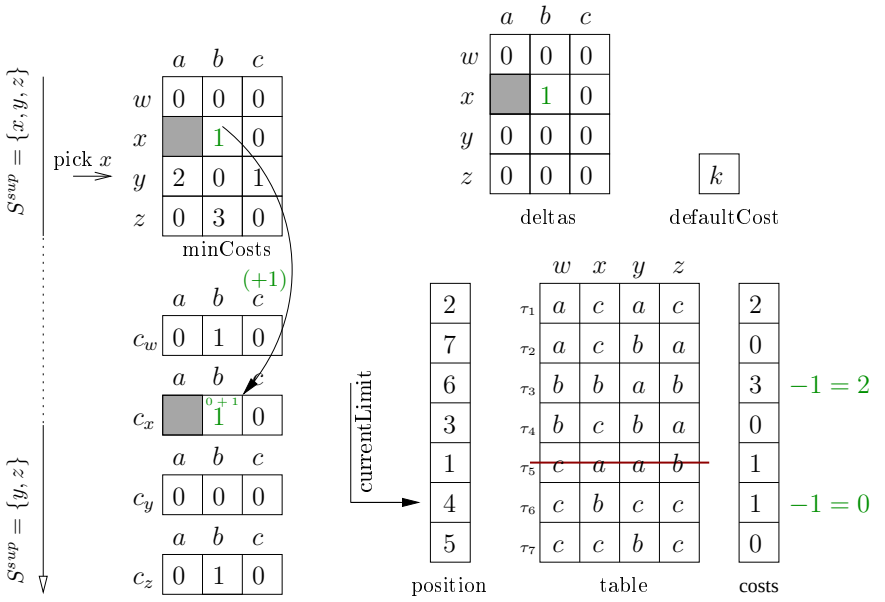
FIG. 3.3 – Initialement. Nous assumons que les valeurs de delta sont toutes à 0 et que (x, a) vient juste d'être supprimée.

Notre seconde illustration concerne la façon dont les tuples implicites sont traités. Cette fois-ci, nous considérons une contrainte table souple ternaire c_{xyz} avec un coût par défaut intermédiaire égal à 10. Nous supposons que les valeurs de delta associées à c_{xyz} ne sont pas toutes égales à 0 et que les valeurs (x, a) et (z, c) ont été supprimées ; voir la figure 3.8. Un premier appel à tableScan() initialise, juste avant d'exécuter handleIntermediateDefaultCost() à la ligne 23, les structures `minCosts`, `nbValidTuples` et `nbExplicitValidTuples` comme illustré en haut de la figure 3.9. Par exemple, nous avons $\mathtt{nbValidTuples}[x] = 6$ puisque $|dom(y)| \times |dom(z)| = 3 \times 2 = 6$, et nous avons $\mathtt{nbExplicitValidTuples}[z][a] = 2$ puisque il y a exactement deux tuples explicites valides conte-



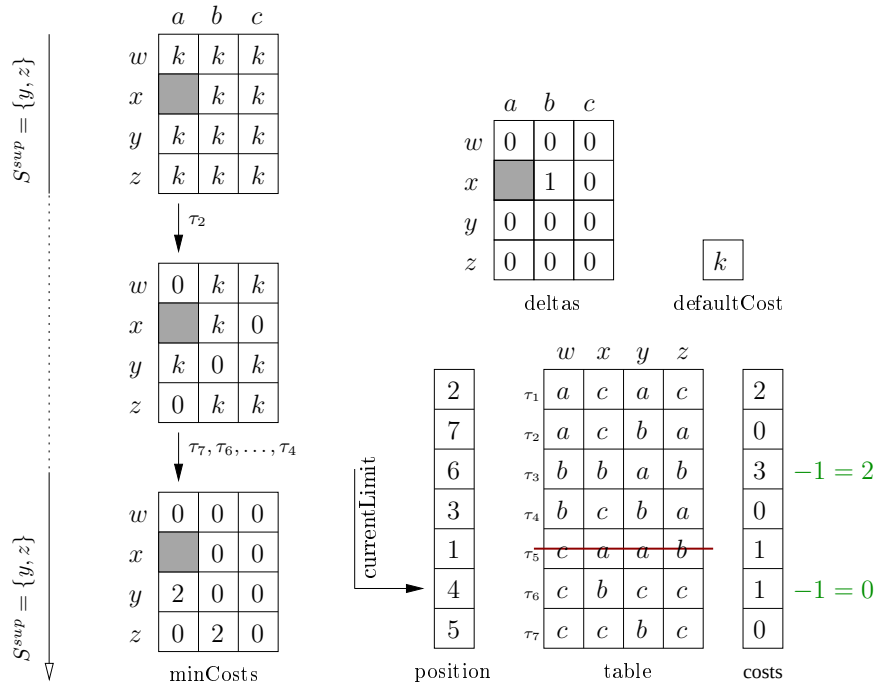
(a) Calcul des coûts minimaux. (b) c_{wxyz} après le scan.

FIG. 3.4 – Premier scan de table de c_{wxyz}.



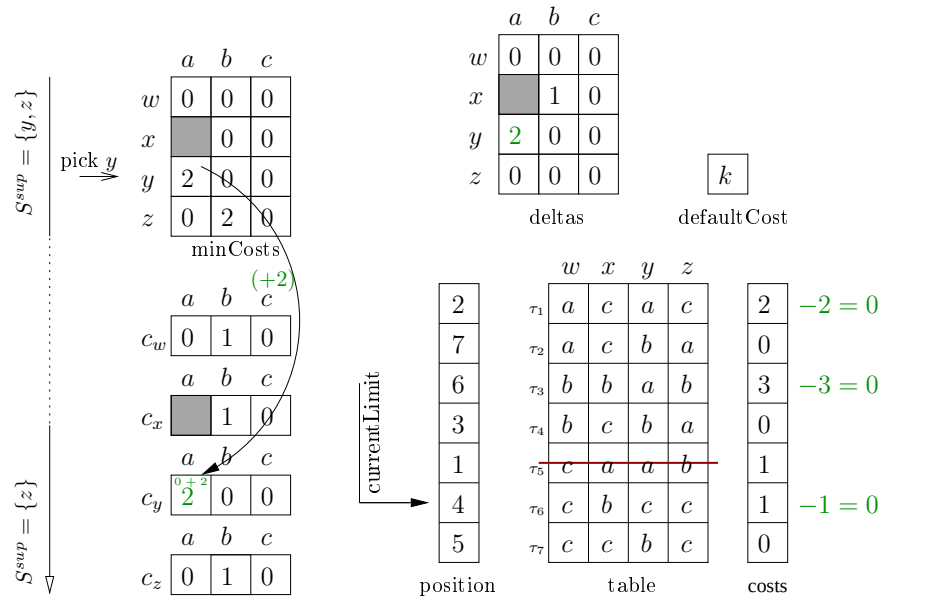
(a) Modification de c_x. (b) Modification du coût de certains tuples de c_{wxyz}.

FIG. 3.5 – Projections sur x : project($c_{wxyz}, x, b, 1$) est exécuté.



(a) Calcul des coûts minimaux. (b) c_{wxyz} inchangée.

FIG. 3.6 – Second scan de table de c_{wxyz}.



(a) Modification de c_y. (b) Modification de (valeurs de delta associées avec) c_{wxyz}.

FIG. 3.7 – Projections sur y : project($c_{wxyz}, y, a, 2$) est exécuté.

nant (z, a). L'exécution des lignes 1 à 9 de handleIntermediateDefaultCost(), Méthode 7, initialise les structures lb, F et `sortedDomains` comme illustré dans le bas de la figure 3.9. En effet, $lb = 10 - 1 - 2 - 1 = 6$, et donc F ne contient pas (x, b), (y, b) et (z, b) puisque c_{xyz}.`minCosts`$[x][b] = c_{xyz}$.`minCosts`$[y][b] = c_{xyz}$.`minCosts`$[z][b] = 3 < 6$ (en d'autres termes, il n'est pas possible de trouver parmi les tuples implicites un coût plus petit que le coût minimal de 3 déjà trouvé pour ces valeurs, puisque le plus petit coût possible envisageable parmi ces tuples implicites est égal à 6). F ne contient pas non plus (y, a) parce qu'il n'y a pas de tuple implicite valide contenant (y, a) : il y a 4 tuples valides et 4 tuples explicites valides. Considérons maintenant la boucle principale de handleIntermediateDefaultCost() débutant à la ligne 10. Si (x, c) est la première valeur extraite de F, nous avons alors le scénario décrit dans la figure 3.10. Le tuple valide contenant (x, c) de plus faible coût (ici 6) est construit en prenant la valeur de plus faible coût dans chaque domaine : (c, b, b). Lorsque nous parcourons la table afin de déterminer si (c, b, b) est implicite ou non, la réponse est positive, ce qui veut dire que nous avons trouvé un tuple implicite de coût 6, qui nous permet de mettre à jour `minCosts` (voir la valeur de couleur verte pour (x, c) dans la figure 3.12). Si (y, c) est la seconde valeur extraite de F, le scénario est celui illustré dans la figure 3.11. Le premier tuple généré considérant (y, c) est (c, c, b), qui correspond à un tuple valide contenant (y, c) de coût le plus faible 8. Ce tuple est intéressant car son coût de 8 est strictement inférieur au coût minimal c_{xyz}.`minCosts`$[y][c]$ déjà enregistré pour (y, c) qui vaut 9. Cependant, ce tuple appartient à la table (en d'autres termes (c, c, b) est un tuple explicite), et par conséquent les deux tuples successeurs de (c, c, b) pour y sont générés et ajoutés dans R. Lorsque le plus petit tuple de R est extrait, en l'occurrence (b, c, b) (les deux tuples présents dans R ayant le même coût, c'est l'ordre lexicographique qui est utilisé pour les distinguer), nous voyons immédiatement que nous ne serons pas capables d'améliorer le score minimal de 9 (déjà trouvé avec un tuple explicite) : en effet, les tuples sont itérés par ordre de coût croissant et le tuple courant (b, c, b) considéré a un coût déjà égal à 9. Nous stoppons alors la boucle en retournant nil. Après avoir traité la dernière valeur (z, a) de F, `minCosts` est alors comme décrit dans la figure 3.12.

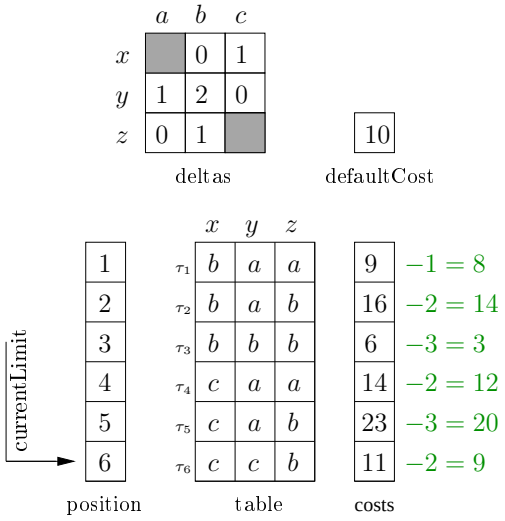


FIG. 3.8 – Une contrainte table souple ternaire c_{xyz} avec un coût par défaut intermédiaire 10. Nous supposons certaines valeurs de delta différentes de 0, et aussi que (x, a) et (z, c) ont été (précédemment) supprimées.

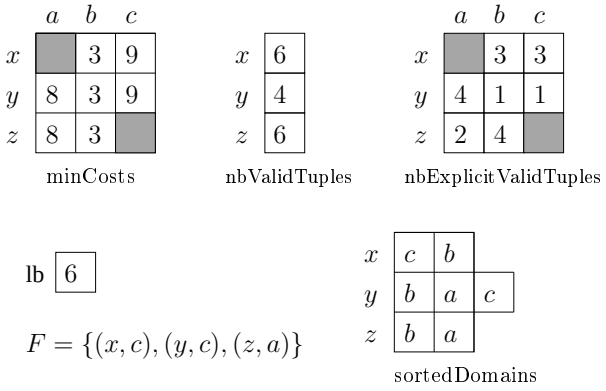


FIG. 3.9 – État des structures avant d'entrer dans la boucle principale (débutant à la ligne 10) de handleIntermediateDefaultCost().

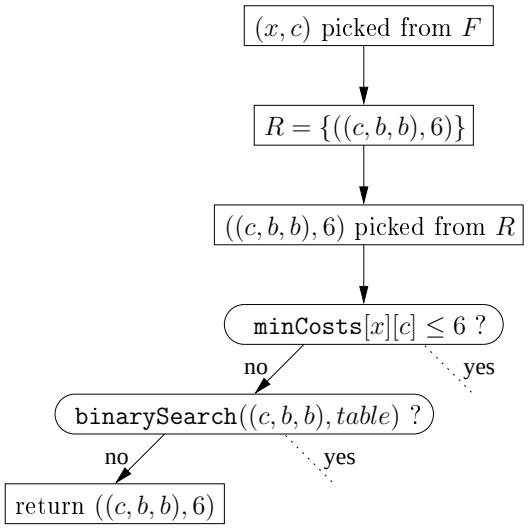


FIG. 3.10 – Traiter les tuples implicites pour (x, c).

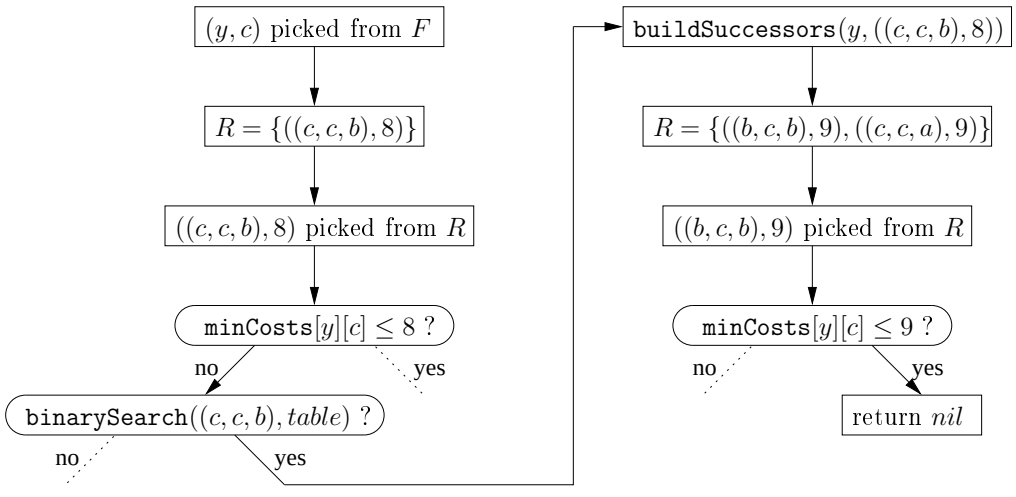


FIG. 3.11 – Traiter les tuples implicites pour (y, c).

	a	b	c
x		3	6
y	8	3	9
z	7	3	

minCosts

FIG. 3.12 – Valeurs finales de minCosts.

3.8 Correction et complexité de l'algorithme

Dans cette section, nous prouvons que l'algorithme que nous proposons maintient effectivement GAC* en temps polynomial quelle que soit la valeur de k. Dans un premier temps, nous prouvons la correction et la polynomialité. Dans un second temps, nous présentons une analyse détaillée de la complexité.

3.8.1 Correction et Polynomialité

Nous commençons par prouver que tableScan(), Méthode 2, maintient une liste correcte de tuples valides et invalides.

Lemme 1 isValidTuple() *identifie correctement les tuples valides et invalides et* tableScan() *assure qu'une fois la méthode terminée les tuples dans* 1..`currentLimit` *sont tous valides et que les tuples au delà de* `currentLimit` *sont tous invalides.*

Preuve Puisque `lastSize`$[x]$ est initialisé à $|dom^{init}(x)|$ au début du programme, S^{val} est vide pour le premier appel à tableScan(), avant que toute décision ne soit prise. Par conséquent, isValidTuple() renvoie toujours vrai durant le premier scan de table, ce qui est correct puisqu'il est supposé que les tuples contiennent uniquement des valeurs de dom^{init} et qu'avant la première décision, nous avons $\forall x, dom(x) = dom^{init}(x)$. Par conséquent, après le premier appel à tableScan(), `currentLimit` n'a pas été modifié et il reste encore égal au nombre de tuples explicites.

Maintenant, supposons qu'à un moment donné de la recherche, tous les tuples dans 1..`currentLimit` sont effectivement valides et que tous les tuples au delà de `currentLimit` sont effectivement invalides, puis qu'une décision soit prise. Puisque les décisions et les propagations ne peuvent que réduire les domaines des variables, une fois que le domaine d'une variable x a été modifié, nous avons nécessairement `lastSize`$(x) \neq |dom(x)|$ et par conséquent x est présent dans S^{val} (voir l'initialisation de findSupports()). La boucle *foreach* de isValidTuple() vérifie pour chaque variable de S^{val} que $\tau[x]$ est toujours dans $dom(x)$. Les variables qui ne sont pas énumérées par cette boucle ne peuvent pas avoir eu leur domaine réduit et donc ne peuvent pas être la cause de l'invalidité d'un tuple. Par conséquent, isValidTuple() est correcte.

Lorsqu'un tuple devient invalide, tableScan() l'échange avec le dernier tuple et décrémente la valeur de `currentLimit`. Lors d'un retour en arrière, `currentLimit` est restauré à la valeur qui était la sienne avant que la décision ne soit prise. Par conséquent, un tuple qui est devenu invalide à cause de cette décision reviendra dans l'état valide après le retour en arrière. Ceci garantit que, après un appel à tableScan(), les tuples dans 1..`currentLimit` sont tous valides et que les tuples au delà de `currentLimit` sont tous invalides. □

Lemme 2 *Au cours de l'exécution de* findSupports() *sur une contrainte* c_S, updateMinCost() *est appelé seulement une fois avec* $\gamma = 0$ *pour une valeur donnée* (x, a) *de* S, *de ce fait les valeurs de* `nbGacValues` *sont correctes.*

Preuve Chaque appel à updateMinCost(x, a, γ) est précédé par un test nous garantissant que $\gamma <$ `minCosts`$[x][a]$. Le seul endroit où ce n'est pas explicite dans le code est dans la méthode 7 à la ligne 14, mais updateMinCost() est précédé par un appel à searchImplicitTupleWithMinimalCost() qui garantit que le coût γ qu'il retourne est strictement inférieur à `minCosts`$[x][a]$.

Lorsque updateMinCost(x, a, γ) est appelé avec $\gamma = 0$, il fixe `minCosts`$[x][a] = 0$ et incrémente `nbGacValues`$[x]$. Étant donnée la remarque précédente, cela peut se produire seulement une fois durant le filtrage d'une contrainte. □

Lemme 3 *Après l'exécution de* tableScan(), *Méthode 2, sur une contrainte c_S, nous avons*

$$\forall x \in S \cap unassigned(P), \forall a \in dom(x), \texttt{minCosts}[x][a] = min(\{c_S(\tau) \mid \tau \in l(S) \wedge \tau[x] = a\})$$

Preuve

1. Nous prouvons d'abord que, à la ligne 16 de l'exécution de la méthode tableScan() sur une contrainte c_S, pour chaque valeur (x, a) de S (avec x non assignée), `minCosts`$[x][a]$ est soit le plus petit coût d'un tuple valide explicite τ tel que $\tau[x] = a$, soit k si aucun tuple explicite contenant (x, a) n'est valide. Dans un premier temps, chaque élément du tableau `minCosts` est initialisé à k (le plus grand coût possible) à la ligne 11 de la méthode 1. Dans la méthode tableScan(), les lignes 1 à 15 énumèrent tous les tuples τ ainsi que leur coût courant associé γ. Lorsque τ est invalide, il est échangé, et lorsqu'il est valide, updateMinCost() est appelé quand γ est plus petit que `minCosts`$[x][\tau[x]]$ pour mettre à jour cette valeur à γ. Pour une valeur (x, a), s'il n'y a pas de tuple valide explicite τ tel que $\tau[x] = a$, alors `minCosts`$[x][a]$ conserve sa valeur initiale k. Sinon, par construction, `minCosts`$[x][a] = min\{c_S(\tau) \mid \tau \in c_S.table \wedge valid(\tau) \wedge \tau[x] = a\}$.
2. Nous prouvons maintenant que le code à partir de la ligne 16 de la méthode tableScan() calcule pour tous les $x \in S \cap unassigned(P)$ et les $a \in dom(x)$ le plus petit coût d'un tuple implicite valide τ tel que $\tau[x] = a$, et modifie `minCosts`$[x][a]$ en conséquence.

   Tout d'abord, si `defaultCost` $= k$, alors tous les tuples implicites ont un coût k et `minCosts`$[x][a]$ ne peut être modifié. De ce fait, la procédure ne fait rien dans ce cas.

   Lorsque `defaultCost` $= 0$, alors tous les tuples implicites ont un coût 0. De ce fait, s'il existe un tuple implicite valide tel que $\tau[x] = a$, alors `minCosts`$[x][a] = 0$. Ceci est vérifié grâce à handleZeroDefaultCost(), Méthode 5, où le nombre total de tuples valides tels que $\tau[x] = a$ calculé par $\prod_{y \in S, y \neq x} |dom(y)|$ est comparé avec le nombre de tuples explicites valides tels que $\tau[x] = a$. Si ces nombres sont différents, un tuple implicite avec un coût 0 tel que $\tau[x] = a$ existe.

   Lorsque `defaultCost` ne correspond ni à 0 ni à k, handleIntermediateDefaultCost(), Méthode 7, est appelée afin d'identifier les valeurs (x, a) pour lesquelles il est pertinent d'identifier le plus petit coût parmi les tuples implicites contenant cette valeur : ces valeurs sont stockées dans un ensemble F. D'abord, on peut observer que pour les variables $x \notin S^{sup}$, qui possèdent déjà un support, aucune factorisation de coût n'est possible et que par conséquent, il est inutile de mettre à jour `minCosts`$[x][a]$ pour tout $a \in dom(x)$. Par conséquent, ces valeurs sont exclues de F. En outre, s'il n'y a pas de tuple implicite contenant une valeur donnée, cette valeur peut être omise de F. Enfin, il est facile de calculer le plus petit coût lb d'un tuple (implicite ou explicite). Les valeurs telles que `minCosts`$[x][a] \leq lb$ n'ont pas à apparaître dans F, parce que les tuples implicites ne peuvent jamais réduire le `minCosts` de ces valeurs. À la ligne 7 de handleIntermediateDefaultCost(), F contient toutes les valeurs exceptées celles qui ne sont clairement pas pertinentes, comme expliqué ci-dessus. Si F est vide, la méthode handleIntermediateDefaultCost() s'arrête là. Sinon, pour

chaque valeur (x, a) dans F, elle identifie un tuple implicite de plus petit coût avec un coût strictement inférieur à $\texttt{minCosts}[x][a]$ dans l'appel à searchImplicitTupleWithMinimalCost(), Méthode 8, dont la correction est prouvée par le lemme 5. S'il y en a un qui est trouvé, $\texttt{minCosts}[x][a]$ est mis à jour et la boucle *foreach* de la ligne 15 vérifie si ce tuple de plus petit coût peut aussi être utilisé pour mettre à jour le coût minimal d'autres valeurs, de manière paresseuse. À cette fin, nous identifions si ce tuple implicite de plus petit coût τ contient une valeur $(y, \tau[y])$ qui est dans F (avec $y \neq x$) et tel que soit son coût est 0, soit c'est le seul tuple implicite contenant $(y, \tau[y])$. Dans un tel cas, il ne peut pas exister un autre tuple implicite contenant $(y, \tau[y])$ avec un coût inférieur à τ et, par conséquent, $\texttt{minCosts}[y][\tau[y]]$ peut être mis à jour directement et $(y, \tau[y])$ peut être supprimée de F.

□

Avant de prouver la correction de searchImplicitTupleWithMinimalCost(), Méthode 8, il nous est nécessaire d'introduire quelques notions.

Rappelons que, par construction, le coût $cost(\tau)$ d'un tuple implicite τ est obtenu en soustrayant du coût par défaut la somme de tous les $\texttt{deltas}[x][\tau[x]]$ pour chaque $x \in vars(\tau)$:

$$cost(\tau) = defaultCost \ominus \bigoplus_{x \in vars(\tau)} \texttt{deltas}[x][\tau[x]]$$

Soit $\preceq$ l'ordre défini sur les tuples par $\tau \preceq \tau' \Leftrightarrow cost(\tau) < cost(\tau') \vee (cost(\tau) = cost(\tau') \wedge \tau \leq_{\text{lex}} \tau')$ où $\leq_{\text{lex}}$ est l'ordre lexicographique habituel sur les tuples. En fait, $\preceq$ est un ordre lexicographique sur les paires $(cost(\tau), \tau)$. Nous utilisons $\tau \prec \tau'$ comme abréviation de $\tau \preceq \tau' \wedge \tau \neq \tau'$.

Soit $succ_x(v)$ la fonction de succession pour les valeurs du domaine de x triées par coût décroissant. Dans les algorithmes, $succ_x(v)$ est noté $\texttt{sortedDomains}[x].\text{succ}(v)$. Un tuple τ' est le successeur direct d'un tuple τ si et seulement si il existe une variable x telle que $\tau'[x] = succ_x(\tau[x])$ et $\forall y \neq x, \tau'[y] = \tau[y]$. Nous notons $\tau \rightarrow \tau'$ si et seulement si τ' est un successeur direct de τ et $\tau \rightarrow_* \tau'$ si et seulement si il existe une séquence de relations $\tau \rightarrow \tau_1, \tau_1 \rightarrow \tau_2, \ldots \tau_n \rightarrow \tau'$ (fermeture transitive). Par construction, $\tau \rightarrow_* \tau'$ implique $\tau \prec \tau'$.

La queue de priorité R utilisée dans la méthode searchImplicitTupleWithMinimalCost() doit garantir de retourner le plus petit tuple selon $\prec$ mais aussi qu'un tuple donné ne puisse pas être inséré deux fois dans la queue. Par exemple, si nous considérons deux variables x et y telles que $dom(x) = dom(y) = \{a, b\}$, $succ_x(a) = b$ et $succ_y(a) = b$, le tuple de plus petit coût sera $\{a, a\}$. À la première itération, nous insérerons dans R les successeurs $\{b, a\}$ et $\{a, b\}$. Lorsque ces tuples sont extraits de la queue, ils vont générer tous deux le même successeur $\{b, b\}$ qui devra être inséré une seule fois dans R afin d'éviter des calculs redondants. Dans le pseudo-code, ceci est réalisé grâce à l'union des ensembles à la ligne 14 de searchImplicitTupleWithMinimalCost(). En pratique, cette queue particulière est obtenue en modifiant légèrement l'implémentation classique d'une queue de priorité avec un tas binomial.

Le lemme 4 garantit que la boucle *while* dans searchImplicitTupleWithMinimalCost(), Méthode 8, ne boucle pas indéfiniment et que les tuples sont énumérés par ordre de coût croissant.

Lemme 4 *Les propriétés suivantes sont garanties par la boucle* while *dans la méthode 8 :*

A) les tuples sont énumérés par ordre de coût croissant ;

B) une fois qu'un tuple τ est extrait de la queue de priorité, il ne peut plus jamais y être inséré à nouveau.

Preuve Soit τ_1 un premier tuple extrait de la queue et τ_2 un tuple extrait de la queue après l'extraction de τ_1. Deux cas possibles : soit τ_2 était déjà présent dans la queue lorsque τ_1 a été extrait, soit τ_2 a a été inséré dans la queue après l'extraction de τ_1.

1. Si τ_2 était déjà présent dans la queue lorsque τ_1 a été extrait, alors la queue de priorité garantit que $\tau_1 \prec \tau_2$, qui implique à la fois que $cost(\tau_1) \leq cost(\tau_2)$ et que $\tau_1 \neq \tau_2$. De ce fait, les propriétés A) et B) sont satisfaites.
2. Si τ_2 a été inséré dans la queue après l'extraction de τ_1, alors il existe un tuple τ tel que $\tau \rightarrow_* \tau_2$ et soit $\tau = \tau_1$, soit τ était présent dans la queue lorsque τ_1 a été extrait. Nous pouvons conclure que $\tau_1 \preceq \tau$ et $\tau \prec \tau_2$. Ceci implique $\tau_1 \prec \tau_2$ ainsi que $cost(\tau_1) \leq cost(\tau_2)$ et $\tau_1 \neq \tau_2$. Par conséquent, les propriétés A) et B) sont satisfaites.

□

Lemme 5 searchImplicitTupleWithMinimalCost(), *Méthode 8, retourne soit un tuple implicite de coût minimal contenant* (x, a) *avec un coût strictement inférieur à* $\texttt{minCosts}[x][a]$*, soit nil si un tel tuple n'existe pas.*

Preuve Il est relativement simple d'observer dans la méthode searchImplicitTupleWithMinimalCost() que la première boucle *foreach* construit un tuple τ avec $\tau[x] = a$ avec le plus petit coût possible. Il est clair aussi que buildSuccessors() retourne tous les successeurs directs du tuple τ laissant $\tau[x]$ inchangé. Si nous considérons la boucle *while* dans la méthode searchImplicitTupleWithMinimalCost() sans les instructions *return*, le lemme 4 nous dit qu'il énumère tous les tuples tels que $\tau[x] = a$ selon un ordre de coût croissant. Dès qu'un tuple a un coût supérieur ou égal à $\texttt{minCosts}[x][a]$, tous les tuples suivants auront un coût supérieur ou égal à $\texttt{minCosts}[x][a]$ et donc la boucle peut être interrompue de manière sûre. Enfin, la recherche binaire dans les tuples explicites de la table effectue seulement une recherche dans les tuples explicites sans modifier les autres propriétés de la boucle. □

Lemme 6 *Après l'exécution de* findSupports(), *Méthode 1, sur une contrainte* c_S*, toutes les valeurs de* S *ont un support dans* c_S*.*

Preuve Les appels à tableScan() regroupent dans S^{sup} les valeurs de $S \cap unassigned(P)$ qui ne possèdent pas encore de support (et mettent à jour $\texttt{minCosts}$). Tant que S^{sup} n'est pas vide, il existe une valeur (x, a) telle que $\texttt{minCosts}[x][a] \neq 0$. Une telle valeur est extraite de S^{sup} et une projection est effectuée ayant pour résultat $\texttt{minCosts}[x][a] = 0$. Puisque $\texttt{minCosts}$ peut seulement diminuer, (x, a) n'appartiendra pas à S^{sup} après les appels suivants à tableScan(). Donc, la boucle *while* dans findSupports() se termine avec $S^{sup} = \emptyset$, ce qui veut dire que toutes les valeurs de chaque variable dans $S \cap unassigned(P)$ ont un support dans c_S.

Une fois que chaque variable non assignée a un support, les variables assignées ont nécessairement un support. Par conséquent, toutes les valeurs de S ont un support dans c_S. □

Lemme 7 *Après l'exécution de GAC*-WSTR, chaque variable est GAC*-cohérente.*

Preuve Une première remarque est que seules des EPT sont utilisées et le WCN obtenu après l'exécution de GAC*-WSTR est ainsi donc équivalent au réseau initial.

Dès lors qu'il n'est pas certain qu'une valeur (x, a) possède un support pour la contrainte c_S, findSupports() est appelée sur c_S pour trouver ce support. Par conséquent, à la fin de l'algorithme GAC*-WSTR, Algorithme 28, nous pouvons conclure que toutes les valeurs ont un support pour toutes les contraintes. À présent, il nous reste juste à prouver que toutes les variables sont NC*.

Dès qu'une variable x peut avoir un de ses coûts unaires modifié par findSupports(), une mise à jour de `touchedVariables` est effectuée pour contenir la paire (x, α) avec $\alpha = min\{c_x(a) \mid a \in dom(x)\}$. Toute projection unaire possible est effectuée à la ligne 8 de l'algorithme 28. Par conséquent, à la fin, toutes les variables ont un support unaire.

Lorsque le domaine ou un coût unaire de la variable x est modifié, updateMaxUCostOf() est appelée et calcule le coût unaire maximum dans `maxUCosts[x]`. Quand $c_\emptyset \oplus$ `maxUCosts[x]` $= k$, au moins une valeur peut être supprimée du domaine de x. Puisque par construction nous avons $\forall x \in unassigned(P)$, `globalMaxUCost` $\geq max\{c_x(a) \mid a \in dom(x)\}$, il est garantit que pruneVar() est appelée à la ligne 14 dès lors qu'un domaine peut être réduit. Il est facile de vérifier que pruneVar() supprime toute valeur a telle que $c_\emptyset \oplus c_x(a) = k$.

Puisque toute variable x a un support unaire et que chaque valeur a telle que $c_\emptyset \oplus c_x(a) = k$ est supprimée, nous pouvons conclure que toutes les variables sont NC*. Par conséquent, le WCN est GAC*-cohérent dès lors que l'algorithme 28 est complet.

□

Un point primordial dans l'algorithme est que la méthode searchImplicitTupleWithMinimalCost() a une complexité polynomiale et qu'elle n'énumère pas plus de $t.r$ tuples.

Proposition 2 searchImplicitTupleWithMinimalCost(), *Méthode 8, est polynomiale et sa boucle* while *s'exécute au plus* $t + 1$ *fois.*

Preuve Dans la méthode searchImplicitTupleWithMinimalCost(), la méthode buildSuccessors(), Méthode 9, est seulement appelée lorsque le tuple courant τ est trouvé dans les tuples explicites. Puisqu'il y a t tuples explicites et puisqu'un tuple ne peut pas être inséré deux fois dans la queue, il y a au plus $t + 1$ appels à la méthode buildSuccessors(). Chacun de ces appels ajoute r tuples dans la queue de priorité R. De ce fait, la boucle *while* de la méthode searchImplicitTupleWithMinimalCost() énumère au plus $t.r$ tuples. Évidemment, le reste de cette méthode est également polynomial. □

3.8.2 Complexité

Nous discutons à présent des complexités spatiales et temporelles de GAC*-WSTR pour une contrainte donnée c_S. Avec $r = |S|$ l'arité, t le nombre de tuples dans la table associée à c_S et d la taille du plus grand domaine des variables de son scope, les complexités spatiales des structures de données utilisées dans GAC*-WSTR sont données dans la table 3.1.

Nous présentons la complexité temporelle dans le pire des cas de nos algorithmes.

Proposition 3 *La complexité temporelle dans le pire des cas de la méthode* findSupports(), *Méthode 1, est :*

- $O(r(d + rt))$ *quand le coût par défaut est* k
- $O(r(d + r(t + d)))$ *quand le coût par défaut est* 0
- $O(r(d + r(t + d(\log(d) + rt(\log(t) + r)))))$ *dans les autres cas*

Preuve

Les premières méthodes updateMinCost() et nbImplicitTuplesWith(), Méthodes 4 et 6, sont en temps constant. Nous discutons à présent de la complexité de la méthode findSupports(), Méthode 1. L'initialisation des structures de données S^{val}, S^{sup} et $minCosts$ est $O(rd)$. Ensuite, la méthode findSupports() effectue un appel à tableScan(), Méthode 2, et boucle alors au plus pour chaque variable appartenant au scope de la contrainte c_S (c'est à dire au plus r fois). En effet aucune variable n'est ajoutée dans

Structure de données	Complexité spatiale
$c_S.S$	$O(r)$
$c_S.table$	$O(tr)$
$c_S.costs$	$O(t)$
$c_S.deltas$	$O(rd)$
$c_S.minCosts$	$O(rd)$
$c_S.position$	$O(t)$
$c_S.S^{sup}$	$O(r)$
$c_S.S^{val}$	$O(r)$
$c_S.lastSize$	$O(r)$
$c_S.nbGacValues$	$O(r)$
$c_S.nbValidTuples$	$O(r)$
$c_S.nbExplicitValidTuples$	$O(r)$
$c_S.sortedDomains$	$O(rd)$

TAB. 3.1 – Complexité spatiale dans le pire des cas des structures de données utilisées par GAC*-WSTR pour une contrainte donnée c_S.

S^{sup} après l'étape d'initialisation. À chaque tour, une variable est considérée : pour chaque valeur du domaine de la variable, c'est à dire d fois, une projection peut être effectuée ainsi qu'un appel à la méthode tableScan(). Globalement, la complexité temporelle dans le pire des cas de findSupports() est $O(rd + O(\text{tableScan}()) + r(d + O(\text{tableScan}())))$, c'est à dire $O(rd + rO(\text{tableScan}()))$.

Il est nécessaire d'étudier la complexité temporelle de la méthode tableScan() selon les trois cas possibles de valeur de coût par défaut. La première partie de l'algorithme (trouver les coûts minimaux parmi les tuples explicites) est $O(rt)$ puisque pour chaque tuple nous devons :

- calculer le coût du tuple courant qui est $O(r)$
- vérifier si le tuple courant est valide ou non qui est $O(r)$ ou $(O(1)$ en fonction de la taille S^{val}. Notons que durant le premier appel à la méthode tableScan(), S^{val} contient au plus r variables tandis que S^{val} est vide durant le second appel et ceux qui suivent (s'il y en a).
- mettre à jour pour chaque variable de S^{sup} la structure $c_S.minCosts$ qui est $O(r)$.

La complexité globale de la première partie de la méthode tableScan() reste $O(rt)$.

- Lorsque le coût par défaut est k, la méthode tableScan() se termine et sa complexité temporelle est $O(rt)$. La complexité globale de la méthode findSupports() dans ce cas est $O(r(d + rt))$.
- Lorsque le coût par défaut est 0, un appel à handleZeroDefaultCost(), Méthode 5, est effectué en plus par la méthode tableScan(). La complexité temporelle de la méthode handleZeroDefaultCost() est $O(rd)$. Lorsque le coût par défaut est 0, la complexité temporelle globale de la méthode tableScan() est $O(r(t + d))$ et celle de la méthode findSupports() est $O(r(d + r(t + d)))$.
- Lorsque le coût par défaut se situe entre 0 et k, la méthode tableScan() réalise en plus un appel à handleIntermediateDefaultCost(), Méthode 7. Dans la méthode handleIntermediateDefaultCost(), l'initialisation de la queue F et le calcul de la valeur lb représentent $O(rd)$. Ensuite pour chaque variable nous devons trier le domaine, ce qui représente $O(rd \log(d))$. Enfin, F contient au plus rd valeurs et pour chaque valeur, un appel à searchImplicitTupleWithMinimalCost(), Méthode 8, est effectué et le coût minimal des r variables doit être mis à jour. Nous avons ainsi une complexité globale de $O(rd(\log(d)+r+O(\text{searchImplicitTupleWithMinimalCost}()))$ pour la méthode handleIntermediateDefaultCost().

La méthode searchImplicitTupleWithMinimalCost() a été prouvée polynomiale (Proposition 2) avec au plus t tours de boucle. À chaque tour une recherche binaire ($O(r\log(t))$) et un appel à buildSuccessors(), Méthode 9, ($O(r^2)$) sont effectués dans le pire des cas. La complexité temporelle de la méthode searchImplicitTupleWithMinimalCost() est $O(r + t(r\log(t) + r^2))$, c'est à dire $O(rt(\log(t) + r))$. La complexité temporelle de la méthode handleIntermediateDefaultCost() est $O(rd(\log(d) + rt(\log(t) + r)))$.
Lorsque le coût par défaut se situe entre 0 et k, la complexité temporelle globale de la méthode tableScan() est $O(r(t + d(\log(d) + rt(\log(t) + r))))$ et celle de la méthode findSupports() est $O(r(d + r(t + d(\log(d) + rt(\log(t) + r)))))$.

□

Pour terminer, nous présentons la complexité temporelle dans le pire des cas de notre algorithme principal GAC*-WSTR.

Proposition 4 *La complexité temporelle dans le pire des cas de l'algorithme GAC*-WSTR, Algorithme 28, est* $O(n^2d^2 + der(O(\text{findSupports}())))$.

Preuve unaryProject() et updateMaxUCostOf() représentent chacune une complexité en temps $O(d)$. La "map" `touchedVariables` pouvant contenir n variables, la complexité des lignes 6 à 9 est donc $O(nd)$. La fonction pruneVar() ayant une complexité en temps $O(d)$, la complexité des lignes 10 à 19 est donc $O(nd)$ car n variables peuvent être non assignées et parcourues à la ligne 12. La complexité des lignes 6 à 19 est donc $O(nd)$. La boucle principale *while* de GAC*-WSTR peut être exécutée nd fois (le domaine des n variables peut être modifié d fois). De ce fait, la complexité en temps dans GAC*-WSTR des lignes 6 à 19 est $O(n^2d^2)$. La complexité en temps des lignes 1 à 4 est $O(der)$ car les e contraintes impliquent au plus r variables dans leur scope (et non pas les n variables) et le domaine de ces r variables peut être modifié d fois (c'est pourquoi la ligne 4 est exécutée $e\boldsymbol{rd}$ fois au lieu de $e\boldsymbol{nd}$). La complexité de la ligne 5 étant $O(\text{findSupports}())$, avec la méthode findSupports() appelée qui dépend de la valeur du coût par défaut, la complexité en temps des lignes 1 à 5 est donc $O(der(O(\text{findSupports}())))$. Par conséquent, la complexité totale de GAC*-WSTR est $O(n^2d^2 + der(O(\text{findSupports}())))$. □

Le tableau 3.2 récapitule les complexités en temps dans le pire des cas des algorithmes GAC3* (Algorithme 26) et GAC*-WSTR (Algorithme 28) pour établir GAC*. On observe bien la polynomialité de notre approche GAC*-WSTR dans tous les cas (c'est à dire quelle que soit la valeur du coût par défaut) par rapport à l'algorithme de base pour établir GAC* qui n'est polynomial que lorsque la taille de l'entrée est en $O(d^r)$ et qui est exponentiel lorsque l'entrée est un polynôme de r, d et n.

Algorithme	Coût par défaut	Complexité en temps
GAC3*	$[0, k]$	$O(n^2d^2 + der(r(d^r + d)))$
GAC*-WSTR	k	$O(n^2d^2 + der(r(d + rt)))$
GAC*-WSTR	0	$O(n^2d^2 + der(r(d + r(t + d))))$
GAC*-WSTR	$]0, k[$	$O(n^2d^2 + der(r(d + r(t + d(\log(d) + rt(\log(t) + r))))))$

TAB. 3.2 – Complexités temporelles dans le pire des cas des algorithmes GAC3* (Algorithme 26) et GAC*-WSTR (Algorithme 28) pour établir l'arc cohérence généralisée, en fonction de la valeur du coût par défaut des contraintes tables souples traitées.

Après avoir étudié les propriétés théoriques de notre approche, nous présentons dans les deux sections suivantes le protocole et les résultats de notre évaluation expérimentale.

3.9 Familles de problèmes

L'algorithme que nous proposons peut être appliqué sur n'importe quelle contrainte table souple, c'est à dire quelle que soit sa valeur de coût par défaut. Nous avons pu constater que, parmi les 31 familles d'instances WCSP listées sur http://costfunction.org, les familles contenant des instances dans lesquelles les contraintes tables souples (de faible arité et majoritairement binaires) possèdent un coût par défaut égal soit à 0, soit à k, apparaissent bien plus fréquemment en pratique. Par conséquent, nous avons eu besoin de générer des instances contenant des contraintes de grande arité d'une part, et pour lesquelles le coût par défaut se situe entre 0 et k exclus d'autre part, afin de tester notre approche.

3.9.1 Instances avec un coût par défaut égal à 0 ou à k

Dans cette section, nous décrivons d'abord les instances composées de contraintes tables souples avec un coût par défaut égal à 0 ou k, de plus ou moins grande arité, que nous avons (après les avoir générées si besoin) utilisées pour nos expérimentations.

Nous avons procédé à une première expérimentation sur une nouvelle série d'instances de mots-croisés (*crossword*), appelée *crossoft*, qui se prêtent naturellement à une représentation par des contraintes tables souples. Étant donné une grille et un dictionnaire, l'objectif est de remplir la grille avec des mots puisés dans le dictionnaire. Pour générer ces instances, nous avons utilisé trois séries de grilles (Herald, Puzzle, Vg) et un dictionnaire, appelé OGD, qui contient des noms communs (avec un coût de 0 associé à chaque mot) et des noms propres (avec un coût associé de r, r correspondant à la longueur du mot). Ces pénalités sont inspirées de la notion de profits associés aux mots décrits sur le site internet français http://ledefi.pagesperso-orange.fr.

Nous avons réalisé une deuxième expérimentation sur des instances WCSP aléatoires. Nous avons généré différentes classes d'instances en considérant le modèle CSP RB [Xu *et al.*, 2007]. Avec les paramètres bien choisis, le théorème 2 dans [Xu *et al.*, 2007] est vérifié pour le cadre CSP : une phase de transition asymptotique est garantie à un point de seuil précis. Des instances CSP obtenues à partir du modèle RB ont également été transformées en instances WCSP en associant un coût aléatoire (entre 1 et k) pour chaque tuple CSP interdit, et en considérant un coût par défaut égal à 0 pour les tuples implicites. On peut naturellement supposer que les instances WCSP ainsi obtenues sont au moins aussi difficiles que les instances CSP d'origine. En fixant $k = 10$, nous avons généré plusieurs séries d'instances WCSP d'arité 3 ; rb-r-n-d-e-t-s est une instance d'arité r avec n variables, une taille de domaine d et e r-aires contraintes de *tightness* t et générées avec une graine s (*seed*).

Ensuite, nous avons transformé des instances CSP en instances WCSP en associant un coût aléatoire entre 1 (inclus) et 10 (inclus) aux tuples explicites présents dans les contraintes. Les valeurs 1 et 10 ont été choisies arbitrairement. Pour les tuples implicites, un coût par défaut a été fixé à 0 ou k (avec $k > 10$) en fonction des séries et plus précisément des contraintes. En effet, pour une contrainte table positive à l'origine, le coût par défaut a été fixé à k, tandis qu'il a été fixé à 0 pour une contrainte table négative. Un premier ensemble a été obtenu en transformant des instances CSP aléatoires (avec des arités de contraintes égales à 8 et 10) en instances WCSP, appelées *rand-8* et *rand-10*. Le coût par défaut utilisé pour les tuples implicites a été fixé à k. Un deuxième ensemble, appelé *crossword-conv*, a été obtenu en transformant des instances CSP *crossword* en WCSP avec un coût par défaut égal à k. Une telle transformation a également été utilisée pour les séries *renault-mod*. Pour ces instances, des contraintes ont un coût par défaut égal à 0 et les autres ont un coût par défaut égal à k.

Enfin, nous avons expérimenté GAC*-WSTR sur des séries du monde réel provenant du site internet http://costfunction.org/en/benchmark. Bien que notre approche soit clairement adaptée aux contraintes souples de grande arité, il était intéressant malgré tout d'observer son comportement sur des contraintes de faible arité. Pour cela, nous avons utilisé les séries *ergo*, *linkage*, *celar* et *spot5*

([Bensana *et al.*, 1999]) qui représentent des instances WCSP structurées avec des contraintes de faible arité.

3.9.2 Instances avec un coût par défaut différent de 0 et de k

Dans cette section, nous décrivons cette fois-ci des instances avec des contraintes tables souples avec un coût par défaut intermédiaire, de plus ou moins grande arité, également utilisées pour tester notre approche.

Nous avons généré une deuxième série d'instances *crossoft* obtenue à partir d'instances CSP *Crossword* insatisfaisables. Les instances CSP ont été générées depuis deux séries de grilles (Herald, Vg) et d'un dictionnaire, appelé LEX, qui contient des noms autorisés. Pour obtenir des instances *Crossword* insatisfaisables, nous avons réduit le nombre de mots pour différentes tailles de mots. Nous avons arbitrairement réduit le nombre de mots de taille 4 (397 noms conservés sur 2176), le nombre de mots de taille 5 (309/3145) et le nombre de mots de taille 6 (1112/3852). Ainsi, des instances WCSP *crossoft* ont été construites avec des contraintes tables souples dans lesquelles les tuples implicites contiennent tous les mots interdits (avec un coût par défaut de 1) et les tuples explicites correspondent aux mots autorisés avec un coût de 0. Les instances *Crossoft* générées pour nos expérimentations possèdent des contraintes tables souples d'arité comprise entre 4 et 6.

Une deuxième famille d'instances, appelée *Kakuro*, a été générée. Kakuro est un puzzle logique rendu populaire par la compagnie japonaise Nikoli. Étant donné une grille composée de cases noires et blanches, l'objectif est de remplir les cases blanches avec un nombre entre 1 et 9 de telle manière que la somme de chaque séquence horizontale ou verticale soit égale à la somme potentiellement imposée par la case noire adjacente et que tous les entiers utilisés dans la séquence soient différents. La figure 3.13 représente un exemple de puzzle Kakuro résolu.

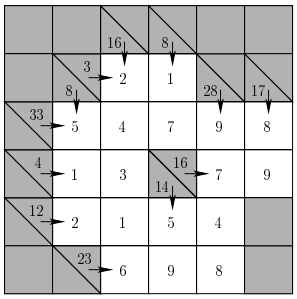


FIG. 3.13 – Solution d'un puzzle Kakuro

Afin d'obtenir des instances Kakuro insatisfaisables, nous avons modifié pour chaque instance la valeur d'une somme imposée dans une case noire. Les instances Kakuro générées pour nos expérimentations possèdent des contraintes tables souples d'arité comprise entre 2 et 9. Notons que les instances Kakuro ont été organisées dans trois séries (easy, medium et hard) qui dépendent de leur difficulté à être résolue par un humain et non par un solveur. C'est pourquoi dans les expérimentations ces catégories ne correspondent pas réellement à trois catégories de difficulté croissante (pour un solveur).

Enfin, nous avons expérimenté notre approche sur une série d'instances *Nonogram*. Étant donné une grille vide, l'objectif est de colorier certaines cases de telle manière que les séquences de cases coloriées ou non coloriées correspondent aux *patterns* (modèles) précisés sur chaque ligne et sur chaque colonne. La figure 3.14 représente un exemple de Nonogram résolu. Afin d'obtenir des instances Nonogram insatisfaisables, nous avons inversé les patterns de deux lignes. Par exemple, le pattern $p = [5, 2, 3, 4]$ sera remplacé par le pattern $p' = [4, 3, 2, 5]$. Les instances Nonogram générées pour nos expérimentations

possèdent des contraintes tables souples d'arité comprise entre 16 et 29.

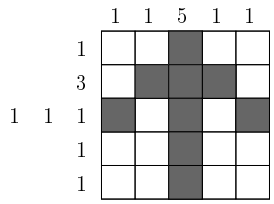


FIG. 3.14 – Solution d'un puzzle Nonogram

3.10 Résultats expérimentaux

Afin de valider expérimentalement notre approche pour filtrer les contraintes tables souples de grande arité, nous avons mené des expérimentations (avec notre solveur $AbsCon$) en utilisant un cluster Bi Xeon 3.0GHz avec 2GiB de RAM sous Linux. Pour résoudre des instances avec des contraintes tables souples, nous avons d'abord implémenté une version de PFC-MRDAC (présentée dans la section 2.3.5) où les coûts minimaux (requis par l'algorithme pour calculer les bornes inférieures) sont obtenus en appelant la fonction tableScan(), Méthode 2. À chaque étape de la recherche, seul un appel à la fonction tableScan() est nécessaire pour chaque contrainte table souple puisque PFC-MRDAC n'exploite pas les opérations de transfert de coûts. Cette version sera appelée PFC-MRDAC-WSTR, tandis que la version classique, dans laquelle les coûts minimaux sont cherchés parmi les tuples de manière naïve, sera appelée ici PFC-MRDAC-STD (STD pour standard).

Nous avons aussi implanté l'algorithme GAC*-WSTR et nous l'avons incorporé dans un algorithme de recherche arborescente en profondeur d'abord et retour en arrière. Cet algorithme de recherche peut également maintenir GAC* via l'algorithme classique 26 présenté précédemment, et itérer, comme nous l'avons expliqué, tous les tuples valides afin de calculer les coûts minimaux et les bornes inférieures. Pour contrôler ces itérations (qui peuvent être exponentielles en fonction de l'arité des contraintes), nous avons fixé de façon pragmatique un paramètre qui retarde l'application de l'algorithme jusqu'à ce que assez de variables dans le scope des contraintes soient assignées. En pratique, l'algorithme se décline en trois versions : GAC$^*_2$ appliqué à une contrainte lorsque au plus deux variables de sa portée ne sont pas assignées, GAC$^*_3$ lorsque au plus trois variables ne sont pas assignées et GAC$^*_u$ qui est appliqué systématiquement pour chaque contrainte.

Enfin, nous avons comparé notre approche avec les autres cohérences de l'état de l'art que sont FDAC et EDAC. Les utilisations de ces méthodes ont également été retardées de façon similaire à GAC*.

Nous avons conduit une première série d'expérimentations sur les familles de problèmes décrites dans la section précédente et nous avons comparé les résultats obtenus par les différentes variantes de PFC-MRDAC et GAC*. Un temps limite de $1,200$ secondes a été fixé par instance. L'heuristique de choix de variable utilisée est $dom/ddeg$ [Bessiere et Régin, 1996] et l'heuristique de choix de valeur consistait à choisir en priorité la valeur de coût (unaire) minimale (qui est toujours à 0 pour GAC*). Les résultats globaux sont donnés dans le tableau 3.3. Dans le tableau, les instances sont triées selon le coût par défaut. Chaque ligne de ce tableau correspond à une série d'instances : *celar*, *spot5*, *rand-3*,... Le nombre total d'instances pour chaque série est donné dans la seconde colonne du tableau, et pour chacune de ces séries, les arités des contraintes qu'elles contiennent sont indiquées dans la troisième colonne. Pour chacune des séries, nous fournissons le nombre d'instances résolues (optimum prouvé) par chaque méthode en 20 minutes.

Parmi les quatre variantes de GAC*, GAC$^*_2$ et GAC$^*_u$ sont globalement les moins bonnes. Pour la

Series	#Inst	Arités	PFC-MRDAC-WSTR	PFC-MRDAC-STD	Maintient-GAC*-WSTR	Maintient-GAC$^*_2$	Maintient-GAC$^*_3$	Maintient-GAC$^*_u$
Coût par défaut = 0								
celar (celar6 and celar7)	15	2	4	**5**	4	**6**	**6**	**6**
spot5	21	2 et 3	2	**3**	**3**	**3**	**3**	**3**
rand-3 (rb)	48	3	22	**31**	21	**33**	32	32
linkage	22	1 à 5	**0**	**0**	**0**	**0**	**0**	**0**
ergo	19	1 à 18	**13**	12	**14**	11	12	13
Coût par défaut = $+\infty$								
crossword-conv	16	5 à 9	**8**	0	**8**	0	0	0
rand-8	20	8	**2**	0	1	**17**	10	0
rand-10	20	10	**20**	0	**20**	0	0	0
crossoft-puzzle	22	3 à 13	**22**	9	**22**	7	9	9
crossoft-vg	65	4 à 20	**12**	6	**14**	4	4	6
crossoft-herald	50	3 à 23	**27**	10	**31**	9	10	10
Coût par défaut $\in]0, +\infty[$								
crossword	8	4 à 6	**7**	3	**6**	1	2	2
kakuro-easy	50	2 à 9	**17**	8	**29**	14	27	19
kakuro-medium	50	2 à 9	**20**	8	**26**	14	23	22
kakuro-hard	50	2 à 9	**24**	6	**32**	15	**32**	23
nonogram	25	16 à 29	**0**	**0**	**0**	**0**	**0**	**0**
Coût par défaut = 0 et $+\infty$								
renault-mod	50	2 à 10	**44**	32	**45**	37	38	40
Total			**244**	133	**276**	171	208	185

TAB. 3.3 – Nombre d'instances résolues par série (temps limite de 1,200 secondes fixé par instance).

variante GAC$^*_u$, ce n'est pas surprenant puisque trouver des supports sur des contraintes de grande arité est très coûteux. En effet, cela signifie itérer sur tous les tuples valides des contraintes pour calculer les bornes inférieures et le nombre d'itérations est exponentiel en fonction de l'arité des contraintes. Pour la variante GAC$^*_2$, ce n'est pas surprenant non plus car la capacité de filtrage est plus réduite en début de recherche que par rapport à la variante GAC$^*_3$. La variante retardée GAC$^*_3$, qui filtre donc plus tôt durant la recherche que GAC$^*_2$, obtient globalement de meilleurs résultats que les versions GAC$^*_2$ et GAC$^*_u$.

Nos approches GAC*-WSTR et PFC-MRDAC-WSTR résolvent plus d'instances que les algorithmes standards, plus particulièrement pour les séries d'instances de grande arité. Par exemple, les séries *rand-10* et *crossword-conv* sont résolues uniquement par les approches STR. Sans surprise, les approches STR ne sont pas si efficaces sur les séries rb (*rand-3*) et *celar*, ce qui peut être expliqué par l'arité faible des contraintes impliquées dans ces instances.

Nous avons mené une deuxième série d'expérimentations afin de comparer notre approche GAC*-WSTR avec certaines autres cohérences (plus fortes) comme EDAC et FDAC. Les résultats sont donnés dans le tableau 3.4. Le protocole expérimental est le même que celui utilisé pour la première série d'ex-

périmentations. À noter que pour les approches EDAC et FDAC nous évoquons uniquement les variantes retardées $EDAC_3$ et $FDAC_3$ puisque les variantes non limitées et les autres variantes retardées sont surpassées (voir la conclusion de la première série d'expérimentations). Il est difficile de conclure avec les résultats expérimentaux obtenus. Notre approche a résolu plus d'instances que $EDAC_3$ et $FDAC_3$ pour la moitié des séries. Concernant l'autre moitié des séries les approches de l'état de l'art sont meilleures. Malgré tout, une comparaison de ces différentes approches reste difficile puisqu'elles n'établissent pas la même cohérence et de ce fait, nous savons déjà que les capacités de filtrage sont supérieures pour les cohérences FDAC* et EDAC* par rapport à AC*. Une perspective intéressante serait d'intégrer la réduction tabulaire simple STR pour établir les cohérences FDAC* et EDAC* et proposer ainsi les algorithmes (G)FDAC*-WSTR et (G)EDAC*-WSTR. Cependant, même si FDAC* et EDAC* (ainsi que VAC) sont considérées en pratique comme les meilleures méthodes pour résoudre les WCSP, ces résultats prouvent qu'un algorithme classique (mais paramétré et optimisé via l'utilisation de STR) établissant l'arc-cohérence souple GAC* reste compétitif.

Series	#Inst	Arités	Maintient-GAC*-WSTR	$FDAC_3$	$EDAC_3$
Coût par défaut = 0					
celar (celar6 and celar7)	15	2	4	**9**	**9**
spot5	21	2 et 3	3	**5**	**5**
rand-3 (rb)	48	3	21	**32**	1
linkage	22	1 à 5	**0**	**0**	**0**
ergo	19	1 à 18	14	**16**	13
Coût par défaut = $+\infty$					
crossword-conv	16	5 à 9	**8**	0	0
rand-8	20	8	1	**10**	**10**
rand-10	20	10	**20**	0	0
crossoft-puzzle	22	3 à 13	**22**	9	9
crossoft-vg	65	4 à 20	**14**	4	4
crossoft-herald	50	3 à 23	**31**	10	10
Coût par défaut $\in]0, +\infty[$					
crossword	8	4 à 6	**6**	2	2
kakuro-easy	50	2 à 9	29	**39**	**39**
kakuro-medium	50	2 à 9	26	**39**	**39**
kakuro-hard	50	2 à 9	32	**41**	**41**
nonogram	25	16 à 29	**0**	**0**	**0**
Coût par défaut = 0 and $+\infty$					
renault-mod	50	2 à 10	**45**	39	0
Total			**276**	255	182

TAB. 3.4 – Nombre d'instances résolues par série (temps limite de 1,200 secondes fixé par instance).

Le tableau 3.5 se focalise sur certaines instances sélectionnées avec la même comparaison d'algo-

rithmes. Nous proposons un aperçu des résultats en termes de temps CPU (en secondes). Comme pour les autres expérimentations, un temps limite de $1,200$ secondes a été fixé par instance. Les instances ont été sélectionnées dans le but de refléter la tendance qui émane des séries d'expérimentations précédentes (voir tableaux 3.3 et 3.4).

Instances	PFC-MRDAC-WSTR	PFC-MRDAC-STD	Maintient-GAC*-WSTR	Maintient-GAC$^*_3$	Maintient-FDAC$_3$	Maintient-EDAC$_3$
celar6-sub0	203	**66.7**	123	24.2	**1.22**	1.25
spot5-29	$> 1,200$	**1,151**	447	236	**1.78**	1.87
rb-3-20-20-60-p2944-7	$> 1,200$	**165**	$> 1,200$	**99**	122	$> 1,200$
munin-2	$> 1,200$	$> 1,200$	840	$> 1,200$	256	**179**
crossword-m1c-words-vg5-6	**982**	$> 1,200$	**1,033**	$> 1,200$	$> 1,200$	$> 1,200$
rb-8-20-5-18-800-11	$> 1,200$	$> 1,200$	$> 1,200$	**907**	1,040	1,046
rb-10-20-10-5-10000-0	**2.08**	$> 1,200$	**1.93**	$> 1,200$	$> 1,200$	$> 1,200$
crossoft-ogd-15-01	**66.4**	$> 1,200$	**181**	$> 1,200$	$> 1,200$	$> 1,200$
crossoft-ogd-puzzle10	**1.06**	$> 1,200$	**1.87**	$> 1,200$	$> 1,200$	$> 1,200$
crossoft-ogd-vg-5-6	**1.13**	118	**2**	$> 1,200$	$> 1,200$	$> 1,200$
crossword-m1c-LEX-vierge4-4	**0.78**	5.1	**1.2**	411	426	423
kakuro-easy-012	**192**	$> 1,200$	594	715	52.9	**48.5**
kakuro-medium-016	$> 1,200$	$> 1,200$	$> 1,200$	$> 1,200$	18.2	**14.9**
kakuro-hard-024	$> 1,200$	$> 1,200$	461	591	**122**	134
nonogram-gp-106	$> 1,200$	$> 1,200$	$> 1,200$	$> 1,200$	$> 1,200$	$> 1,200$
renault-mod-10	**802**	$> 1,200$	**641**	$> 1,200$	$> 1,200$	$> 1,200$

TAB. 3.5 – Temps CPU (en secondes) pour prouver l'optimalité sur des instances sélectionnées variées (temps limite de 1,200 secondes fixé par instance).

Chapitre 4

Résolution WCSP par l'extraction de noyaux insatisfaisables minimaux

Sommaire

Nous présentons dans cette section les travaux constituant notre seconde contribution. Ce travail a fait l'objet d'une publication dans [Lecoutre *et al.*, 2013]. Nous présentons également de nouveaux résultats non encore publiés.

4.1 Problématique

Comme nous l'avons vu dans le chapitre 1, le problème de satisfaction de contraintes (CSP) consiste à déterminer si un réseau de contraintes (CN) donné, ou instance CSP, est satisfaisable ou non. Il s'agit d'un problème de décision, où l'on cherche une instanciation complète satisfaisant toutes les contraintes. Lorsque des préférences doivent être représentées, il est possible de les exprimer à l'aide de contraintes souples. Dans le cadre WCSP (Weighted CSP), présenté dans le chapitre 2, chaque contrainte souple associe un coût aux différentes instanciations des variables sur lesquelles elle porte, permettant ainsi de

modéliser différents degrés de violation. L'objectif est alors de trouver une instanciation qui minimise le coût combiné des différentes contraintes souples.

Nous avons constaté également dans le chapitre 2 que la plupart des méthodes actuelles de résolution de réseaux de contraintes pondérées (WCN pour Weighted Constraint Network), ou instances WCSP, se fondent sur une recherche par séparation et évaluation (Branch and Bound) et utilisent diverses cohérences locales souples pour estimer le coût minimum du réseau simplifié au cours de la recherche. Ces cohérences locales (AC*, FDAC*, EDAC*, VAC, OSAC), présentées dans la section 2.3.2, sont fondées sur des méthodes de transferts de coût qui préservent l'équivalence. Il faut savoir que les algorithmes établissant les cohérences locales souples sur les WCSP sont souvent plus complexes que leurs équivalents CSP car ils doivent prendre en compte plus d'informations, en l'occurrence les coûts attribués par les contraintes souples.

Pour cette deuxième contribution, nous proposons une approche originale pour le cadre WCSP. Elle consiste à résoudre un WCN donné en résolvant une séquence de CN générés itérativement à partir de ce WCN, ce qui permet ainsi la ré-utilisation de solveurs de contraintes éprouvés incorporant notamment les différentes techniques de résolution présentées dans la section 1.3. À partir du WCN d'origine, les CN sont obtenus en durcissant les contraintes souples, c'est à dire en ne retenant que les tuples ayant un coût donné. Le principe de notre méthode est alors d'énumérer ces CN, et lorsqu'un CN est insatisfaisable lors de cette énumération, un noyau insatisfaisable minimal (MUC pour Minimal Unsatisfiable Core) est identifié pour déterminer les contraintes souples pour lesquelles il faut être prêt à accepter un coût plus important afin d'obtenir une solution. Cette approche est déclinée en un algorithme glouton (et donc incomplet) et deux algorithmes complets. Notons que cette approche a été utilisée avec succès pour le cadre MaxSAT [Fu et Malik, 2006, Ansótegui *et al.*, 2010, Marques-Sila et Planes, 2011].

Ce type d'approche (intégrer des techniques CSP afin de résoudre des instances WCSP) a déjà été utilisée avec succès pour établir la cohérence souple VAC présentée dans la section 2.3.2. Cela consiste à établir (une ou plusieurs fois) la cohérence d'arc sur un CN généré à partir d'un WCN dans le but, contrairement à notre approche, d'identifier des transferts de coûts possibles. Dans cette approche, les CN sont également obtenus en durcissant les contraintes souples du WCN, cependant seuls les tuples ayant un coût nul sont autorisés dans le CN obtenu. Notre transformation de WCN en CN est différente dans le sens où, au delà de n'autoriser uniquement que les tuples de coût nul comme dans l'approche utilisée pour la cohérence VAC, il va être possible de définir le ou les coûts (pas forcément nuls) des tuples qui seront autorisés au sein de chaque contrainte souple.

Ce chapitre est construit comme suit : après quelques définitions, nous présentons les structures de données utilisées ainsi que le fonctionnement général des algorithmes. Nous détaillons alors une première version complète ainsi qu'une version gloutonne de cette approche. Nous présentons ensuite une deuxième approche complète basée sur la première et dans laquelle nous proposons et nous apportons certaines améliorations qui nous ont permis de la rendre plus compétitive. Enfin, nous accompagnons ces différentes approches par quelques résultats expérimentaux.

4.2 Méthodes de l'état de l'art

Bien évidemment, les approches complètes actuelles de résolution WCSP basées sur une recherche par séparation et évaluation et sur le maintien de cohérences locales souples représentent les méthodes de référence de l'état de l'art pour la résolution de WCSP. Elles sont particulièrement efficaces pour les contraintes binaires et ternaires comme nous l'avons vu dans l'état de l'art général et dans la contribution précédente. Nous ne reviendrons donc pas ici sur ces méthodes déjà décrites dans le chapitre 2. De même, comme l'ont montré [Lee et Leung, 2009, Lee et Leung, 2010], elles peuvent également être exploitées pour un ensemble de contraintes globales souples pouvant s'exprimer sous la forme de problèmes de

flot. Les approches incomplètes de référence, ayant également déjà été présentées dans la section 2.5, ne seront pas rappelées non plus ici. Nous comparerons nos approches (complètes et incomplètes) à ces méthodes de référence implantées dans $ToulBar2$ (complètes) et $INCOP$ (bibliothèque de méthodes incomplètes incorporée dans $ToulBar2$). De plus, il est à noter que durant notre travail sur cette contribution, une approche prometteuse assez semblable à la nôtre a également été proposée de manière indépendante dans [Delisle et Bacchus, 2013]. Dans cette approche, il est question de résoudre là aussi un WCSP par une succession de relaxations de CN. Nous reviendrons plus en détail sur cette approche dans la section 4.8.

4.3 Strates, contraintes tables souples stratifiées et fronts

Dans notre étude, afin de simplifier la présentation, nous considérons les contraintes souples représentées en extension, c'est à dire des contraintes tables souples (voir définition 40), cependant l'approche s'applique aussi à de l'intention. Nous commençons par introduire les notions de *strate*, *contrainte table souple stratifiée* et *front* sur lesquelles sont basées les approches que nous proposons.

Les strates représentent les différents degrés de violation possibles au niveau d'une contrainte. Tous les tuples ayant le même coût sur une contrainte souple peuvent être regroupés dans un même sous-ensemble nommé *strate*. Une strate particulière est associée au coût par défaut : cette strate ne contient aucun tuple, mais représente implicitement tous les tuples qui n'apparaissent pas explicitement dans une autre strate. Au sein d'une contrainte, nous considérons les strates ordonnées par coûts strictement croissants. De la même manière que dans la section 3.4.3, nous avons choisi dans nos algorithmes une représentation orientée objet des contraintes tables. Étant donné que pour cette contribution nous avons également travaillé avec des contraintes tables dures, commençons par introduire la représentation objet de ces contraintes, à savoir la classe HardTableConstraint (illustrée par `Class HardTableConstraint` dans ce manuscrit) que nous avons choisie. En complément des attributs correspondant au scope (`S`) et à la table (`table`) de ces contraintes, nous avons également un attribut `semantics` pour définir si la relation associée à la contrainte est de type support (`SUPPORTS`) ou conflit (`CONFLICTS`).

Class HardTableConstraint

Attributs :

`S` : set of r variables ; `// portée`

`table` : array of t tuples ; `// tuples explicites`

`semantics` : enumeration (`SUPPORTS` or `CONFLICTS`) ; `// sémantique de la table`

Constructeur :

HardTableConstraint(set s, array t, enumeration e) ; `// S` $\leftarrow s$, `table` $\leftarrow t$, `semantics` $\leftarrow e$

Concernant les contraintes tables souples, nous avons bien évidemment suivi la même représentation objet que celle proposée dans 3.4.3, à savoir pour rappel la classe SoftTableConstraint. Cependant, afin de pouvoir gérer les strates dans une contrainte, nous proposons la classe LayeredSoftTableConstraint (illustrée par `Class LayeredSoftTableConstraint` dans ce manuscrit) représentant une contrainte table souple stratifiée. D'une part, comme la classe SoftTableConstraint, cette classe possède les attributs `S` et `defaultCost`, d'autre part les attributs `table` et `costs` proposés dans la classe SoftTableConstraint sont regroupés ici dans l'attribut supplémentaire `layers` qui correspond à l'ensemble des strates de la contrainte. Il s'agit plus précisément d'un tableau contenant des objets de la classe Layer (illustrée par `Class Layer` dans ce manuscrit) : un objet Layer est caractérisé par un coût (`cost`) et regroupe un ensemble de tuples (`tuples`) dont le coût est égal à `cost`. À noter que `tuples` est vide pour l'objet Layer correspondant au coût par défaut. Le nombre de strates d'une contrainte w est désigné par

`w.layers.length`. Par souci de simplicité, nous utiliserons des indices (de 0 à `w.layers.length` $-$ 1) pour identifier les différentes strates d'une contrainte, et quand le contexte est suffisamment clair, nous confondrons les indices avec les strates qu'ils représentent.

Class LayeredSoftTableConstraint	
Attributs :	
`S` : set of r variables ;	`// portée`
`defaultCost` : integer ;	`// coût de chaque tuple implicite`
`layers` : array of l Layers ;	`// tuples explicites regroupés par coût`

Class Layer	
Attributs :	
`tuples` : array of tuples ;	`// tuples explicites groupés par layer`
`cost` : integer ;	`// coût de chaque tuple explicite dans le layer`

Dans un souci de clarté, une contrainte souple sera confondue avec sa représentation sous la forme LayeredSoftTableConstraint.

Un *front* f est une fonction qui à chaque contrainte d'un WCN associe l'indice, à partir de 0, de l'une de ses strates. Un front représente une frontière entre les strates qui seront considérées (autorisées) à un instant donné de notre approche, et celles qui seront rejetées (interdites). Un front peut être vu comme une représentation indirecte d'un réseau de contraintes avec l'identification pour chaque contrainte des tuples autorisés (appartenant aux strates autorisées) et des tuples interdits (appartenant aux strates interdites). Nous utiliserons une notation de tableau pour les fronts : $f[w_i]$ représente la strate associée à la contrainte w_i par le front f. Nous noterons aussi un front sous la forme d'un tuple entre $<$ et $>$, à savoir $f :< s_0, \ldots, s_e >$ dans lequel par exemple s_0 représente la strate associée à la contrainte w_0 par le front f. Soit un WCN composé des contraintes w_0 et w_1, le front f, noté $f :< 1, 0 >$, associe alors à la contrainte w_0 sa strate 1 et il associe à la contrainte w_1 sa strate 0, et $f[w_0]$ représente par exemple la strate courante associée à la contrainte w_0 dans le front f, à savoir la strate 1. La figure 4.1 présente ces deux contraintes w_0 et w_1 avec leurs différentes strates. La contrainte w_0 possède un ensemble de strates énumérées à partir de 0 : la strate 0 de coût minimal $cout_0$ contient les tuples τ_1, τ_5 et τ_7, et la strate suivante (strate 1) de coût $cout_1$ contient les tuples τ_2 et τ_3. Dans cet exemple, nous considérons donc un front $f :< 1, 0 >$ tel que $f[w_0]$ est égal à 1 et $f[w_1]$ est égal à 0. Les strates associées par le front f aux contraintes w_0 et w_1 sont identifiées par un fond grisé dans la figure 4.1.

Le coût d'un front f, $cost(f)$, est obtenu en additionnant le coût des strates identifiées pour chaque contrainte par le front f :

$$cost(f) = \bigoplus_{w \in cons(W)} cost(f[w])$$

Il est important de noter que dans notre approche, nous ne considérons jamais de front f tel que $\exists w \in cons(W)$ avec $f[w]$ désignant la strate du coût interdit. En effet, atteindre un CN satisfaisable en ayant autorisé, pour une contrainte dure, des tuples qui sont totalement interdits pour la contrainte souple d'origine, n'a pas de sens. Le CN a beau être satisfaisable, il est clair que le WCN ne l'est pas car les tuples ayant permis de satisfaire le CN sont interdits dans le WCN.

Enfin, nous introduisons la relation de successeur direct entre les fronts. Un front f' est le successeur direct d'un front f si et seulement si il existe une contrainte w_i telle que $f'[w_i] = f[w_i] + 1$ et $\forall j \neq$

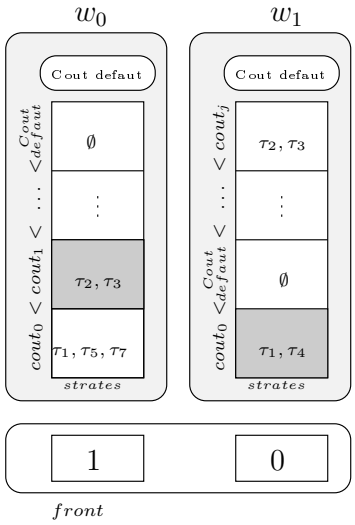


FIG. 4.1 – Front et strates pour deux contraintes w_0 et w_1.

$i, f'[w_j] = f[w_j]$. Nous notons $f \rightarrow f'$ si et seulement si f' est un successeur direct de f et $f \rightarrow_* f'$ si et seulement si il existe une séquence de relations de la forme $f \rightarrow f_1, f_1 \rightarrow f_2, \ldots f_n \rightarrow f'$ (fermeture transitive).

Comme les strates des contraintes sont ordonnées par ordre de coût croissant, nous pouvons en déduire que

$$f \rightarrow_* f' \Rightarrow (cost(f) < cost(f') \vee cost(f) = cost(f') = k)$$

Nous pouvons observer que l'ensemble de tous les fronts possibles et la relation de successeur direct forment une structure de *treillis*, dans laquelle le plus petit front (noté $\perp$) associe à chacune des contraintes sa première strate (de coût le plus faible) et le plus grand front (noté $\top$) associe à chaque contrainte sa dernière strate (de coût le plus grand). Les fronts $\perp$ et $\top$ représentent donc respectivement un minorant et un majorant de la structure de treillis. Le diagramme de Hasse de la figure 4.2 représente un exemple de treillis (sur la base d'une instance comportant trois contraintes ayant chacune deux strates).

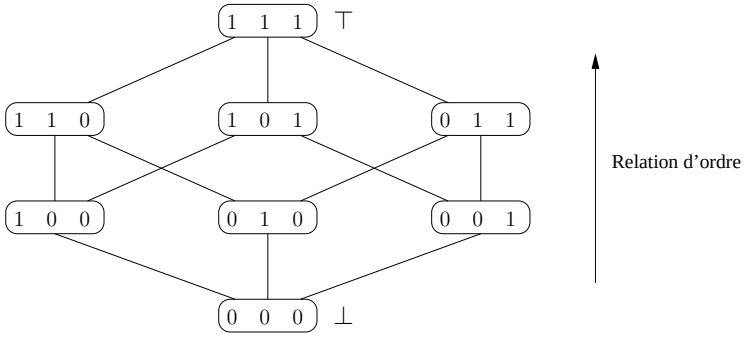


FIG. 4.2 – Diagramme de Hasse (structure de treillis).

Dans le treillis de la figure 4.2, le plus petit front est $< 0, 0, 0 >$, le plus grand front est $< 1, 1, 1 >$, et un front, par exemple $< 1, 0, 1 >$, est situé au dessus de deux autres fronts $< 1, 0, 0 >$ et $< 0, 0, 1 >$ car il est plus grand selon la relation d'ordre définie (en effet, $< 1, 0, 1 >$ est le successeur direct de $< 1, 0, 0 >$ et $< 0, 0, 1 >$). Nous observons aussi que par exemple $< 0, 1, 0 > \rightarrow < 0, 1, 1 >$ (le

front $< 0, 1, 1 >$ est un successeur direct du front $< 0, 1, 0 >$) et $< 0, 1, 0 > \rightarrow_* < 1, 1, 1 >$ (le front $< 1, 1, 1 >$ est un successeur du front $< 0, 1, 0 >$).

Notons cependant que nous utiliserons principalement dans nos approches une autre relation d'ordre qui utilise les coûts.

4.4 Principe général de résolution

La figure 4.3 décrit le fonctionnement général de l'approche que nous proposons dans cette contribution pour la résolution de réseaux de contraintes pondérées. Pour commencer, à partir d'un réseau de contraintes pondérées W, un réseau de contraintes initial P est construit de la façon suivante : pour chaque contrainte de W, les tuples appartenant à la strate de coût minimal sont considérés comme autorisés dans P, tandis que les autres tuples sont considérés comme interdits. Cette méthode de construction est similaire à celle proposée pour construire $Bool(P)$, une version durcie d'un réseau P, dans [Cooper *et al.*, 2008] (cohérence d'arc virtuelle VAC présentée dans la sous section 2.3.3 du manuscrit). Alors que $Bool(P)$ est construit uniquement sur la base des tuples ayant un coût égal ou différent de 0, notre version offre plus de liberté dans le sens où le coût autorisé est paramétrable. Ceci est possible à travers un front passé en paramètre.

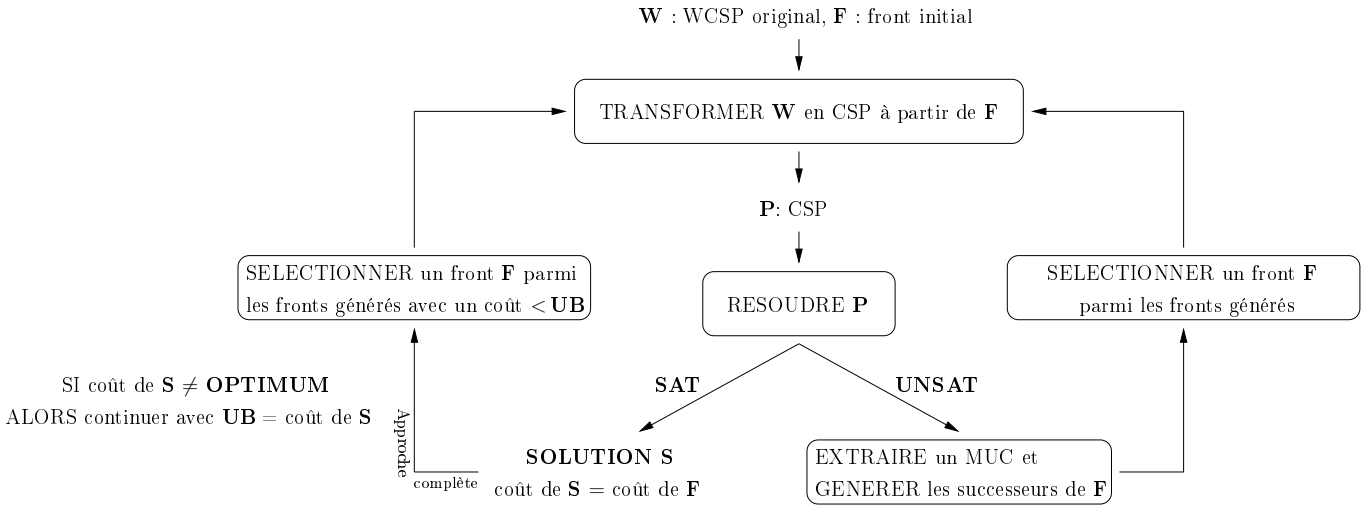


FIG. 4.3 – Principe général de résolution.

Le réseau de contraintes P ainsi construit est alors résolu par un solveur de contraintes qui résout un CN donné en paramètre et retourne soit une solution, soit $\perp$ si le CN est insatisfaisable.

Si P est insatisfaisable (UNSAT sur la figure 4.3), l'algorithme extrait un MUC à partir de P qui est alors exploité soit dans une approche gloutonne (incomplète), soit dans une approche complète. Dans la version gloutonne, certaines contraintes du MUC vont être successivement relâchées jusqu'à ce que la satisfaisabilité du MUC soit restaurée. Plus précisément, un sous-ensemble des strates de certaines contraintes présentes dans le MUC vont basculer du statut "interdit" au statut "autorisé". Lorsque ces contraintes sont relâchées suffisamment pour rendre satisfaisable le MUC, le processus général recommence jusqu'à ce que la satisfaisabilité globale de P soit atteinte et qu'une solution soit retournée. Dans les versions complètes, une fois le MUC identifié, on rassemble tous les successeurs directs du front courant en relâchant au moins une des contraintes, dans le but d'essayer de casser le MUC. Les fronts sont extraits les uns après les autres, transformés en CN, et résolus successivement jusqu'à ce que un front représentant un réseau de contraintes satisfaisable soit trouvé.

Si P est satisfaisable (SAT sur la figure 4.3), alors une solution est retournée et son coût dans le WCSP d'origine correspond au coût du front courant. Dans la version gloutonne, le processus s'arrête. Dans la première approche complète que nous proposons, le coût du front courant correspond au coût d'une solution optimale pour le WCSP W. Dans la deuxième approche, le coût du front courant correspond à une borne supérieure (UB) du coût d'une solution optimale pour le WCSP W et on continue de chercher parmi tous les fronts générés un front de coût minimal strictement inférieur à UB et correspondant à un CN satisfaisable.

Il est intéressant de noter que le principe décrit ci-dessus peut être généralisé pour prendre en compte tout type de contraintes : les contraintes dures du cadre CSP, les contraintes "violables" du cadre Weighted Max-CSP et les contraintes souples du cadre WCSP. En effet, il suffit simplement pour chaque type de contraintes de spécifier le processus de transformation à suivre. En d'autres termes, notre approche permet de traiter facilement et naturellement ces différents cadres. Toutefois, nous n'aborderons pas ces différentes variantes dans le cadre de ce manuscrit.

4.5 Algorithmes annexes

Nous présentons dans cette section les bases de nos algorithmes principaux, à savoir l'algorithme toCN() qui permet de transformer un réseau souple WCN en un réseau dur CN et la fonction extractMUC() qui consiste à extraire un MUC à partir d'un CN insatisfaisable.

4.5.1 Transformation d'un réseau souple (WCN) en un réseau classique (CN)

Les fronts sont exploités par l'algorithme toCN qui construit un CN à partir d'un WCN et les strates sélectionnées. Deux versions de l'algorithme toCN sont considérées : toCN$_=$ et toCN$_\leq$, chacune prenant en entrée un WCN W et un front f. La première version, notée toCN$_=(W, f)$, construit un CN en sélectionnant comme tuples autorisés pour une contrainte souple w uniquement les tuples appartenant à la strate $f[w]$; cette strate est dite autorisée. La seconde version, notée toCN$_\leq(W, f)$, construit un CN en sélectionnant comme tuples autorisés pour une contrainte w les tuples des strates d'indice inférieur ou égal à $f[w]$ (donc de coût inférieur ou égal) ; ces strates sont dites autorisées. Autrement dit, pour toute contrainte dure c construite à partir d'une contrainte souple w de W, les tuples autorisés dans c sont ceux ayant un coût égal (ou inférieur ou égal) au coût de la strate $f[w]$.

L'algorithme toCN$_=$ (algorithme 29) transforme donc un WCN en CN à partir d'un front donné, en autorisant uniquement pour chaque contrainte les tuples appartenant à la strate sélectionnée dans le front. Pour chaque contrainte souple w_S du WCN initial W, l'algorithme commence par vérifier si le coût de la strate sélectionnée pour cette contrainte dans le front f, à savoir $w.\texttt{layers}[f[w]].\texttt{cost}$, correspond au coût par défaut de cette contrainte (ligne 4). Si c'est le cas, cela veut dire que seuls les tuples appartenant à la strate du coût par défaut sont autorisés. Pour rappel, il s'agit des tuples implicites et ils ne doivent pas (par définition) être listés. En pratique, deux représentations d'une contrainte dure peuvent alors être envisagées. Lorsque la strate correspondant au coût par défaut est autorisée, toCN crée une contrainte dure négative qui liste les tuples des strates non autorisées comme étant des tuples interdits, et ceci pour éviter d'énumérer tous les tuples implicites ayant un coût égal au coût par défaut (car il y a un risque d'explosion combinatoire spatiale). Les tuples contenus dans toutes les autres strates sont alors regroupés dans T (ligne 5 à 7) qui représente, pour la contrainte table dure c_S créée, la liste des tuples dans $c_S.\texttt{table}$ qui sont alors interdits ($c_S.\texttt{semantics} = \texttt{CONFLICTS}$). Lorsque la strate correspondant au coût par défaut n'est pas autorisée, toCN crée une contrainte dure positive qui liste les tuples de la strate autorisée comme étant des tuples autorisés. Seuls les tuples de la strate sélectionnée dans f sont autorisés (ligne 10) et représentent pour la contrainte table dure c_S créée (ligne 12) la liste des tuples

dans c_S.`table` qui sont alors autorisés (c_S.`semantics` = `SUPPORTS`). L'ensemble des contraintes dures créées (ainsi que l'ensemble des variables du WCN d'origine) constituent le CN à retourner.

Algorithm 29: toCN$_=$(W : WCN, f : front) : CN

```
1  cons ← ∅ ;
2  foreach w ∈ cons(W) do
3      T ← ∅ ;
4      if w.layers[f[w]].cost = w.defaultCost then
5          foreach layer ∈ w.layers do
6              if layer.cost ≠ w.defaultCost then
7                  T ← T ∪ layer.tuples ;
8          semantics ← CONFLICTS ;
9      else
10         T ← w.layers[f[w]].tuples ;
11         semantics ← SUPPORTS ;
12     c ← new HardTableConstraint(w.S, T, semantics) ;
13     cons ← cons ∪ {c} ;
14 return (vars(W),cons) ;
```

La figure 4.4 illustre un exemple de fonctionnement de l'algorithme toCN$_=$ dans lequel nous considérons un WCN W composé des contraintes w_{xy}, w_{yz}, w_x, et le front $< 1, 0, 1 >$ pour ce WCN.

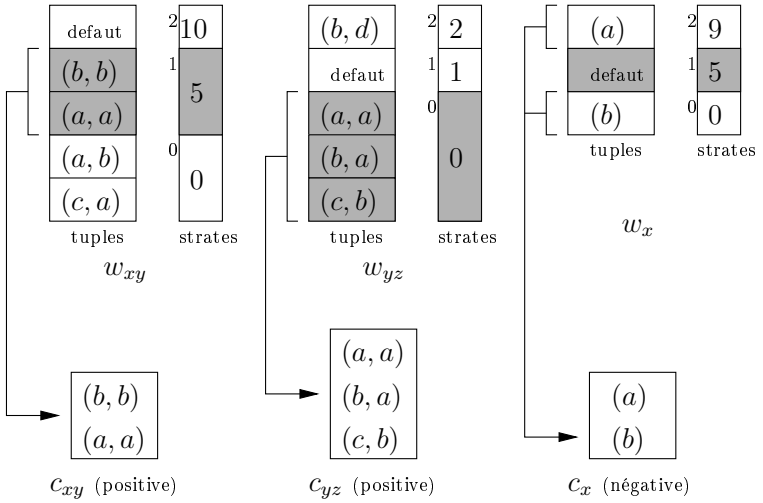


FIG. 4.4 – Appel à l'algorithme toCN$_=(W, f)$ avec W composé des contraintes w_{xy}, w_{yz}, w_x et f le front $< 1, 0, 1 >$.

Pour chacune des contraintes, les tuples (tableau de gauche sur la figure 4.4) sont regroupés par coût dans des strates (tableau de droite sur la figure 4.4, avec chaque case contenant le coût de la strate pour laquelle l'indice est précisé juste à côté de cette case, en haut à gauche) qui sont triées par coût strictement croissant. Par exemple, au sein de la contrainte w_{xy}, les tuples (a, b) et (c, a) ont un coût de 0 et sont donc regroupés dans la strate de coût 0, les tuples (b, b) et (a, a) ont un coût de 5 et sont regroupés dans la strate 1 de coût 5, puis la dernière strate de coût 10 correspond au coût par défaut et ne contient aucun

tuple explicite (on ne liste pas les tuples implicites). Les strates sont triées selon leur coût : la première strate (indice 0) correspond à celle de coût 0, la deuxième strate (indice 1) correspond à celle de coût 5, la troisième strate (indice 2) correspond à celle de coût 10. Lors de l'appel à toCN$_=(W, f)$, chacune de ces trois contraintes souples est transformée en contrainte table dure dont la sémantique dépend du fait que la strate correspondant au coût par défaut soit autorisée ou non. Les strates autorisées par le front apparaissent en grisé sur la figure. Pour la contrainte w_{xy}, c'est la strate d'indice 1 qui est autorisée : il ne s'agit pas de la strate du coût par défaut, la contrainte table dure c_{xy} obtenue est donc positive et contient comme tuples autorisés les tuples (b, b) et (a, a). Même constat pour la contrainte w_{yz} : ce n'est pas la strate du coût par défaut qui est autorisée et donc la contrainte table dure c_{yz} obtenue est positive et contient les tuples (a, a), (b, a) et (c, b). C'est différent pour la contrainte w_x : cette fois-ci, c'est la strate du coût par défaut qui est autorisée : ce sont donc les tuples de toutes les autres strates qui sont récupérées et stockés dans la contrainte table dure c_x obtenue et cette table est négative car les tuples stockés ne sont pas autorisés.

L'algorithme toCN$_\leq$ (algorithme 30) transforme un WCN en CN à partir d'un front donné, en autorisant uniquement pour chaque contrainte les tuples appartenant aux strates de coût inférieur ou égal au coût de la strate sélectionnée dans le front. L'algorithme commence par vérifier pour chaque contrainte souple w_S du WCN initial W si le coût de la strate sélectionnée pour cette contrainte dans le front f est strictement inférieur au coût par défaut de cette contrainte (ligne 4). Si les tuples implicites ne sont pas autorisés, c'est à dire que le coût de la strate sélectionnée pour w_S est strictement inférieur au coût par défaut, les tuples appartenant aux strates de coût inférieur ou égal au coût de cette strate sont alors regroupés dans T (ligne 5 à 7) qui représente, pour la contrainte table dure c_S créée (ligne 14), la liste des tuples dans c_S.`table` qui sont alors autorisés (c_S.`semantics` $=$ `SUPPORTS`). Si par contre les tuples implicites sont autorisés, les tuples appartenant aux strates de coût strictement supérieur au coût de la strate sélectionnée pour w_S dans f sont alors regroupés dans T (ligne 10 à 12) qui représente, pour la contrainte table dure c_S créée, la liste des tuples dans c_S.`table` qui sont alors interdits (c_S.`semantics` $=$ `SUPPORTS`). L'ensemble des contraintes dures créées (ainsi que l'ensemble des variables du WCN d'origine) constituent le CN à retourner.

Algorithm 30: toCN$_\leq$(W : WCN, f : front) : CN

```
1  cons ← ∅ ;
2  foreach w ∈ cons(W) do
3      T ← ∅ ;
4      if w.layers[f[w]].cost < w.defaultCost then
5          foreach layer ∈ w.layers do
6              if layer.cost ≤ w.layers[f[w]].cost then
7                  T ← T ∪ layer.tuples ;
8          semantics ← SUPPORTS ;
9      else
10         foreach layer ∈ w.layers do
11             if layer.cost > w.layers[f[w]].cost then
12                 T ← T ∪ layer.tuples ;
13         semantics ← CONFLICTS ;
14     c ← new HardTableConstraint(w.S, T, semantics) ;
15     cons ← cons ∪ {c} ;
16 return (vars(W),cons) ;
```

La figure 4.5 illustre un exemple de fonctionnement de l'algorithme toCN$_{\leq}$. Il s'agit du WCN introduit dans la figure 4.4, cependant la transformation est quelque peu différente. Cette fois-ci, un appel à l'algorithme toCN$_{\leq}(W, f)$ va autoriser pour chaque contrainte souple les tuples appartenant à une strate d'indice inférieur ou égal à l'indice de la strate sélectionnée dans le front pour la contrainte. Par exemple, pour la contrainte w_{xy}, l'indice de la strate autorisée dans le front est égale à 1, cela veut dire que les strates d'indice 0 et 1 sont autorisées. Comme aucune de ces strates ne correspond à la strate correspondant au coût par défaut, la contrainte table dure c_{xy} créée est positive et contient les tuples (b, b) et (a, a) (strate d'indice 0), plus les tuples (a, b) et (c, a) (strate d'indice 1). Pour la contrainte w_{yz}, la strate d'indice 0 sélectionnée, qui n'est pas la strate avec le coût par défaut, donne une contrainte table dure c_{yz} positive et contenant les tuples (a, a), (b, a) et (c, b). Par contre, pour la contrainte w_x, les strates autorisées par le front sont les strates d'indice 0 et 1 et cette fois-ci la strate pour le coût par défaut (indice 1) est incluse et donc autorisée. La contrainte table dure c_x créée est alors négative et contient tous les tuples appartenant à une strate d'indice strictement supérieur à l'indice de la strate sélectionnée dans le front. Ces tuples sont donc ceux appartenant à la strate d'indice 2, c'est à dire le tuple (a).

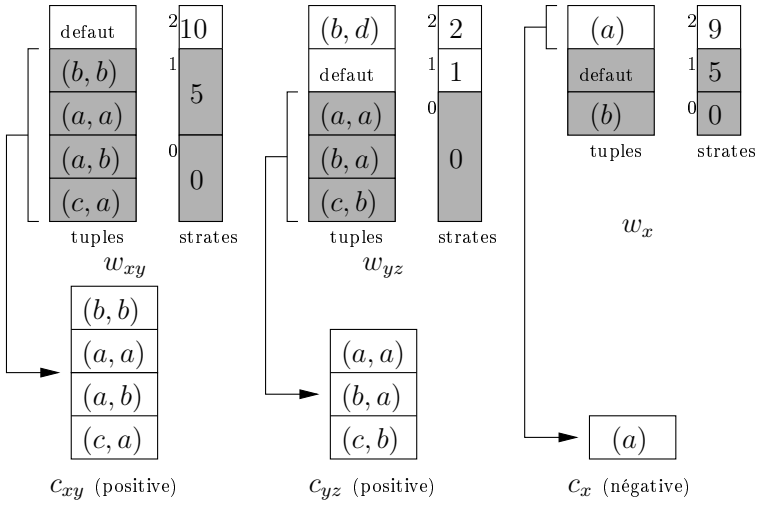


FIG. 4.5 – Appel à l'algorithme toCN$_{\leq}(W, f)$ avec W composé des contraintes w_{xy}, w_{yz}, w_x et f le front $< 1, 0, 1 >$.

4.5.2 Extraction de noyaux insatisfaisables minimaux

L'algorithme extractMUC(P) retourne un MUC à partir d'un CN P (insatisfaisable) passé en paramètre. Dans le cadre de notre approche, afin d'extraire les noyaux insatisfaisables depuis un réseau, nous avons choisi d'utiliser la méthode assez efficace proposée par [Hemery *et al.*, 2006] déjà implémentée dans le solveur $AbsCon$ et sujette à de récents travaux [Gregoire *et al.*, 2013]. N'oublions pas cependant que d'autres méthodes, comme [de Siqueira et Puget, 1988], [Junker, 2004] et [Wieringa, 2012], ont également été proposées pour extraire ce type de noyaux. Basée sur la notion de preuve d'insatisfaisabilité, l'approche présentée se compose de deux étapes : extraire un noyau de contraintes insatisfaisable représentant une preuve d'insatisfaisabilité du réseau, puis affiner ce noyau en réduisant le nombre des contraintes au minimum tout en conservant sa propriété d'insatisfaisabilité et obtenir ainsi un noyau insatisfaisable minimal.

Pour extraire un noyau de contraintes insatisfaisable depuis un réseau de contraintes, on effectue lors d'une première étape une succession de résolutions à partir de ce réseau. Il s'agit de résolutions complètes avec recherche en profondeur d'abord et retours en arrière, de type MAC (voir section 1.3.5), guidées

par l'heuristique de choix de variables *dom/wdeg* dirigée par les conflits (voir section 1.3.3). D'une résolution à l'autre, seules les contraintes actives lors de la dernière résolution (dont la propagation a entrainé la suppression d'au moins une valeur d'un des domaines des variables) sont conservées, ce qui amène alors une réduction progressive du réseau. Chaque sous réseau correspond alors à un noyau de contraintes insatisfaisable, et donc à une nouvelle preuve d'insatisfaisabilité. De plus, lors de chaque résolution, le poids de chaque contrainte est conservé pour les résolutions suivantes afin d'être exploitées par l'heuristique de choix de variables *dom/wdeg*. En effet, lors des résolutions suivantes, les variables dont le degré pondéré (somme des poids impliquant la variable) est le plus élevé vont être choisies prioritairement pour orienter la recherche vers la partie la plus difficile du réseau. Ainsi, des noyaux insatisfaisables pourront être décelés plus rapidement : ils seront potentiellement petits et concentrés sur les contraintes appartenant à la partie la plus difficile du réseau. Lorsqu'une résolution ne permet pas de réduire le nombre de contraintes du noyau obtenu lors de la résolution précédente (et n'améliore donc pas la preuve d'insatisfaisabilité courante), le processus s'arrête et le noyau courant constitue le noyau insatisfaisable cherché (lors de la première étape). Afin d'illustrer ce processus d'extraction, considérons par exemple le réseau binaire insatisfaisable représenté par son graphe de contraintes dans la figure 4.6.

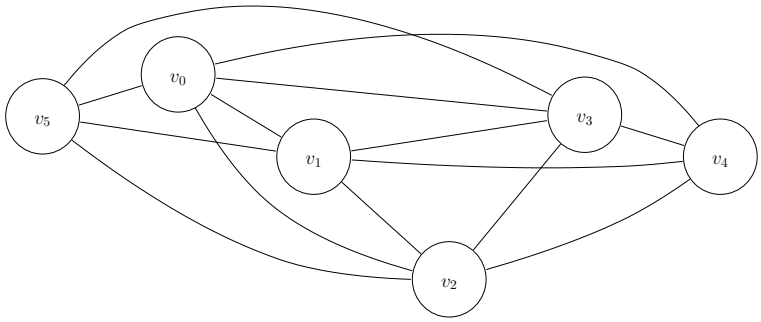


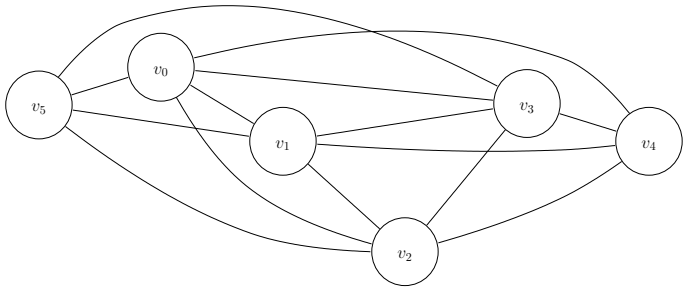
FIG. 4.6 – Réseau original insatisfaisable.

Considérons qu'une première résolution de ce réseau soit effectuée avec *MAC+dom/wdeg* à la figure 4.7. La figure 4.7(a) illustre les contraintes du réseau qui ont été actives lors de cette résolution, ainsi que leur poids. Par exemple, la contrainte $C_{v_1v_3}$ a été active car elle a amené la suppression d'au moins une valeur et son poids a été incrémenté 3 fois, c'est à dire qu'un conflit a été détecté 3 fois pendant la recherche alors que la contrainte était sollicitée pour le filtrage. Après une première résolution, on fait l'hypothèse que toutes les contraintes ont été actives : elles sont donc toutes conservées pour la résolution suivante. La première preuve d'insatisfaisabilité (noyau insatisfaisable) trouvée correspond alors au réseau original rappelé à la figure 4.7(b). Cependant, cette première résolution a permis de mettre à jour les poids des contraintes et ces données vont pouvoir être exploitées par l'heuristique de choix de variable *dom/wdeg* durant la deuxième résolution présentée à la figure 4.8.

La figure 4.8(a) illustre les contraintes actives et leurs poids respectifs après la seconde résolution et on constate que seules 6 contraintes ont été actives : ceci est dû à l'heuristique *dom/wdeg* qui a permis de se concentrer prioritairement sur les variables impliquées dans les contraintes les plus difficiles. Une nouvelle preuve d'insatisfaisabilité est donc obtenue à la figure 4.8(b), composée des variables v_0, v_1, v_2, v_3 et des contraintes $C_{v_0v_1}$, $C_{v_0v_1}$, $C_{v_0v_1}$, $C_{v_0v_1}$, $C_{v_0v_1}$ et $C_{v_0v_1}$, en supprimant les contraintes qui n'ont pas été actives. De plus, cette nouvelle preuve d'insatisfaisabilité est exploitée car le nombre de contraintes de ce nouveau noyau, à savoir 6, est strictement inférieur au nombre de contraintes du noyau précédemment trouvé, à savoir 14. Dans la figure 4.9, on effectue alors une troisième et quatrième résolutions à partir du réseau illustré à la figure 4.8(b). La troisième résolution décèle les contraintes actives illustrées dans la figure 4.9(a) et une nouvelle preuve d'insatisfaisabilité illustrée à la figure 4.9(b), meilleure que la

actives	poids
$c_{v_0v_1}$	2
$c_{v_0v_2}$	2
$c_{v_0v_3}$	2
$c_{v_1v_2}$	5
$c_{v_1v_3}$	4
$c_{v_2v_3}$	3
⋮	⋮

(a) Contraintes actives et poids.

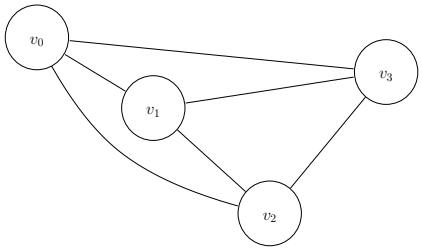


(b) Preuve d'insatisfaisabilité.

FIG. 4.7 – Une première résolution avec *MAC+dom/wdeg* sur le réseau original.

actives	poids
$c_{v_0v_1}$	1
$c_{v_0v_2}$	1
$c_{v_0v_3}$	1
$c_{v_1v_2}$	4
$c_{v_1v_3}$	3
$c_{v_2v_3}$	2

(a) Contraintes actives et poids.



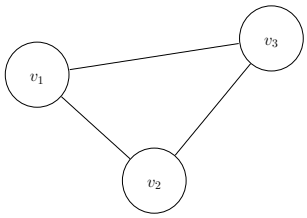
(b) Preuve d'insatisfaisabilité.

FIG. 4.8 – Une deuxième résolution avec *MAC+dom/wdeg* sur le dernier réseau obtenu.

précédente car elle possède 3 contraintes au lieu de 6, a été trouvée. Finalement, la quatrième résolution ne permet pas de diminuer le nombre de contraintes de cette dernière preuve de satisfaisabilité trouvée (également 3 contraintes actives présentées dans la figure 4.9(c)) et donc le processus s'arrête avec le noyau composé des variables v_0, v_1, v_2 et des contraintes $C_{v_1v_2}$, $C_{v_1v_3}$ et $C_{v_2v_3}$.

actives	poids
$c_{v_1v_2}$	4
$c_{v_1v_3}$	3
$c_{v_2v_3}$	2

(a) Contraintes actives et poids.



(b) Preuve d'insatisfaisabilité.

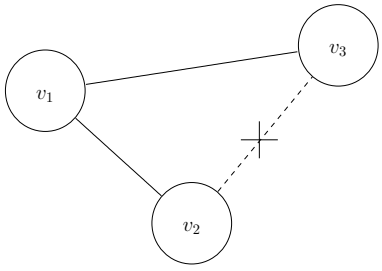
actives	poids
$c_{v_1v_2}$	3
$c_{v_1v_3}$	2
$c_{v_2v_3}$	1

(c) Contraintes actives et poids.

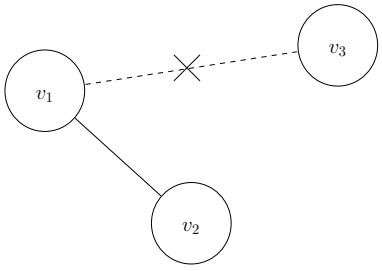
FIG. 4.9 – Troisième et quatrième résolutions avec *MAC+dom/wdeg*.

Bien sûr, le noyau insatisfaisable obtenu à l'issue de cette première étape (figure 4.9(b)) n'est pas forcément un noyau minimal. Une deuxième étape permet alors de calculer un noyau minimal à partir du noyau trouvé lors de la première étape. Parmi les approches proposées par [Hemery *et al.*, 2006], nous avons privilégié celle basée sur une recherche dichotomique, suggérée (et montrée) comme la plus efficace. Comme son nom l'indique, cette méthode (récursive) consiste à séparer les contraintes du noyau (ordonnées par poids décroissant) en deux ensembles de taille égale et de conserver à chaque étape l'ensemble de contraintes qui ne correspond pas à un réseau satisfaisable. Le processus se poursuit jusqu'à

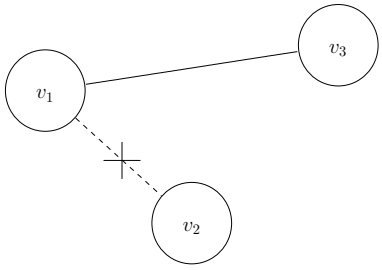
ce qu'une *contrainte de transition* soit isolée : il s'agit d'une contrainte dont la suppression rend satisfaisable l'ensemble qui était insatisfaisable au préalable. Lorsqu'une contrainte de transition a été trouvée, on la place dans le noyau qui est en train d'être construit. Le noyau insatisfaisable minimal obtenu correspondra à l'ensemble des contraintes de transition trouvées durant le processus. Dans l'exemple illustré à la figure 4.10, on tente de minimiser le noyau obtenu à la figure 4.9(b) lors de la première étape. On tente de trouver une première contrainte de transition : on supprime la contrainte $C_{v_2v_3}$ et le réseau obtenu à la figure 4.10(a) est toujours insatisfaisable. On supprime alors la contrainte $C_{v_1v_3}$ et cette fois-ci le réseau obtenu à la figure 4.10(b) est satisfaisable : $C_{v_1v_3}$ correspond donc à une première contrainte de transition. Pour la prochaine itération, nous considérons donc le réseau constitué de la première contrainte de transition trouvée, à savoir $C_{v_1v_3}$, et on conserve uniquement les contraintes du réseau original qui sont supérieures (selon l'ordre fixé sur le poids) à cette contrainte de transition (on a ainsi la garantie de conserver l'insatisfaisabilité), en l'occurrence ici la contrainte $C_{v_1v_2}$ (elle a un poids de 3 alors que $C_{v_1v_3}$ a un poids de 2, les contraintes étant triées par ordre de coût décroissant). On sépare à nouveau en deux ensembles les deux contraintes restantes, on traite l'ensemble composé uniquement de la contrainte $C_{v_1v_2}$, et on s'aperçoit dans la figure 4.10(c) qu'en supprimant cette contrainte $C_{v_1v_2}$, le réseau constitué des contraintes $C_{v_1v_2}$ et $C_{v_1v_3}$ qui était insatisfaisable devient satisfaisable : la contrainte $C_{v_1v_2}$ correspond donc également à une contrainte de transition. Le processus est terminé, le noyau insatisfaisable minimal illustré à la figure 4.10(d) est donc composé des deux contraintes de transition trouvées durant le processus, à savoir les contraintes $C_{v_1v_2}$ et $C_{v_1v_3}$.



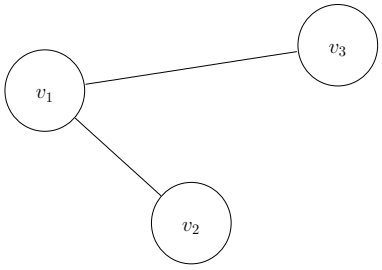
(a) Noyau insatisfaisable.



(b) Noyau satisfaisable, $C_{v_1v_3}$ est une contrainte de transition.



(c) Noyau satisfaisable, $C_{v_1v_2}$ est une contrainte de transition.



(d) Noyau insatisfaisable minimal.

FIG. 4.10 – Extraction d'un noyau insatisfaisable minimal (approche dichotomique).

À noter que nous avons mis en place dans le cadre de cette approche une des perspectives proposées dans [Hemery *et al.*, 2006], à savoir que nous effectuons une extraction sur la base des variables en amont de l'extraction présentée ici, basée sur les contraintes. Le principe dichotomique est le même sauf qu'il est basé sur un noyau de variables réduit jusqu'à un noyau minimal constitué de *variables de transition* (et des contraintes associées à ces variables). De ce fait, avant de commencer l'extraction d'un noyau insatisfaisable de contraintes, le nombre de contraintes est préalablement réduit. Comme nous le verrons

par la suite, nous avons testé également d'extraire un noyau insatisfaisable minimal uniquement sur la base des variables, ce qui nous a permis d'améliorer certains de nos résultats. Pour conclure, notons que dans un souci de gain d'efficacité, nous avons choisi de suivre une des améliorations possibles proposées là encore dans les perspectives de [Hemery *et al.*, 2006]. Cela concerne les sous réseaux intermédiaires : contrairement à l'approche implémentée dans [Hemery *et al.*, 2006], nous avons choisi de ne pas sauver ni charger chaque sous réseau intermédiaire en XML. Pour cela, nous avons basé notre implémentation sur le principe d'activer et de désactiver les variables et les contraintes via des booléens dans le réseau original pour simuler les différents sous-réseaux.

Nous terminons cette section par une brève description de fonctions annexes également utilisées par nos algorithmes. La fonction cons(W, M) retourne l'ensemble des contraintes souples du WCN W qui correspondent à celles du MUC M (par construction une contrainte dure est associée à une contrainte souple). La fonction restrict(W, M) retourne un WCN contenant uniquement les contraintes du WCN W qui correspondent à des contraintes présentes dans le MUC M.

4.6 Première approche

4.6.1 Algorithme complet préliminaire

Afin de faciliter la compréhension de notre approche, nous présentons un premier algorithme complet qui n'exploite pas les MUC et qui n'est donc pas très efficace. L'algorithme completeSearchNoMUC(), Algorithme 31, commence avec un premier front qui ne retient que la strate 0 de chaque contrainte (lignes 1 et 2).

Algorithm 31: completeSearchNoMUC(W : WCN) : sol

```
1  foreach w ∈ cons(W) do
2      f[w] ← 0;
3  Q ← {f} ;
4  while Q ≠ ∅ do
5      f ← pick and delete smallest cost front in Q ;
6      P ← toCN_=(W, f) ;
7      sol ← solveCN(P) ;
8      if sol ≠ ⊥ then
9          return sol
10     else
11         foreach w ∈ cons(W) do
12             if f[w] < w.layers.length − 1 then
13                 f' ← f ;
14                 f'[w] ← f'[w] + 1 ;
15                 if cost(f') ≠ k then
16                     Q ← Q ∪ f' ;
17 return ⊥
```

Ensuite, tant qu'il reste un front à considérer, on extrait d'une queue de priorité le front de coût minimal. Le coût d'un front peut être facilement calculé en sommant les coûts associés aux strates sélectionnées pour chaque contrainte souple. Ensuite, nous pouvons construire avec l'algorithme 29 le CN

correspondant au front traité. Le CN obtenu est résolu via un appel solveCN. Si une solution est trouvée, elle est optimale, et l'algorithme s'arrête. En effet, l'algorithme énumère les fronts par ordre de coût croissant (voir plus tard la proposition 5) et garantit que tous les fronts précédents étaient insatisfaisables. Si le CN testé s'avère insatisfaisable, nous énumérons les successeurs directs du front courant (ligne 11 à 16) et on les insère dans la queue de priorité. Bien évidemment, il faut vérifier pour chaque contrainte que l'on ne dépasse pas la strate maximale. Par ailleurs, les fronts ayant un coût égal à k doivent être ignorés.

La queue de priorité Q utilisée dans cet algorithme est une variante des queues de priorité habituelles. En effet, d'une part elle doit garantir que les fronts sont extraits par ordre croissant de coût, d'autre part elle doit garantir qu'un front donné n'est pas inséré deux fois dans la queue. Par exemple, si l'on considère un réseau de deux contraintes possédant chacune au moins 2 strates, le premier front sera le front $< 0, 0 >$. Lors de la première itération, nous insérerons dans la queue les voisins $< 1, 0 >$ et $< 0, 1 >$. Lorsque ces fronts sont à leur tour extraits de la queue, ils généreront tous deux le même voisin $< 1, 1 >$ qui ne doit être inséré qu'une seule fois dans la queue pour éviter des calculs redondants. Cette queue particulière s'obtient en modifiant très légèrement une implémentation classique d'une queue de priorité par tas binomial.

On peut facilement vérifier que l'algorithme 31 énumère tous les fronts possibles quand solveCN ne trouve jamais de solution. En effet, en faisant abstraction des coûts, il s'agit d'une exploration en largeur d'abord d'un arbre énumérant les fronts. La proposition 5 garantit en plus que cet algorithme ne peut pas boucler indéfiniment et que les fronts sont énumérés par ordre de coût croissant.

Proposition 5 [19] *Les propriétés suivantes sont garanties par l'algorithme 31 :*

A) les fronts sont énumérés par ordre croissant de coût ;

B) une fois qu'un front f est extrait de la queue de priorité, il ne peut plus jamais y être inséré.

Preuve Soit f_1 un premier front extrait de la queue et f_2 un front extrait de la queue après l'extraction de f_1.

A) Deux cas sont possibles. Soit (A.1) f_2 se trouvait déjà dans la queue quand f_1 en a été extrait, soit (A.2) f_2 a été placé dans la queue après l'extraction de f_1. Dans le cas (A.1), la queue de priorité garantit que $cost(f_1) \leq cost(f_2)$. Dans le cas (A.2), f_2 est issu d'un front f tel que $f \rightarrow_* f_2$ et tel que soit $f = f_1$, soit f était présent dans la queue quand f_1 a été extrait. Dans les deux cas, $cost(f_1) \leq cost(f)$. Comme $f \rightarrow_* f_2 \Rightarrow cost(f) < cost(f_2)$, on en conclut que $cost(f_1) < cost(f_2)$. Par conséquent, on obtient dans tous les cas que $cost(f_1) \leq cost(f_2)$.

B) Supposons que $f_1 = f_2$. Comme la queue garantit qu'elle ne contient jamais deux fronts identiques à un instant donné, on en conclut que l'on se situe nécessairement dans le cas (A.2) précédent. Or, nous avons prouvé que dans ce cas, $cost(f_1) < cost(f_2)$, ce qui contredit $f_1 = f_2$. □

Il est à noter que dans notre approche, les contraintes unaires sont considérées comme des contraintes tout à fait ordinaires. Par ailleurs, dans la mesure où une seule strate d'une contrainte est retenue à un instant donné, les CN générés sont de taille beaucoup plus réduite que le WCN de départ. On peut espérer que la résolution soit assez simple (donc rapide) à réaliser dans la plupart des cas. Une dernière observation est que cet algorithme exploite une forme d'abstraction en considérant que, d'un point de vue global, il n'y a pas lieu de distinguer entre eux les tuples de même coût d'une même contrainte.

Pour finir, l'inconvénient de cette première version est qu'elle énumère toutes les combinaisons possibles de strates. Dans le pire des cas, elle énumère un nombre de fronts égal au produit du nombre de

[19] Cette proposition est similaire au lemme 4 proposé dans la section 3.8.1, mais sous une autre forme. En effet, ici il s'agit de fronts et non pas de tuples

strates de chaque contrainte, qui correspond à $\prod_{w \in cons(W)} w.\texttt{layers}.length$. Selon le nombre de variables et de contraintes, cette complexité peut s'avérer nettement plus grande que dans une approche par séparation et évaluation classique (i.e. quand $\prod_{w \in cons(W)} w.\texttt{layers}.length \ggg \prod_i |dom(x_i)|$). Cela s'explique en particulier par le fait que, dans notre approche, une même instanciation peut être explorée plusieurs fois dans les différentes résolutions. Néanmoins, même dans ce cas, cette approche peut se révéler payante si le coût de la solution optimale est faible.

4.6.2 Exploitation des noyaux insatisfaisables minimaux

La complexité de l'algorithme précédent peut être fortement réduite en pratique en remarquant qu'il est vain de passer à la strate supérieure pour une contrainte qui ne participe pas à l'insatisfaisabilité du CN. L'idée est donc d'identifier un MUC et de ne passer à la strate supérieure uniquement pour les contraintes appartenant à ce MUC. Dans le pire des cas, le MUC retourné contient toutes les contraintes du problème et donc la complexité dans le pire des cas reste inchangée. Cependant, en pratique, sur des instances concrètes, les MUC sont souvent de petite taille. De ce fait, le voisinage généré est bien plus petit et la complexité pratique est fortement réduite.

Approche complète

L'algorithme completeSearch1(), Algorithme 32, inspiré de [Fu et Malik, 2006, Ansótegui *et al.*, 2010, Marques-Sila et Planes, 2011], présente l'algorithme modifié pour prendre en compte les MUC dans une version complète.

Algorithm 32: completeSearch1(W : WCN) : sol

```
1  foreach w ∈ cons(W) do
2      f[w] ← 0;
3  Q ← {f} ;
4  while Q ≠ ∅ do
5      f ← pick and delete smallest cost front in Q ;
6      P ← toCN_=(W, f) ;
7      sol ← solveCN(P) ;
8      if sol ≠ ⊥ then
9          return sol
10     else
11         M ← extractMUC(P) ;
12         foreach w ∈ cons(W, M) do
13             if f[w] < w.layers.length − 1 then
14                 f' ← f ;
15                 f'[w] ← f'[w] + 1 ;
16                 if cost(f') ≠ k then
17                     Q ← Q ∪ {f'} ;
18 return ⊥
```

Les premières étapes de l'algorithme sont les mêmes que pour l'algorithme completeSearchNoMUC(), Algorithme 31. Quand le CN courant est insatisfaisable, un MUC est identifié, et on va progressivement autoriser des strates plus élevées dans les contraintes du MUC. Plus précisément, l'algorithme énumère

tous les successeurs directs du front courant afin de relâcher ce MUC. Pour chaque contrainte du MUC (obtenue par l'appel cons(W, M)), l'algorithme génère un successeur f' du front courant qui ne diffère de ce dernier que par le relâchement de cette contrainte (incrémentation de la strate autorisée). Ce nouveau front est alors inséré dans la queue Q à une position dépendante de la valeur de $cost(f')$.

Proposition 6 *L'algorithme* completeSearch1()*, Algorithme 32, est complet.*

Preuve Par abus de langage, nous utiliserons le terme "front" pour désigner implicitement le CN associé à un front f par toCN$_=(W, f)$. Soit f_u le front qui est identifié comme insatisfaisable et M le MUC extrait de ce front. La seule différence entre les algorithmes completeSearch1() et completeSearchNoMUC(), c'est à dire les algorithmes 32 et 31, est que lorsqu'un MUC est identifié, on restreint la recherche en s'imposant de relâcher en priorité ce MUC.

Si le WCN n'admet pas de solution, restreindre ainsi la recherche ne fait pas perdre de solution. On peut donc se restreindre au cas où le WCN admet au moins une solution S. Soit f_S le front correspondant à une solution S. L'algorithme 31 garantit qu'il existe toujours un front f dans la queue de priorité tel que $f \rightarrow_* f_S$. C'est en particulier vrai pour le premier front $\perp$ placé dans la queue et lorsque cette propriété est vérifiée pour un front extrait de la queue, elle est vérifiée pour au moins l'un de ses successeurs directs (par définition de $\rightarrow_*$). Par conséquent, par récurrence, cette propriété est toujours assurée.

Si $f_u \not\rightarrow_* f_s$, la restriction proposée ne peut pas nous faire perdre la solution S. Par conséquent, nous pouvons donc faire l'hypothèse que $f_u \rightarrow_* f_s$. De ce fait, $\forall w \in cons(W), f_s[w] \geq f_u[w]$. Par ailleurs, il existe nécessairement une contrainte $w_i \in cons(W, M)$ telle que $f_s[w_i] > f_u[w_i]$, faute de quoi f_S serait toujours insatisfaisable. Soit f' le front défini par $f'[w_i] = f_u[w_i] + 1$ et $\forall w \in cons(W)$ tel que $w \neq w_i$, $f'[w] = f_u[w]$. f' est un front qui est généré par l'algorithme et par construction $f' \rightarrow_* f_s$. □

La figure 4.11 présente un exemple où l'on ne peut pas se contenter de relâcher un MUC d'une seule manière, fût-ce le relâchement local de coût optimal, sans perdre la solution optimale.

coût	tuple
100	{(x,c)}
10	{(x,b)}
0	{(x,a)}

(a) w_x

coût	tuple
100	default
5	{(x,c),(y,a)}
0	{(x,a),(y,b)}

(b) w_{xy}

coût	tuple
100	{(y,c)}
10	{(y,b)}
0	{(y,a)}

(c) w_y

FIG. 4.11 – De la nécessité d'énumérer tous les relâchements d'un MUC.

Dans cet exemple, un WCN est composé de trois contraintes w_x, w_{xy} et w_y. Le premier front sélectionne les strates de coût 0, ce qui ne conserve comme tuples autorisés que $\{(x, a)\}$, $\{(x, a), (y, b)\}$ et $\{(y, a)\}$. Bien évidemment, dans ce cas, les contraintes w_{xy} et w_y forment un MUC. Il y a deux manières de relâcher ce premier MUC : la première consiste à passer à la strate suivante de w_{xy} (c'est localement le meilleur choix puisqu'elle a un coût égal à 5), alors que la seconde passe à la strate suivante de w_y. Dans le premier cas, le nouveau front obtenu conserve comme tuples autorisés $\{(x, a)\}$, $\{(x, c), (y, a)\}$ et $\{(y, a)\}$. Cette fois, le MUC contient les contraintes w_x et w_{xy}. On peut soit relâcher w_{xy} pour aboutir à une solution ayant un coût de 100, soit relâcher w_x pour aboutir à un autre MUC que l'on peut relâcher de deux manières pour aboutir soit à une solution de coût 105, ou à une solution de coût 110. Si dans le premier MUC on avait relâché w_y, nous aurions directement obtenu une solution de coût 10 qui est l'optimum. Cela montre clairement que tous les relâchements d'un MUC sont requis.

Approche gloutonne

Nous présentons maintenant une version incomplète de notre approche utilisant l'extraction et le relâchement de MUC pour la résolution de WCN. Cette approche est incomplète dans le sens où le processus s'arrête dès qu'une solution est trouvée, celle-ci n'étant pas prouvée optimale. L'objectif ici est de satisfaire rapidement les MUC en relâchant de manière gloutonne (plusieurs strates à la fois) les contraintes présentes dans ces MUC, tout en veillant à assurer un relâchement optimal uniquement en local (c'est à dire au niveau du MUC). De ce fait, elle est gloutonne car il est possible que beaucoup de contraintes aient été plus ou moins relâchées pour satisfaire chacun des MUC rencontrés alors que pour certaines de ces contraintes, un tel relâchement n'était pas indispensable dans le relâchement global nécessaire pour la satisfaisabilité du WCN.

À partir d'un WCN W, l'algorithme incompleteSearch(), Algorithme 33, retourne une solution d'un CN dérivé de W. La structure $front$ est tout d'abord initialisée à 0, ce qui correspond à la première strate de chaque contrainte du WCN.

À partir d'un WCN W et d'un front f, on extrait un CN et ce réseau de contraintes est alors résolu. Si une solution est trouvée, alors elle est renvoyée par l'algorithme 33 (ligne 7). Si le réseau est prouvé insatisfaisable, il est alors nécessaire de relâcher des contraintes du WCN. Pour cela, l'algorithme extrait un WCN W' à partir d'un MUC calculé (lignes 9 et 10). Le front f est alors mis à jour par l'algorithme relax (ligne 11). Cette procédure relâchera une (ou plusieurs contraintes) de W' afin de casser le MUC associé à W'. Ce processus boucle jusqu'à ce qu'une solution soit trouvée pour le réseau de contraintes associé à W.

Algorithm 33: incompleteSearch(W : WCN) : sol

```
1  foreach w ∈ cons(W) do
2      f[w] ← 0 ;
3  repeat
4      P ← toCN≤(W, f) ;
5      sol ← solveCN(P) ;
6      if sol ≠ ⊥ then
7          return sol ;
8      else
9          M ← extractMUC(P) ;
10         W' ← restrict(W, M) ;
11         f ← relax(W', f) ;
12 until sol ≠ ⊥;
```

L'algorithme relax(W, f) met à jour le front f afin d'autoriser de nouvelles strates. Il faut noter que le corps de cet algorithme est similaire à celui de l'algorithme completeSearch1(), mais que le résultat est de nature différente et que l'algorithme toCN$_{\leq}$ est appelé à la place de toCN$_{=}$. Le CN obtenu est résolu avec solveCN.

Le principe général est d'incrémenter $f[w]$ pour au moins une contrainte w, de manière à rendre le MUC satisfaisable. Autrement dit, pour au moins l'une des contraintes, on s'autorise une augmentation du coût pour cette contrainte. Dans l'approche gloutonne adoptée ici, nous continuons jusqu'à ce que le MUC soit cassé.

L'algorithme 34 utilise une queue de priorité locale Q_{loc} qui est tout d'abord initialisée avec la structure f. Tant que Q_{loc} n'est pas vide (ligne 2), on extrait de la queue le front f ayant le plus petit coût $cost(f)$. L'algorithme toCN$_{\leq}$ construit un réseau de contraintes à partir du WCN et du front f précédem-

Algorithm 34: relax(W : WCN, f : front) : front

```
1  Q_loc ← {f} ;
2  while Q_loc ≠ ∅ do
3      f ← pick and delete smallest cost front in Q_loc ;
4      P ← toCN_≤(W, f) ;
5      if solveCN(P) ≠ ⊥ then
6          return f ;
7      else
8          M ← extractMUC(P) ;
9          foreach w ∈ cons(W, M) do
10             if f[w] < w.layers.length − 1 then
11                 f' ← f ;
12                 f'[w] ← f'[w] + 1 ;
13                 if cost(f') ≠ k then
14                     Q_loc ← Q_loc ∪ {f'} ;
```

ment sélectionné (ligne 4). Si ce réseau de contraintes est satisfaisable, f passé initialement en paramètre est mis à jour et retourné par l'algorithme 34. Au contraire, si le réseau de contraintes est insatisfaisable, cela veut dire que le relâchement n'est pas suffisant et l'algorithme met à jour alors sa queue de fronts en générant tous les successeurs directs de f. Pour chaque contrainte appartenant au WCN (obtenue à partir du MUC), nous générons un nouveau front, différent du front initial, en incrémentant la strate d'une seule des contraintes.

Notons que, dans les appels à l'algorithme relax(), Algorithme 34, on peut se restreindre à travailler seulement avec un sous-ensemble des contraintes. En effet les contraintes susceptibles d'être relâchées sont celles faisant partie du WCN W associé au MUC (restrict(W, M)). En pratique, nous travaillons simplement avec le sous-ensemble du front correspondant aux contraintes présentes dans le MUC, ce qui permet une économie d'espace.

Dans la version incomplète, nous transformons un WCN en CN en utilisant l'algorithme toCN$_\leq$ à la place de toCN$_=$ utilisée dans la version complète. En effet, dans la version complète, tous les relâchements possibles d'un MUC sont considérés et les fronts sont énumérés par ordre de coût croissant. De ce fait, lorsque l'on teste la satisfaisabilité d'un CN associé à un front f, nous savons que tous les CN associés à des fronts f' tel que $f' \rightarrow_* f$ ont déjà été prouvés insatisfaisables. Dans la version complète, l'algorithme toCN$_=$ extrait donc un CN composé des strates courantes d'un front. Dans la version incomplète, tous les relâchements possibles d'un MUC ne sont pas envisagés. Par conséquent, on ne peut donc pas garantir que lorsqu'on teste la satisfaisabilité d'un CN associé à un front f, tous les CN associés à des fronts f' tel que $f' \rightarrow_* f$ ont déjà été considérés et prouvés insatisfaisables. L'algorithme toCN$_\leq$ extrait donc un CN à partir des strates courantes et inférieures d'un front.

Cette approche ne garantit pas l'identification de la solution optimale. La raison est que tous les relâchements de MUC ne sont pas considérés contrairement à la version complète (voir exemple dans la figure 4.11). En effet, nous identifions seulement le premier relâchement permettant de restaurer la satisfaisabilité d'un MUC.

4.6.3 Résultats expérimentaux

Afin de montrer l'intérêt pratique de notre approche, nous avons mené des expérimentations sur un ordinateur équipé de processeurs Intel(R) Core(TM) i7-2820QM CPU 2.30GHz. Dans un premier temps, nous avons choisi de comparer notre approche gloutonne, notée GMR (GMR pour Greedy Muc Relaxation), à deux approches complètes avec algorithme de transferts de coûts maintenant la cohérence EDAC, et proposées par notre solveur $AbsCon$ et par le solveur $ToulBar2$ (version 0.9.4.0). Le temps limite (time-out) alloué pour résoudre chaque instance est de 600 secondes. Le temps CPU total pour résoudre chaque instance est donné ainsi que la borne trouvée (UB). Si l'exécution de l'algorithme n'est pas terminée avant le temps limite (CPU supérieur à 600 secondes), la borne affichée correspond à la borne trouvée au bout des 600 secondes. Le tableau 4.12(a) représente les résultats obtenus pour la série d'instances $spot5$ (exceptées les instances trop triviales) constituées de contraintes d'arité au maximum égale à 3. Le tableau 4.12(b) représente les résultats obtenus pour la série d'instances $celar$.

Pour les instances $spot5$, nous observons que notre approche permet d'obtenir de meilleures bornes que celles obtenues par les deux approches complètes, excepté pour les instances $spot5$-404, $spot5$-503 et $spot5$-1502. Au delà du fait que notre approche n'est pas complète, ceci s'explique par la facilité relative de ces trois instances (moins de $1,400$ contraintes) et donc les algorithmes à transferts de coûts parviennent à atteindre une meilleure borne dans le temps imparti. Cependant, il est intéressant de noter que la borne trouvée par notre approche pour ces trois instances n'est pas si éloignée que ça de celles obtenues par les approches complètes, et cela dans un temps très court. Concernant les autres instances, considérées comme plus difficiles (en particulier plus de $13,000$ contraintes pour les instances $spot5$-1403, $spot5$-1407 et $spot5$-1506), nous observons que notre approche gloutonne obtient de meilleures bornes que celles obtenues par les approches complètes et dans un temps bien plus court que le temps limite.

Malheureusement, nous pouvons constater que GMR n'est pas aussi compétitif que $ToulBar2$ pour la plupart des instances des séries $celar$, $scen$ et $graph$. Par contre, si l'on compare GMR avec les mêmes approches, mais implémentées dans notre solveur $AbsCon$, nous constatons alors que GMR surpasse la version EDAC d'$AbsCon$. Une explication plausible pouvant expliquer cette différence pourrait être les choix d'implémentation propres aux deux solveurs. Malgré tout, sur l'instance $graph$-05, même si GMR n'est pas aussi rapide que $ToulBar2$, notre approche est capable de trouver la borne optimale.

Nous avons également comparé notre approche gloutonne à une autre approche incomplète proposée par la bibliothèque INCOP (section 2.2.3) dans le solveur $ToulBar2$. Les résultats sont proposés dans les tableaux 4.13(a) (pour les instances $spot5$) et 4.13(b) (pour les instances $celar$). Nous constatons que les bornes retournées par INCOP sont dans la majorité des cas meilleures que celles retournées par notre approche. Évidemment, nous pouvons nous demander si ces deux approches sont bien comparables. Bien qu'il s'agisse de méthodes incomplètes, la bibliothèque INCOP exploite bon nombre de techniques sophistiquées de recherche locale, présentées dans la section 2.5, qui sont naturellement plus efficaces qu'une simple approche gloutonne. Malgré cette différence flagrante de stratégie, on peut toutefois noter que les bornes trouvées par notre approche sont relativement proches de celles retournées par INCOP et dans des temps très faibles. Exploiter notre méthode représente donc une approche intéressante si l'on souhaite obtenir pour un problème une borne de bonne qualité dans un temps relativement faible.

Concernant notre approche complète avec extraction de MUC, les expérimentations menées ont montré qu'elle n'est pas compétitive en l'état. Nous avons donc cherché à déterminer les raisons à l'origine de cette inefficacité et nous en sommes arrivés à trois constats. Le premier constat concerne le nombre de fronts à traiter par notre approche avant de trouver le premier front (CN) satisfaisable. Certes, une solution du premier front satisfaisable trouvé correspond à une solution optimale du WCN d'origine. Cependant, s'il s'agit d'un front "éloigné" du front d'origine, c'est à dire un front obtenu depuis le front initial après un nombre considérable de transformations, alors il est possible de ne pas obtenir de solution

Instances		AbsCon GMR	AbsCon EDAC	Toulbar2 EDAC
*spot*5-42	CPU	5.65	> 600	> 600
	UB	162,050	**161,050**	**161,050**
*spot*5-404	CPU	3.17	> 600	217
	UB	118	**114**	**114**
*spot*5-408	CPU	7	> 600	> 600
	UB	**6,235**	8,238	6,240
*spot*5-412	CPU	11.4	> 600	> 600
	UB	**33,403**	43,390	37,399
*spot*5-414	CPU	23.5	> 600	> 600
	UB	**40,500**	56,492	52,492
*spot*5-503	CPU	3.67	> 600	> 600
	UB	12,125	13,119	**12,117**
*spot*5-505	CPU	7.61	> 600	> 600
	UB	**22,266**	28,258	25,268
*spot*5-507	CPU	13.3	> 600	> 600
	UB	**30,417**	37,429	37,420
*spot*5-509	CPU	19.5	> 600	> 600
	UB	**37,469**	48,475	46,477
*spot*5-1401	CPU	45.6	> 600	> 600
	UB	**483,110**	513,097	516,095
*spot*5-1403	CPU	85	> 600	> 600
	UB	**493,265**	517,260	507,265
*spot*5-1407	CPU	334	> 600	> 600
	UB	**495,615**	517,623	507,633
*spot*5-1502	CPU	2.47	0.98	0.02
	UB	28,044	**28,042**	**28,042**
*spot*5-1504	CPU	40.9	> 600	> 600
	UB	**175,308**	204,314	198,318
*spot*5-1506	CPU	207	> 600	> 600
	UB	**388,556**	426,551	399,568

(a) instances *spot5*.

Instances		AbsCon GMR	AbsCon EDAC	ToulBar2 EDAC
graph-05	CPU	16.6	> 600	0.62
	UB	**221**	4,645	**221**
scen-06	CPU	88.5	> 600	485.4
	UB	3,616	12,013	**3,389**
scen-06-16	CPU	60.6	> 600	237.7
	UB	4,149	11,286	**3,277**
scen-06-18	CPU	59.2	> 600	102.4
	UB	3,640	8,723	**3,263**
scen-06-20	CPU	68.5	> 600	67.9
	UB	3,402	8,594	**3,163**
scen-06-30	CPU	32.6	177.7	1.10
	UB	2,208	**2,080**	**2,080**
scen-07	CPU	209.9	> 600	> 600
	UB	**426,423**	$31,230K$	505,731
scen-07_10K	CPU	28.7	> 600	> 600
	UB	**270K**	$17,000K$	$1,500K$
*celar*6-*sub*2	CPU	23.3	> 600	4.74
	UB	2,927	**2,746**	**2,746**
*celar*6-*sub*3	CPU	26.6	> 600	15.7
	UB	3,271	3,279	**3,079**
*celar*6-*sub*4	CPU	41.5	> 600	28.7
	UB	3,704	5,178	**3,230**
*celar*7-*sub*2	CPU	60.8	> 600	17.8
	UB	283,955	252,436	**173,252**
*celar*7-*sub*3	CPU	48.7	> 600	93.4
	UB	414,161	$1,342K$	**203,460**
*celar*7-*sub*4	CPU	57	> 600	221
	UB	272,945	302,541	**242,443**

(b) instances *celar*.

FIG. 4.12 – Temps CPU en secondes et borne trouvée avant le temps limite.

dans le temps imparti. Cela peut être gênant dans un contexte où une solution, qu'elle soit optimale ou non, est requise. Il faudrait donc pouvoir effectuer des "sauts" dans l'espace de recherche correspondant à l'ensemble des fronts. Ensuite, il suffirait d'utiliser le processus de "séparation et évaluation" avec le coût du front satisfaisable trouvé le plus récemment comme borne supérieure et ne pas traiter les fronts de coût supérieur ou égal qui ne permettront pas quoiqu'il arrive d'améliorer le front satisfaisable déjà trouvé. Le deuxième constat concerne le processus d'extraction des noyaux insatisfaisables minimaux. Nous sommes conscients que cette extraction est coûteuse et pénalisante, pourquoi ne pas reconsidérer cette recherche : les résultats seraient-ils meilleurs si on se contenter d'extraire un noyau non minimal dans le sens où nous l'avons exploité jusqu'à présent, et ainsi revoir la méthode d'extraction. Troisième et dernier constat, nous nous sommes aperçus que les mêmes noyaux insatisfaisables minimaux étaient

Instances		AbsCon GMR	Toulbar2 INCOP
*spot*5-42	CPU	5.65	> 600
	UB	162,050	**155,055**
*spot*5-404	CPU	3.17	> 600
	UB	118	**114**
*spot*5-408	CPU	7	> 600
	UB	6,235	**6,229**
*spot*5-412	CPU	11.4	> 600
	UB	33,403	**32,398**
*spot*5-414	CPU	23.5	> 600
	UB	**40,500**	40,509
*spot*5-503	CPU	3.67	> 600
	UB	12,125	**11,113**
*spot*5-505	CPU	7.61	> 600
	UB	22,266	**21,266**
*spot*5-507	CPU	13.3	> 600
	UB	30,417	**27,413**
*spot*5-509	CPU	19.5	> 600
	UB	37,469	**37,462**
*spot*5-1401	CPU	45.6	> 600
	UB	483,110	**479,109**
*spot*5-1403	CPU	85	> 600
	UB	493,265	**482,267**
*spot*5-1407	CPU	334	> 600
	UB	495,615	**484,609**
*spot*5-1502	CPU	2.47	> 600
	UB	28,044	**28,042**
*spot*5-1504	CPU	40.9	> 600
	UB	175,308	**166,318**
*spot*5-1506	CPU	207	> 600
	UB	388,556	**377,555**

(a) instances *spot5*.

Instances		AbsCon GMR	ToulBar2 INCOP
graph-05	CPU	16.6	> 600
	UB	**221**	243
scen-06	CPU	88.5	> 600
	UB	3,616	**3,466**
scen-06-16	CPU	60.6	> 600
	UB	4,149	**3,277**
scen-06-18	CPU	59.2	> 600
	UB	3,640	**3,274**
scen-06-20	CPU	68.5	> 600
	UB	3,402	**3,166**
scen-06-30	CPU	32.6	> 600
	UB	2,208	**2,080**
scen-07	CPU	209.9	> 600
	UB	426,423	**394,006**
scen-07_10K	CPU	28.7	535
	UB	**270,000**	360,000
*celar*6-*sub*2	CPU	23.3	> 600
	UB	2,927	**2,746**
*celar*6-*sub*3	CPU	26.6	> 600
	UB	3,271	**3,079**
*celar*6-*sub*4	CPU	41.5	> 600
	UB	3,704	**3,230**
*celar*7-*sub*2	CPU	60.8	> 600
	UB	283,955	**173,252**
*celar*7-*sub*3	CPU	48.7	> 600
	UB	414,161	**203,460**
*celar*7-*sub*4	CPU	57	> 600
	UB	272,945	**242,443**

(b) instances *celar*.

FIG. 4.13 – Temps CPU en secondes et borne trouvée avant le temps limite.

extraits plusieurs fois. En d'autres termes, au delà du coût que représente l'extraction d'un noyau insatisfaisable minimal, ce même travail peut donc être effectué plusieurs fois, ce qui pénalise clairement l'efficacité. Pourquoi donc ne pas enregistrer des informations sur les noyaux insatisfaisables minimaux extraits afin de les réutiliser lors des résolutions suivantes. Nous proposons donc une seconde approche de résolution complète que nous décrivons dans la section suivante. Il s'agit de l'approche complète décrite précédemment dans laquelle nous avons apporté des modifications pour pallier ces points faibles qui sont très certainement à l'origine de l'inefficacité de cette première approche complète proposée. Nous verrons d'ailleurs que les résultats de nos expérimentations nous confortent dans ce sens.

4.7 Deuxième approche

4.7.1 Recherche en profondeur d'abord

L'approche complète que nous avons proposée dans la section précédente est basée sur une recherche en largeur d'abord. En effet, la queue de priorité exploitée trie les fronts par coût et retourne à chaque fois le front de coût minimal. En d'autres termes, des fronts sont analysés, des successeurs de ces fronts sont générés dans le cas où ils ne sont pas satisfaisables et ils sont insérés dans la queue de priorité. Bien sûr, les successeurs d'un front ont des coûts relativement proches de ce front (car uniquement un coût modifié correspondant au coût de la contrainte relâchée), ils sont donc traités dans un temps relativement proche de celui du front d'origine. De ce fait, quand on observe l'ordre de parcours des nœuds, on constate qu'il ressemble à une recherche en largeur d'abord. La figure 4.14 illustre un exemple de ce parcours en largeur d'abord. Nous considérons un front initial $< 0, 0, 0 >$ à partir duquel nous cherchons à trouver un front satisfaisable. Les fronts traités sont grisés, et pour chacun de ces fronts est indiqué le coût (à sa droite) ainsi que la position (par rapport au traitement des autres fronts) à laquelle il a été traité (à sa gauche et en gras). Par exemple, le front $< 0, 0, 1 >$ a un coût égal à 1 et il est traité en deuxième position, après le front $< 0, 0, 0 >$ qui a un coût égal à 0 et qui est traité en première position. Dans ce petit exemple, nous admettons que le CN satisfaisable de coût minimal correspond au front $< 1, 1, 0 >$. Nous voyons bien à travers les fronts traités qui sont grisés un parcours de type largeur d'abord pour arriver à ce front $< 1, 1, 0 >$. À noter que les fronts encerclés sont des doublons et qu'ils ne sont donc pas insérés dans la queue de priorité.

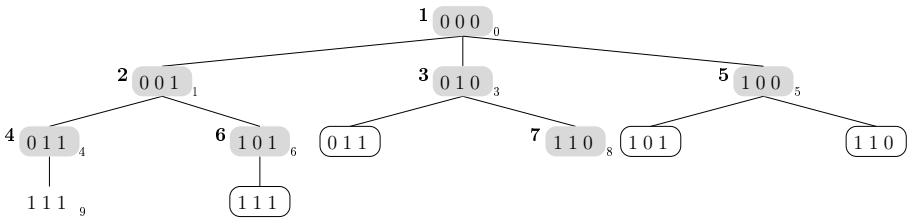


FIG. 4.14 – Arbre de parcours en largeur d'abord.

Le front satisfaisable $< 1, 1, 0 >$ est obtenu après avoir traité sept fronts : en effet, le front satisfaisable $< 1, 1, 0 >$ correspond au septième front traité. C'est un des premiers points faibles de l'approche complète que nous avons proposée précédemment. En effet, cette recherche en largeur d'abord peut prendre du temps avant de trouver un premier front de CN satisfaisable, même s'il s'agit de l'optimum. Nous pouvons constater qu'après avoir traité le front $< 0, 1, 1 >$, son voisin ($< 1, 1, 1 >$ avec un coût égal à 9) va être stocké dans la queue alors qu'il a un coût supérieur au coût du front de coût optimal (pour rappel, le front optimal est $< 1, 1, 0 >$ avec un coût de 8). Dans la nouvelle approche que nous proposons, nous utilisons une recherche en profondeur d'abord et pour ce faire, nous remplaçons la file par une pile. Cette pile contient donc les fronts et il est important de noter que, contrairement à la queue utilisée dans l'approche précédente, ces fronts ne sont pas triés par ordre de coût et plusieurs occurrences de ces fronts peuvent être présents. La figure 4.15 illustre, dans le même contexte que l'exemple précédent, ce parcours en profondeur d'abord et la figure 4.16 illustre l'état de la pile utilisée durant ce parcours.

Bien évidemment, le premier front traité est le front initial $< 0, 0, 0 >$. Ce front étant insatisfaisable, ses voisins sont générés dans l'ordre $< 0, 0, 1 >$, $< 0, 1, 0 >$, $< 1, 0, 0 >$ et sont placés sur la pile dans le même ordre. La figure 4.16 illustre l'état de la pile et les fronts grisés représentent les fronts extraits de la pile et traités : au début, la pile ne contenait que le front $< 0, 0, 0 >$, et ensuite ont été insérés les fronts $< 0, 0, 1 >$, $< 0, 1, 0 >$, puis $< 1, 0, 0 >$ qui correspond au front en haut de la pile. Ensuite, le

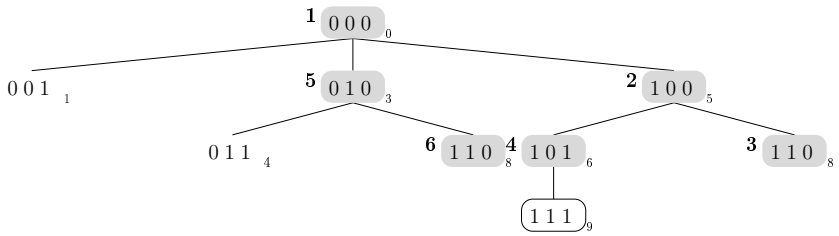


FIG. 4.15 – Arbre de parcours en profondeur d'abord.

front au dessus de la pile, c'est à dire $< 1, 0, 0 >$, est traité. Insatisfaisable, ses voisins sont générés, placés sur la pile et selon le même principe le front $< 1, 1, 0 >$ est extrait de la pile et traité. Le premier front satisfaisable (même s'il n'y a aucune garantie qu'il s'agisse d'un front de coût optimal) est ainsi trouvé seulement après avoir traité 3 fronts. De plus, si nous continuons le processus, le front $< 1, 1, 1 >$ de coût égal à 9 rencontré par la suite n'aurait pas été traité : en effet, par mécanisme de séparation et évaluation, il n'aurait pas été stocké dans la pile (donc ce front est encerclé sur la figure) car sont coût (9) est supérieur ou égal au coût du dernier front satisfaisable trouvé (en l'occurrence le front $< 1, 1, 0 >$ de coût 8).

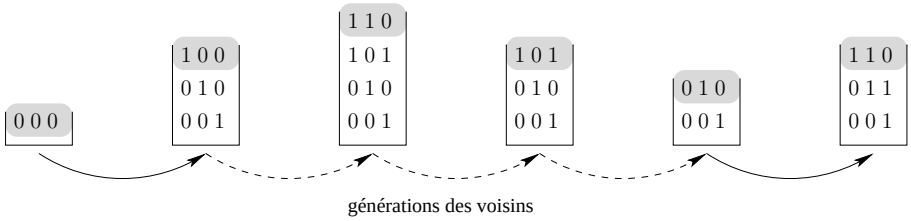


FIG. 4.16 – État de la pile de fronts lors d'une recherche en profondeur d'abord.

Sans aucune garantie que ce premier front satisfaisable trouvé ne corresponde à un front optimal, il a permis malgré tout de trouver une solution assez rapidement. Toutefois, il est important de noter que cette approche de recherche en profondeur d'abord est bien complète : en effet, elle parcourt la totalité de l'arbre de recherche dans le sens où tous les fronts qui sont insérés dans la pile sont traités.

4.7.2 Algorithme d'approche complète

Notre nouvelle approche est présentée dans l'algorithme completeSearchDFS(), Algorithme 35, dans lequel la pile S présentée dans le paragraphe précédent remplace la queue de priorité et va permettre d'effectuer une recherche en profondeur d'abord.

Dans le cadre d'une recherche en profondeur d'abord, le raisonnement est bien sûr différent. Dans l'approche précédente, dès que nous trouvions un front correspondant à un CN satisfaisable, les fronts étant traités par ordre de coût croissant, nous savions alors qu'une solution de ce CN correspondait alors à une solution de coût optimal dans le WCN et nous pouvions arrêter le processus. Ce n'est plus le cas en utilisant la pile car les éléments n'y sont pas triés. Pour être sûr d'avoir trouvé le front satisfaisable de coût minimal, tous les fronts présents dans la pile ainsi que les fronts pas encore stockés ayant un coût strictement inférieur au coût de la meilleure solution trouvée jusqu'à présent, doivent obligatoirement être considérés. C'est l'une des différences majeures de cet algorithme avec celui présenté dans la section précédente. La nouvelle boucle principale doit traiter tous les fronts présents dans la pile (via la fonction pop()) sans quoi la complétude ne peut être assurée. De plus, le coût minimal parmi les fronts prouvés

Algorithm 35: completeSearchDFS(W : WCN) : sol

```
1  foreach w ∈ cons(W) do
2      f[w] ← 0;
3  S ← {f} ;
4  while not S.empty() do
5      f ← S.pop() ;
6      if cost(f) ≠ k then
7          P ← toCN_=(W, f) ;
8          sol ← solveCN(P) ;
9          if sol ≠ ⊥ then
10             solOptim ← sol ;
11             k ← cost(f) ;
12         else
13             M ← extractMUC(P) ;
14             foreach w ∈ cons(W, M) do
15                 if f[w] < w.layers.length − 1 then
16                     f' ← f ;
17                     f'[w] ← f'[w] + 1 ;
18                     if cost(f') ≠ k then
19                         S.push(f') ;
20 return solOptim ;
```

satisfaisables constituant une borne supérieure, seuls les fronts ayant un coût strictement inférieur à cette borne doivent être ajoutés dans la pile pour être considérés (test à la ligne 6). C'est pour cela que seuls les nouveaux fronts dont le coût respecte cette condition sont ajoutés (via la fonction push()) dans la pile (ligne 19) et quand on dépile et que le coût du front extrait est supérieur ou égal au coût de la meilleure solution connue, on peut l'ignorer.

4.7.3 Noyaux insatisfaisables minimaux par rapport aux variables

Un autre changement que nous avons apporté à notre approche concerne l'extraction des noyaux insatisfaisables, et plus précisément la nature du noyau insatisfaisable extrait. Dans l'approche précédente il s'agissait d'extraire un noyau insatisfaisable minimal par rapport au nombre de contraintes contenues : en d'autres termes, retirer une et une seule contrainte de ce noyau permet de le rendre satisfaisable. Cette fois-ci nous allons chercher à extraire un noyau insatisfaisable minimal par rapport au nombre de variables contenues : retirer une et une seule variable couverte par au moins une contrainte de ce noyau, c'est à dire extraire aussi du noyau toutes les contraintes couvrant cette variable, permet de rendre ce noyau satisfaisable. Cette extraction est toujours réalisée par l'algorithme extractMUC(P) mais la minimisation est assurée par rapport aux variables comme expliquée dans la section 4.5.2.

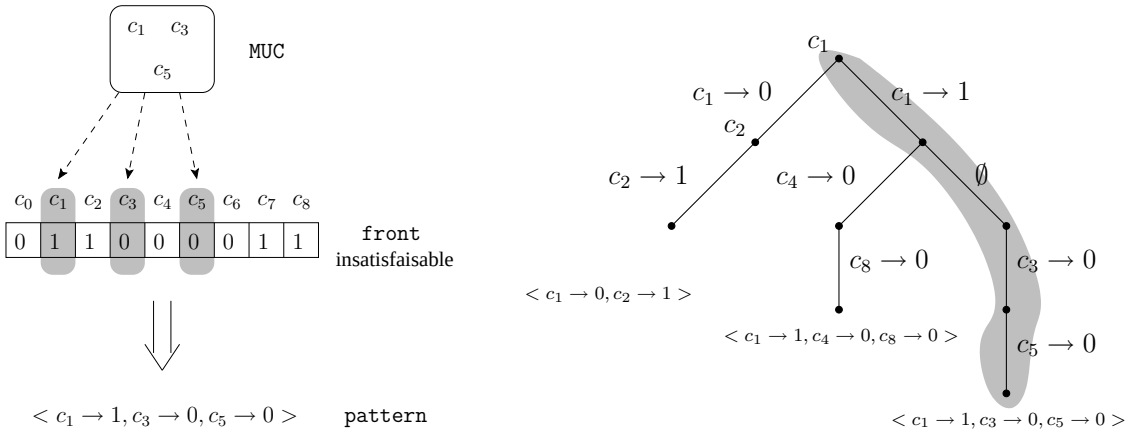
4.7.4 Apprentissage de noyaux insatisfaisables

Comme nous l'avons vu dans l'approche proposée précédemment, beaucoup de fronts sont traités. Pour chaque front une résolution va être réalisée, avec à chaque fois que le CN correspondant est insatisfaisable l'extraction d'un noyau insatisfaisable minimal afin de le relâcher. Nous savons que l'extraction

d'un noyau insatisfaisable minimal est coûteuse, et vu que le nombre de fronts peut être considérable, on doit veiller à réduire ce travail si l'on veut être compétitif. Le but de notre réflexion était donc d'éviter ces recherches de noyaux insatisfaisables minimaux. Nous nous sommes alors aperçus que les mêmes noyaux insatisfaisables minimaux pouvaient être extraits plusieurs fois (avec le calcul que ça représente) à partir de fronts différents. Nous sommes donc partis sur l'idée d'enregistrer ces noyaux insatisfaisables déjà rencontrés, afin de les détecter dans les fronts suivants sans avoir à les recalculer. Les fronts contenant un MUC déjà identifié ne seront pas traités et les voisins de ces fronts seront directement générés. Clairement, apprendre des noyaux insatisfaisables déjà rencontrés permet de réduire les efforts de recherche nécessaires durant l'extraction de noyaux insatisfaisables minimaux.

Base de patterns de noyaux insatisfaisables

Lorsque nous traitons un front qui s'avère être insatisfaisable, un MUC peut donc en être extrait. Considérons la figure 4.17(a). Nous imaginons que le front $< 0, 1, 1, 0, 0, 0, 0, 1, 1 >$ correspond à un CN insatisfaisable et que le MUC extrait se compose des contraintes c_1, c_3 et c_5. Nous définissons le *pattern* correspondant à un MUC comme la structure qui associe à chaque contrainte du MUC, via son identifiant, l'indice de la strate sélectionnée pour cette contrainte dans le front insatisfaisable. Un pattern sera noté sous la forme $< c_0 \rightarrow s_0, \ldots, c_q \rightarrow s_q >$ où par exemple s_0 représente l'indice de la strate associée à la contrainte c_0 dans ce pattern. Dans la figure 4.17(a), le pattern correspondant au MUC est $< c_1 \rightarrow 1, c_3 \rightarrow 0, c_5 \rightarrow 0 >$. Ce pattern couvre trois contraintes (celles du MUC) et associe la strate d'indice 1 à la contrainte d'identifiant 1 (c_1), la strate d'indice 0 à la contrainte d'identifiant 3 (c_3) et la strate d'indice 0 à la contrainte d'identifiant 5 (c_5). Nous stockons les différents patterns calculés au cours du processus dans une base de patterns représentée sous la forme d'un arbre illustré dans la figure 4.17(b). Un nœud de cet arbre correspond à une contrainte et les arêtes correspondent à des couples (`identifiant contrainte` $\rightarrow$ `indice strate`). Toutes les arêtes qui partent d'un nœud référencent la contrainte associée à ce nœud. Pour signaler qu'une contrainte n'est pas couverte par un pattern, nous trouverons l'arête $\emptyset$ au niveau du branchement pour cette contrainte. Chaque branche de cet arbre allant de la racine jusqu'à une feuille (qui ne représente pas une contrainte mais plutôt un point de terminaison) représente un pattern qui couvre les contraintes associées aux nœuds de cette branche et qui se compose de l'ensemble de ses arêtes.



(a) Pattern extrait depuis un front insatisfaisable.

(b) Vue partielle d'une base de patterns.

FIG. 4.17 – Enregistrement de noyaux insatisfaisables connus via leurs patterns.

Considérons par exemple dans la figure 4.17(b) l'insertion dans l'arbre du pattern $< c_1 \rightarrow 1, c_3 \rightarrow$

$0, c_5 \rightarrow 0 >$ (tout juste trouvé) dans l'arbre contenant déjà les patterns $< c_1 \rightarrow 0, c_2 \rightarrow 1 >$ et $< c_1 \rightarrow 1, c_4 \rightarrow 0, c_8 \rightarrow 0 >$. Depuis la racine de l'arbre référençant la contrainte c_1, les premières arêtes, à savoir $c_1 \rightarrow 0$ et $c_1 \rightarrow 1$, sont considérées. Comme la première association `contrainte` $\rightarrow$ `strate` dans le pattern correspond à $c_1 \rightarrow 1$, nous suivons l'arête correspondante et nous considérons son nœud d'arrivée qui représente la contrainte c_4. Une seule arête part du nœud correspondant à c_4, il s'agit de $c_4 \rightarrow 0$. La contrainte c_4 n'étant pas présente dans le pattern à insérer, nous ajoutons une nouvelle arête $\emptyset$ partant du nœud c_4 dans laquelle nous branchons. Enfin, nous ajoutons les deux arêtes suivantes, à savoir $c_3 \rightarrow 0$ et $c_5 \rightarrow 0$, depuis deux nouveaux nœuds que nous ajoutons également et qui correspondent aux deux dernières contraintes couvertes par le pattern, à savoir c_3 et c_5. Cet arbre est clairement construit de manière gloutonne et une perspective intéressante évidente serait de ré-ordonner de temps en temps cet arbre, voir même subsumer certaines branches qui le permettent. À noter également qu'aucun test n'est nécessaire pour savoir si un pattern est déjà présent ou non : les doublons de patterns dans l'arbre sont impossibles, car lorsqu'on trouve un MUC déjà présent, on n'insère pas le pattern correspondant.

Comme nous le disions précédemment, ces patterns vont permettre de vérifier l'insatisfaisabilité de fronts en vérifiant l'inclusion de ces patterns dans ces fronts. Nous abordons l'utilisation de ces patterns dans le paragraphe suivant.

Exploitation des patterns existants

Lorsqu'un front est extrait de la pile, au lieu de chercher directement à résoudre le CN correspondant et extraire un MUC dans le cas où il est insatisfaisable, on va vérifier dans un premier temps si ce front ne correspond pas à un pattern de MUC déjà extrait et qui est donc enregistré dans l'arbre des patterns. L'arbre des patterns est parcouru par l'algorithme existsPattern(), Algorithme 36.

Algorithm 36: existsPattern(n : node, f : front) : boolean

```
1  if n is a leaf then
2      Record the constraints found from the Root in the pattern ;
3      return true
4  c ← constraint corresponding to the node n ;
5  if an edge (c → s) from n exists and f[c] = s then
6      Let n' the arrival node of this edge ;
7      if existsPattern(n', f) then
8          return true
9  if an edge ∅ from n exists then
10     Let n' the arrival node of this edge ;
11     if existsPattern(n', f) then
12         return true
13 return false ;
```

Depuis la racine, à chaque nœud de l'arbre associé à une contrainte c_i, l'algorithme sélectionne l'arête $c_i \rightarrow s_i$ qui correspond à la strate s_i sélectionnée dans le front pour la contrainte c_i et la suit. De plus, l'algorithme suit également l'arête $\emptyset$ partant de la contrainte c_i si elle existe, car potentiellement même si la contrainte c_i n'est pas couverte par le pattern en cours d'exploration, la suite du parcours de cette branche peut amener à trouver un autre pattern. Lorsqu'une feuille est atteinte, cela signifie qu'un pattern a été trouvé. En effet, un pattern déjà enregistré est inclus dans un front si et seulement si un

chemin est trouvé dans l'arbre depuis la racine jusqu'à une des feuilles. Considérons à présent un petit exemple de recherche de pattern illustré dans la figure 4.18.

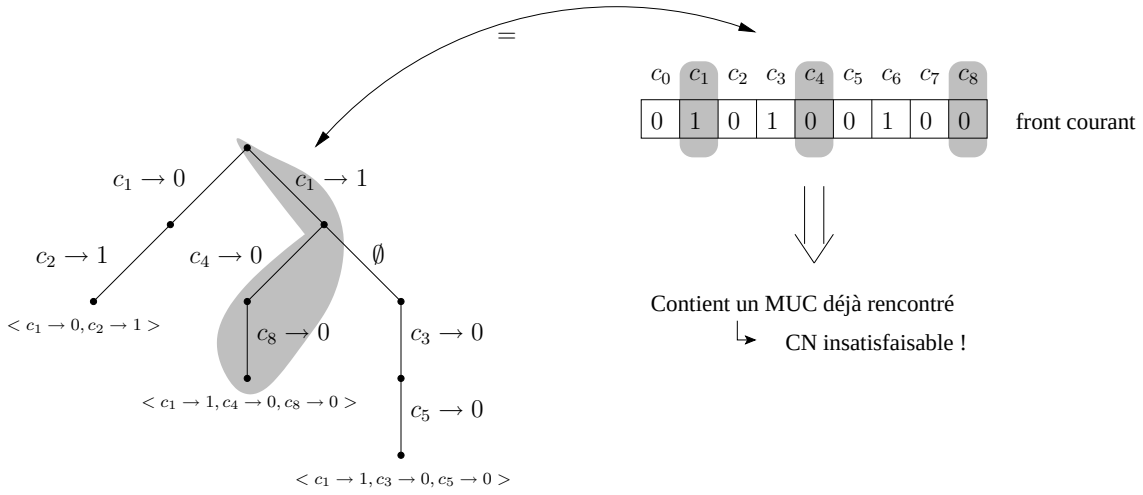


FIG. 4.18 – Recherche d'un pattern inclus dans le front courant.

Dans la figure 4.18, on cherche si un pattern déjà enregistré dans la base est inclus dans le front courant $< 0, 1, 0, 1, 0, 0, 1, 0, 0 >$. Le premier nœud analysé (racine) correspond à la contrainte c_1 et c'est l'arête $c_1 \rightarrow 1$ qui est sélectionnée car dans le front courant, c'est la strate d'indice 1 qui est associée à la contrainte c_1. À noter qu'il n'y a pas d'arête $\emptyset$ sortant du nœud c_1. Le second nœud analysé correspond à la contrainte c_4 : les arêtes $c_4 \rightarrow 0$ (car dans le front courant, c'est bien la strate d'indice 0 qui est associée à la contrainte c_4) et $\emptyset$ sont sélectionnées et suivies. Lors du branchement sur l'arête $c_4 \rightarrow 0$, nous arrivons ensuite sur le nœud correspondant à la contrainte c_8 : une seule arête émane de ce nœud, à savoir $c_8 \rightarrow 0$. Cette arête est donc suivie (dans le front courant, c'est bien la strate d'indice 0 qui est associée à la contrainte c_8) et comme elle mène à une feuille, alors nous avons trouvé un pattern (complet) qui est inclus dans le front. En d'autres termes, sans avoir à le résoudre, nous savons que le CN correspondant au front courant est insatisfaisable : en effet, il contient un MUC constitué des contraintes c_1, c_4 et c_8 associées respectivement aux strates 1, 0 et 0 et qui a déjà été extrait depuis un front précédent. Il est utile de préciser que si cela avait été $c_8 \rightarrow 1$ dans le front au lieu de $c_8 \rightarrow 0$, alors un retour en arrière avant le choix $c_4 \rightarrow 0$ aurait été effectué, correspondant en fait au branchement sur $\emptyset$ à partir du nœud c_4, suivi de $c_3 \rightarrow 0$ où la recherche se serait arrêtée.

À noter que dès qu'on a trouvé un pattern qui est inclus dans le front courant, on peut s'arrêter là. Nous savons alors qu'il y a déjà au moins un MUC présent dans le CN associé au front courant et que quoi qu'il arrive, il sera insatisfaisable. Dès lors que ce pattern a été trouvé dans la base, nous allons l'utiliser.

Les voisins de ce front vont être créés par rapport aux contraintes englobées dans ce pattern, c'est à dire qu'un voisin de ce front va être généré pour chaque contrainte en relâchant son coût autorisé. La figure 4.19 illustre le fonctionnement. Nous savons donc que le front $< 0, 1, 0, 1, 0, 0, 1, 0, 0 >$ est insatisfaisable car le pattern $< c_1 \rightarrow 1, c_4 \rightarrow 0, c_8 \rightarrow 0 >$ est inclus. De plus, nous considérons dans notre exemple que chacune des contraintes admet deux strates d'indices 0 et 1. Ainsi, un premier voisin du front va être généré en incrémentant la strate autorisée pour la contrainte c_4 (nous ne pouvons pas relâcher la contrainte c_1 car c'est déjà la dernière strate, à savoir d'indice 1, qui est sélectionnée au niveau du front). Un deuxième front est obtenu de la même façon en relâchant la contrainte c_8.

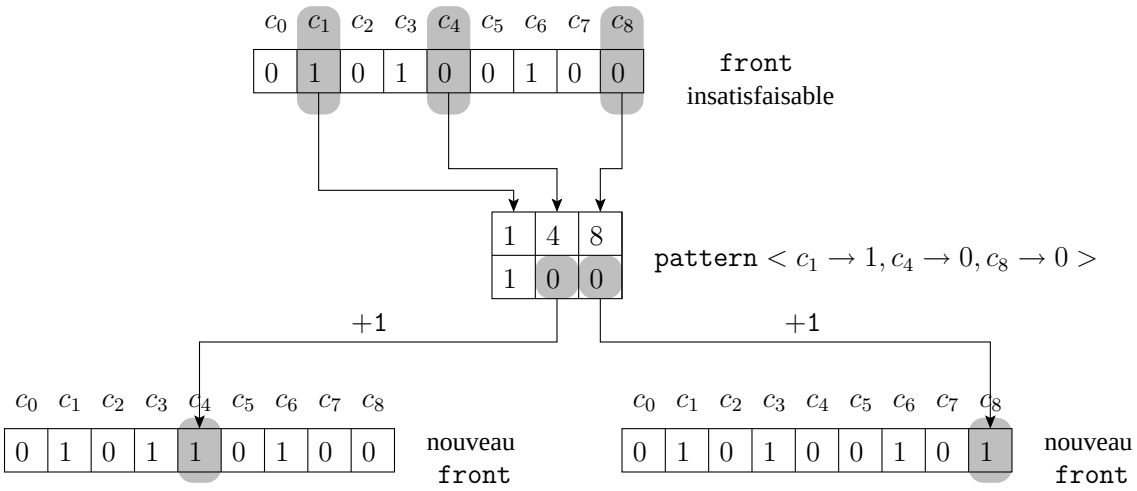


FIG. 4.19 – Exploitation d'un pattern inclus dans le front courant.

Détection d'insatisfaisabilités triviales : les cliques

Un autre constat que nous avons fait, au delà se s'apercevoir que beaucoup de MUC étaient extraits plusieurs fois, c'est qu'un grand nombre des MUC extraits étaient relativement petits et assez souvent de taille 3. Après avoir analysé plus en détail ces MUC, nous nous sommes aperçus que ces MUC de taille 3 correspondaient généralement à des *3-cliques* de contraintes. Une $3-clique$ de contraintes est un ensemble de 3 contraintes qui sont liées deux à deux par (au moins) une variable commune. Un exemple de $3-clique$, constituée des contraintes w_{xy}, w_{yz} et w_{xz}, est illustrée en haut à gauche de la figure 4.20.

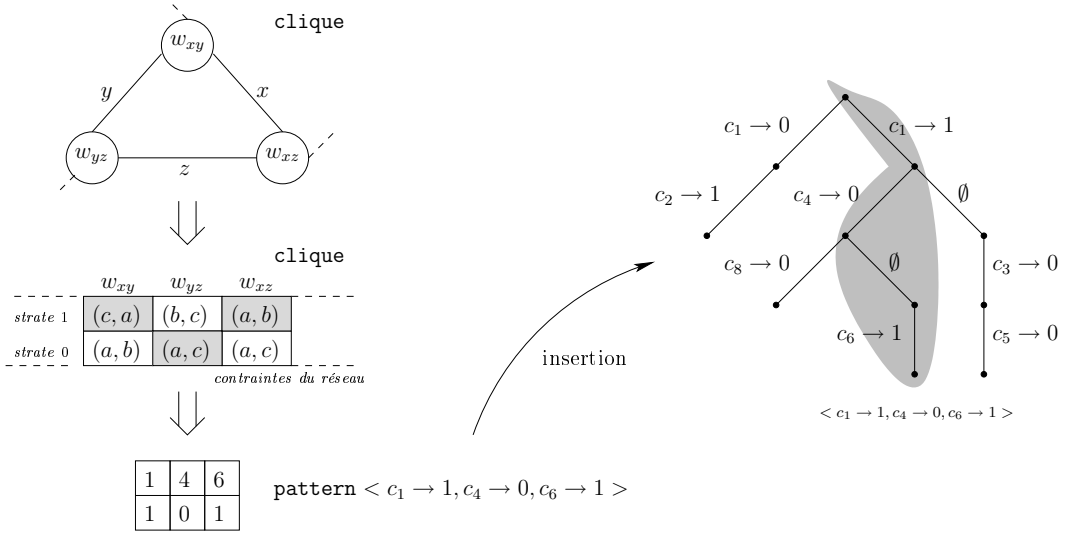


FIG. 4.20 – Création et enregistrement d'un pattern correspondant à une clique.

Si l'on s'attarde sur les différentes strates de ces trois contraintes, on peut s'apercevoir (en grisé) que si les strates 1, 0 et 1 sont sélectionnées dans un front pour les contraintes w_{xy}, w_{yz} et w_{xz}, alors le CN correspondant sera forcément insatisfaisable du fait de l'incompatibilité des valeurs pour la variable z. En pré-traitement, l'idée est donc de déterminer toutes les cliques de taille 3 et de tester la cohérence de chemin [Mackworth, 1977] pour toutes les combinaisons possibles de strates pour ces contraintes.

Pour chaque combinaison de strates insatisfaisable trouvée, nous insérons un nouveau pattern dans notre arbre de patterns de noyaux insatisfaisables comme illustré dans la partie droite de la figure 4.20. Les contraintes du pattern correspondent aux contraintes de la clique et les strates correspondent aux strates des combinaisons insatisfaisables. Le pattern $< c_1 \rightarrow 1, c_4 \rightarrow 0, c_6 \rightarrow 1 >$ (c_1, c_4 et c_6 correspondant respectivement à w_{xy}, w_{yz} et w_{xz}) est alors ajouté dans l'arbre qui est ainsi alimenté avant de commencer à parcourir les différents fronts. De plus, ces patterns étant relativement petits (composés de 3 contraintes), il est intéressant de voir qu'ils seront plus généralistes que les autres patterns et donc potentiellement inclus dans plus de fronts, et ainsi vérifier leur inclusion sera plus rapide vu leur petite taille.

Malheureusement, nous avons pu voir en pratique qu'éviter les MUC de taille 3 ne permettaient pas d'améliorer nos résultats. Bien évidemment, nous avons constaté que des premiers fronts représentant des CN satisfaisables étaient obtenus plus rapidement, du fait que tous les MUC dus à des cliques insatisfaisables n'avaient plus à être calculés. Cependant, c'est simplement le parcours de recherche au sein des fronts qui s'est avéré être modifié : au lieu de rencontrer les MUC les plus difficiles à calculer relativement tard, comme c'est le cas avec les approches proposées dans les sections précédentes, ils sont traités plus tôt. Au delà de cette idée, nous avons décidé de réfléchir plus en détail à des causes d'insatisfaisabilité facilement décelables et pas trop coûteuses pour éviter d'avoir à calculer des noyaux insatisfaisables. Nous avons ensuite tenté de généraliser cette détection d'insatisfaisabilité depuis les cliques. Une autre idée que nous avons tenté a été de vérifier, avant de traiter chaque front (durant le processus et non plus durant le pré-traitement), que chaque variable possède au moins une valeur supportée dans chaque contrainte, c'est à dire qu'il existe un tuple support pour cette valeur dans la strate sélectionnée dans un front pour cette contrainte. Il s'agit d'une sorte d'une cohérence généralisée pour les valeurs. En effet, si une variable ne possède pas une valeur support, alors on sait que quoiqu'il arrive le front ne sera pas satisfaisable. La figure 4.21 illustre un exemple de détection d'insatisfaisabilité d'un front à partir des valeurs des domaines des variables. On constate par exemple que pour le front sélectionnant respectivement les strates d'indice 1, 0 et 1 (grisées) pour les contraintes w_{xy}, w_{yz} et w_{xz}, la variable z ne possède pas une même valeur supportée par toutes les contraintes (en l'occurrence w_{yz} et w_{xz}) : la valeur (z, c) est la seule valeur supportée par la contrainte w_{yz} (par le tuple (a, c)) mais elle n'est pas supportée par la contrainte w_{xz} dans laquelle seule la valeur (z, b) est supportée (par le tuple (a, b)). Si un front s'avère insatisfaisable, on ne le traite pas : on génère directement ses voisins par rapport à l'intégralité des contraintes du réseau.

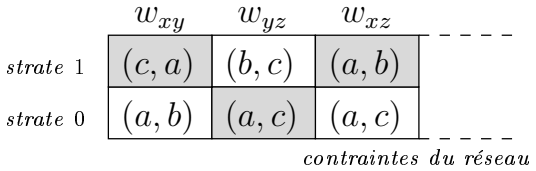


FIG. 4.21 – Création et enregistrement d'un pattern correspondant à une clique.

Cette cohérence est relativement facile et peu coûteuse à vérifier, malheureusement le cas n'est pas souvent vérifié dans les instances utilisées pour nos tests. Ceci s'explique assez facilement. Dans les instances testées, les strates de coût minimal pour chacune des contraintes correspondent à la strate du coût par défaut. Autrement dit, ces strates contiennent un très grand nombre de tuples et de ce fait un support est quasiment toujours trouvé pour chaque valeur. Pour les diverses idées que nous avons tentées, malheureusement les résultats n'ont pas été bons. Cependant, nous pensons réellement qu'il serait intéressant de déterminer des causes d'insatisfaisabilité basiques au sein des fronts pour éviter d'avoir à calculer des noyaux insatisfaisables minimaux.

L'algorithme completeSearch2(), Algorithme 37, reprend donc le principe de base de recherche DFS de l'algorithme completeSearchDFS(), Algorithme 35, auquel on a rajouté la recherche des noyaux déjà rencontrés (à la ligne 7) et l'ajout des noyaux jamais rencontrés (à la ligne 29).

Algorithm 37: completeSearch2(W : WCN) : sol

```
1  foreach w ∈ cons(W) do
2      f[w] ← 0;
3  S ← {f} ;
4  while not S.empty() do
5      f ← S.pop() ;
6      if cost(f) ≠ k then
7          P ← Search pattern already recorded in the patterns Tree and included in f ;
8          if P ≠ ⊥ then
9              foreach w ∈ cons(P) do
10                 if f[w] ≠ w.layers.length − 1 then
11                     f' ← f ;
12                     f'[w] ← f'[w] + 1 ;
13                     if cost(f') ≠ k then
14                         S.push(f') ;
15         else
16             P ← toCN_=(W, f) ;
17             sol ← solveCN(P) ;
18             if sol ≠ ⊥ then
19                 solOptim ← sol ;
20                 k ← cost(f) ;
21             else
22                 M ← extractMUC(P) ;
23                 foreach w ∈ cons(W, M) do
24                     if f[w] ≠ w.layers.length − 1 then
25                         f' ← f ;
26                         f'[w] ← f'[w] + 1 ;
27                         if cost(f') ≠ k then
28                             S.push(f') ;
29                 Insert pattern obtained from M in the patterns Tree ;
30 return solOptim ;
```

4.7.5 Résultats expérimentaux

Les expérimentations ont été réalisées dans les mêmes conditions matérielles que les expérimentations précédentes. Là aussi, le temps limite alloué a été fixé à 600 secondes. Nous avons choisi tout d'abord de comparer notre approche, notée CMR (CMR pour Complete MUC Relaxation) aux approches maintenant EDAC et proposées par $AbsCon$ et $ToulBar2$ (version 0.9.4.0). Le tableau 4.22(a) représente les résultats obtenus pour la série d'instances $spot5$. Le temps CPU pour résoudre chaque instance

Instances		AbsCon CMR	AbsCon EDAC	ToulBar2 EDAC
*spot*5-42	CPU	> 600	> 600	> 600
	UB	$170,050$	$\mathbf{161,050}$	$\mathbf{161,050}$
*spot*5-404	CPU	> 600	> 600	217
	UB	116	$\mathbf{114}$	$\mathbf{114}$
*spot*5-408	CPU	> 600	> 600	> 600
	UB	$\mathbf{6,234}$	$8,238$	$6,240$
*spot*5-412	CPU	> 600	> 600	> 600
	UB	$\mathbf{33,402}$	$43,390$	$37,399$
*spot*5-414	CPU	> 600	> 600	> 600
	UB	$\mathbf{39,499}$	$56,492$	$52,492$
*spot*5-503	CPU	> 600	> 600	> 600
	UB	$12,125$	$13,119$	$\mathbf{12,117}$
*spot*5-505	CPU	> 600	> 600	> 600
	UB	$\mathbf{22,265}$	$28,258$	$25,268$
*spot*5-507	CPU	> 600	> 600	> 600
	UB	$\mathbf{30,418}$	$37,429$	$37,420$
*spot*5-509	CPU	> 600	> 600	> 600
	UB	$\mathbf{36,469}$	$48,475$	$46,477$
*spot*5-1401	CPU	> 600	> 600	> 600
	UB	$\mathbf{487,113}$	$513,097$	$516,095$
*spot*5-1403	CPU	> 600	> 600	> 600
	UB	$\mathbf{488,270}$	$517,260$	$507,265$
*spot*5-1407	CPU	> 600	> 600	> 600
	UB	$\mathbf{490,628}$	$517,623$	$507,633$
*spot*5-1504	CPU	> 600	> 600	> 600
	UB	$\mathbf{180,323}$	$204,314$	$198,318$
*spot*5-1506	CPU	> 600	> 600	> 600
	UB	$\mathbf{389,573}$	$426,551$	$399,568$
*spot*5-5	CPU	> 600	> 600	> 600
	UB	$\mathbf{279}$	286	$\mathbf{279}$
*spot*5-28	CPU	> 600	> 600	> 600
	UB	$\mathbf{273,105}$	$284,105$	$282,105$
*spot*5-1405	CPU	> 600	> 600	> 600
	UB	$\mathbf{498,450}$	$517,450$	$507,457$

(a) instances *spot5*.

Instances		AbsCon CMR	ToulBar2 RDS
*spot*5-42	CPU	> 600	1.3
	UB	$170,050$	$\mathbf{155,050}$
*spot*5-404	CPU	> 600	0.5
	UB	116	$\mathbf{114}$
*spot*5-408	CPU	> 600	6.7
	UB	$6,234$	$\mathbf{6,228}$
*spot*5-412	CPU	> 600	15.5
	UB	$33,402$	$\mathbf{32,381}$
*spot*5-414	CPU	> 600	41.3
	UB	$39,499$	$\mathbf{38,478}$
*spot*5-503	CPU	> 600	0.9
	UB	$12,125$	$\mathbf{11,113}$
*spot*5-505	CPU	> 600	6.7
	UB	$22,265$	$\mathbf{21,253}$
*spot*5-507	CPU	> 600	31.9
	UB	$30,418$	$\mathbf{27,390}$
*spot*5-509	CPU	> 600	55.7
	UB	$36,469$	$\mathbf{36,446}$
*spot*5-1401	CPU	> 600	> 600
	UB	$\mathbf{487,113}$	–
*spot*5-1403	CPU	> 600	> 600
	UB	$\mathbf{488,270}$	–
*spot*5-1407	CPU	> 600	> 600
	UB	$\mathbf{490,628}$	–
*spot*5-1504	CPU	> 600	> 600
	UB	$\mathbf{180,323}$	–
*spot*5-1506	CPU	> 600	> 600
	UB	$\mathbf{389,573}$	–
*spot*5-5	CPU	> 600	37.4
	UB	279	$\mathbf{261}$
*spot*5-28	CPU	> 600	> 600
	UB	$\mathbf{273,105}$	–
*spot*5-1405	CPU	> 600	> 600
	UB	$\mathbf{498,450}$	–

(b) instances *spot5*.

FIG. 4.22 – Temps CPU en secondes et borne trouvée avant le temps limite.

est donné ainsi que la borne (UB) obtenue dans le temps imparti. Nous pouvons donc voir que notre approche retourne pour la grande majorité des instances de meilleures bornes que celles retournées par les approches standard complètes maintenant EDAC. Les bornes obtenues sont moins bonnes pour les instances relativement petites ($spot5$-42, $spot5$-404, $spot5$-503), c'est à dire possédant peu de contraintes (< 1400). Malheureusement, concernant les autres familles de problèmes $pedigree$ et notamment $celar$, pour laquelle nous pouvons observer les résultats dans la figure 4.23, les bornes retournées par notre approche sont moins bonnes. Une question que l'on peut se poser est pourquoi notre méthode semble plutôt bien fonctionner pour les instances $spot5$. Il s'agit de problèmes réels très structurés : facilement décomposables en noyaux insatisfaisables dont les résolutions indépendantes permettent d'atteindre rapidement des solutions. Si on considère par exemple l'instance $spot5$-507 dont le graphe primaire de contraintes est illustré dans la figure 4.24, on y distingue clairement des sous-réseaux qui semblent se détacher les

		AbsCon		ToulBar2
Instances		CMR	EDAC	EDAC
graph-05	CPU	> 600	> 600	0.62
	UB	–	4,645	**221**
scen-06	CPU	> 600	> 600	485.4
	UB	–	12,013	**3,389**
scen-06-16	CPU	> 600	> 600	237.7
	UB	–	11,286	**3,277**
scen-06-18	CPU	> 600	> 600	102.4
	UB	148,686	8,723	**3,263**
scen-06-20	CPU	> 600	> 600	67.9
	UB	–	8,594	**3,163**
scen-06-30	CPU	> 600	177.7	1.10
	UB	–	**2,080**	**2,080**
scen-07	CPU	> 600	> 600	> 600
	UB	–	$31,230K$	**505,731**
scen-07_10K	CPU	> 600	> 600	> 600
	UB	–	$17,000K$	**1,500K**
*celar*6-*sub*2	CPU	> 600	> 600	4.74
	UB	–	**2,746**	**2,746**
*celar*6-*sub*3	CPU	> 600	> 600	15.7
	UB	44,118	3,279	**3,079**
*celar*6-*sub*4	CPU	> 600	> 600	28.7
	UB	49,164	5,178	**3,230**
*celar*7-*sub*2	CPU	> 600	> 600	17.8
	UB	–	252,436	**173,252**
*celar*7-*sub*3	CPU	> 600	> 600	93.4
	UB	–	$1,342K$	**203,460**
*celar*7-*sub*4	CPU	> 600	> 600	221
	UB	–	302,541	**242,443**

FIG. 4.23 – Temps CPU en secondes et borne trouvée avant le temps limite (instances *celar*).

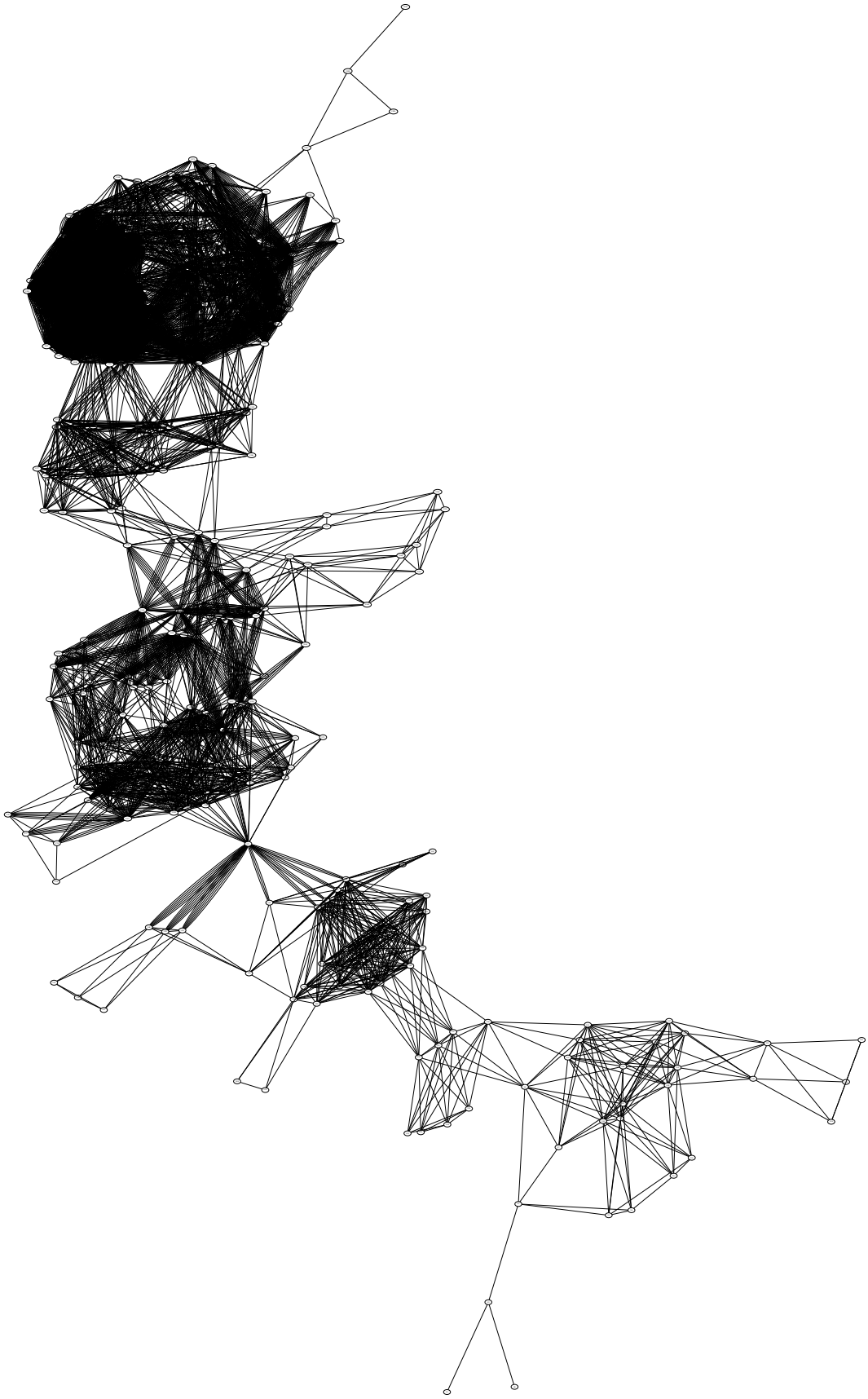


FIG. 4.24 – Graphe primaire de contraintes du réseau de contraintes associé à l'instance $spot5$-507.

uns des autres et qui sont constitués de variables fortement connectées entre elles. Bien évidemment, d'une part cela nous intéresse particulièrement car l'extraction de noyaux insatisfaisables relativement indépendants (et donc traitables de manière relativement indépendante les uns par rapport aux autres par nos relaxations) est envisageable. D'autre part, on imagine très bien dans ce contexte l'exploitation possible de la décomposition arborescente de manière efficace comme décrite dans la section 2.4.2. De plus, les contraintes dans les instances $spot5$ ne se composent que de deux strates, contrairement aux instances $celar$ qui peuvent en contenir beaucoup plus (> 10). Comme nous l'avons vu, notre approche consiste à satisfaire les noyaux insatisfaisables rencontrés en relâchant des contraintes, c'est à dire en autorisant les tuples de la strate suivante. On imagine naturellement que la satisfaction de ces noyaux peut être plus rapide à atteindre dans le cas où les contraintes sont découpées en peu de strates. Nous avons voulu ensuite comparer, toujours sur la série d'instances $spot5$, notre approche CMR à la méthode RDS (pour *Russian Doll Search*) proposée dans [Verfaillie *et al.*, 1996] qui s'avère être efficace pour cette série d'instances. Inspirée de la décomposition arborescente présentée dans la section 2.4.2, cette méthode consiste à résoudre un WCN initial en le décomposant en n sous-réseaux "imbriqués" à résoudre (n étant le nombre de variables dans le WCN initial). À partir d'un ordre établi sur les variables pour toute la résolution, le premier sous-réseau est composé d'une unique variable (la dernière selon l'ordre) : plus formellement, le $i^{\text{ème}}$ sous-WCN est composé de $n - i + 1$ variables (les $n - i + 1$ dernières). Ainsi, les sous-réseaux sont considérés par nombre de variables croissant et le dernier sous-réseau correspond au WCN initial complet. Une recherche de type séparation et évaluation en profondeur d'abord (DFBB) est effectuée sur chacun de ces sous-problèmes pour déterminer l'instanciation (partielle) des variables optimale dans ce sous-réseau. Les instanciations optimales (et leurs coûts) de sous-réseaux sont ensuite exploitées pour améliorer les recherches sur les sous-réseaux suivants. Ils peuvent offrir une borne inférieure de meilleure qualité au niveau local (pour un sous réseau en particulier) et au niveau global (pour le réseau complet). Les résultats peuvent être observés dans le tableau 4.22(b). Lorsqu'une méthode n'a pas trouvé de borne dans le temps imparti, le symbole "-" est indiqué dans la colonne correspondante. Nous pouvons voir que la méthode RDS parvient à trouver l'optimum pour 59% des instances de la série $spot5$ et dans un temps bien plus que faible que le temps imparti, de ce fait elle surclasse clairement l'approche CMR. Cependant, nous pouvons constater que pour les 41% d'instances restantes (qui correspondent notamment aux instances de $spot5$ les plus grandes d'un point du vue du nombre de contraintes), elle ne parvient pas à trouver une borne globale au problème dans le temps imparti contrairement à l'approche CMR.

4.8 Comparaison avec la résolution des WCSP par relaxations successives

Comme nous le disions en début de chapitre, parmi les approches de l'état de l'art visant à répondre à notre problématique figure une solution proposée simultanément et de manière indépendante par [Delisle et Bacchus, 2013]. Le principe de cette méthode est relativement proche de notre contribution, dans le sens où elle se base sur la résolution d'un WCN à partir de résolutions successives de CN. Nous proposons alors dans ce paragraphe un petit comparatif avec notre approche. La figure 4.25 illustre le fonctionnement de la méthode proposée par [Delisle et Bacchus, 2013] et permet de mettre en évidence les différences majeures (indiquées par des croix dans la figure) avec notre méthode.

Ainsi donc, le WCN à résoudre va être transformé en une succession de CN représentés par des *relaxations* (équivalentes à nos *fronts*) qui vont pour chaque contrainte souple du WCN indiquer les tuples autorisés et les tuples interdits au travers de leurs coûts (équivalents à nos *strates* autorisées ou interdites). Le poids d'une relaxation correspond à la somme des différents poids sélectionnés pour chaque contrainte dans cette relaxation. Dès le début du processus, on peut constater une première différence majeure avec notre approche : les auteurs de [Delisle et Bacchus, 2013] appliquent la cohérence d'arc

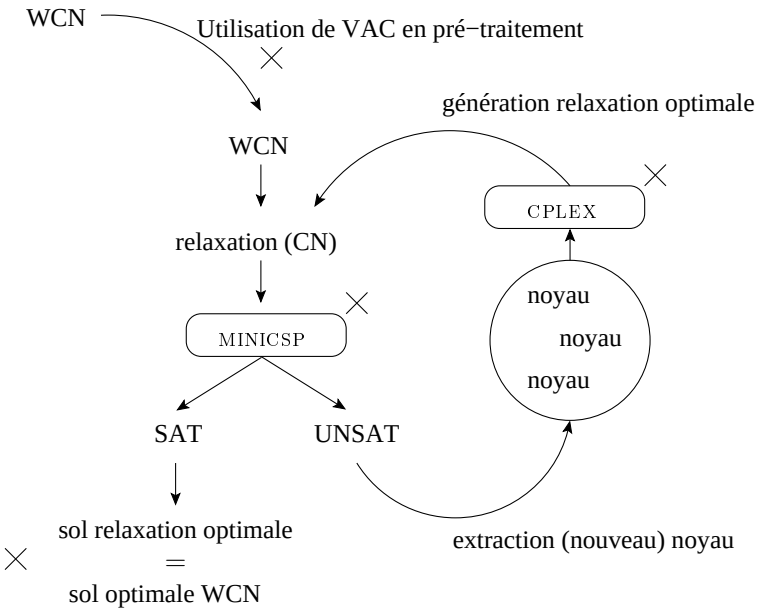


FIG. 4.25 – Principe général de la résolution de WCSP par des relaxations successives.

virtuelle VAC (section 2.3.3) sur le WCN d'origine. Nous pouvons alors penser que cette action permet de filtrer les domaines des variables et supprimer des relaxations avant même le début du traitement, ce qui n'est pas prévu dans notre approche. Une deuxième différence majeure avec notre approche apparaît ensuite. Alors que dans notre approche le CN correspondant à un front est résolu par un solveur CSP classique, le CN correspondant à une relaxation va être résolu par le solveur par apprentissage de clauses[20] MINICSP. Une variable dite "bloquante" est ainsi créée pour chacun des coûts distincts de toutes les contraintes souples, et pour chacune des clauses associées aux tuples souples dans MINICSP est ajoutée la variable correspondante selon son coût. Les variables dites "bloquantes" sont utilisées comme des hypothèses dans MINICSP. Si le CN est insatisfaisable, alors un noyau insatisfaisable (pas forcément minimal) sous la forme d'un sous-ensemble des hypothèses responsables de l'insatisfaisabilité est retourné et il est ajouté à l'ensemble des noyaux déjà rencontrés depuis le début du processus. Il s'agit ensuite de trouver une nouvelle relaxation optimale (dont le coût, c'est à dire la somme des poids des contraintes sélectionnées, est minimal) qui satisfait tous les noyaux rencontrés précédemment. Trouver cette relaxation optimale consiste à résoudre un problème de programmation en nombres entiers mixtes (MIP pour Mixed Integer Programming), dans lequel les variables bloquantes citées précédemment correspondent à des variables binaires (0 ou 1), et dont l'objectif est de minimiser le coût de cette relaxation tout en respectant les contraintes suivantes : les variables bloquantes choisies pour chaque contrainte doivent être de coût minimal et chaque noyau extrait précédemment durant le processus doit être satisfait par cette relaxation optimale. Ce problème est alors résolu par CPLEX. Une différence majeure apparaît également ici : dans notre approche la plus efficace, à savoir celle basée sur une recherche en profondeur d'abord (DFS), nous extrayons à chaque fois le front en haut de la pile, hors ici il y a un travail plus profond en terme de recherche de l'optimalité qui est réalisé au travers de la recherche d'une relaxation optimale satisfaisant tous les noyaux insatisfaisables rencontrés. Le processus s'arrête dès qu'une relaxation optimale trouvée est satisfaisable : une solution de cette première relaxation satisfaisable correspond à une solution optimale du WCSP considéré. C'est une autre différence majeure avec notre approche dans laquelle il nous faut vider la pile et considérer tous les fronts pour assurer la complétude et ne pas

[20] Dans le cadre *SAT*, une clause est une proposition disjonctive de littéraux

simplement considérer le premier front satisfaisable trouvé. Enfin, une dernière proposition d'amélioration a été proposée dans cette approche pour diminuer l'effort coûteux que représente la recherche d'une relaxation optimale satisfaisant l'ensemble des noyaux déjà extraits. Il s'agit en fait de faire en sorte que le nombre de variables bloquantes retourné par MINICSP soit réduit. Il s'agit d'une dernière différence, à savoir que ici qu'il ne s'agit pas de noyaux insatisfaisables minimaux (en termes de nombre de variables ou de contraintes) qui sont cherchés contrairement à nous.

Les expérimentations dans [Delisle et Bacchus, 2013] ont été effectuées sur les séries de problèmes *linkage* (*pedigree*) et *spot5*. Concernant les bornes obtenues pour les instances *linkage*, notre approche est clairement dépassée dans le sens où elle ne trouve jamais la borne optimale alors que l'approche proposée par [Delisle et Bacchus, 2013] parvient pour la grande majorité de ces instances à trouver la borne optimale dans un temps bien plus faible que nos 600 secondes de temps imparti. Malheureusement, il est difficile de comparer nos résultats expérimentaux et ceux obtenus par l'approche [Delisle et Bacchus, 2013] pour la série *spot5*. Il y est montré que l'optimum a été trouvé dans un temps inférieur à 600 secondes pour uniquement 3 instances. Cependant, il s'agit d'instances relativement petites, et plus précisément avec un nombre de contraintes inférieur à 2300. Pour les autres instances, il est juste indiqué que l'optimum n'a pas été trouvé. Sachant que notre approche se révèle vraiment efficace pour les instances présentant un grand nombre de contraintes, les quelques résultats proposés dans [Delisle et Bacchus, 2013] ne sont pas suffisants pour se faire une idée de comparaison. Il aurait été intéressant de tester le solveur proposé par [Delisle et Bacchus, 2013] depuis la même plateforme que celle utilisée pour nos tests, malheureusement les auteurs nous ont précisé que celui-ci n'est pas portable à l'heure actuelle et donc irrécupérable pour tests de comparaison. Pour finir, il est intéressant de noter que [Delisle et Bacchus, 2013] n'a pas obtenu non plus de bons résultats pour les séries d'instances *celar* comparés à l'approche proposée par $ToulBar2$ pour les mêmes raisons que celles que nous avons avancées, à savoir l'exploitation de techniques spécifiques par $ToulBar2$. On peut donc vraiment se demander si ces approches (notre approche et celle de [Delisle et Bacchus, 2013]) qui sont sensiblement basées sur le même principe, à savoir la résolution d'un WCSP par la résolution successive de CSP obtenus à partir de ce WCSP, s'avèrent adaptées à ce type de problème.

Conclusion générale

Dans le cadre de cette thèse, nous nous sommes intéressés à l'intégration de techniques CSP pour la résolution d'instances WCSP. Comme nous l'avons rappelé dans ce manuscrit, de nombreuses approches ont été proposées pour traiter ce problème d'optimisation. Les méthodes les plus efficaces utilisent des cohérences locales souples sophistiquées, établies par transferts de coût, qui permettent d'accélérer la résolution en réduisant l'espace de recherche via la suppression de valeurs et le calcul de bornes inférieures utiles en pratique. Cependant, l'utilisation de ces méthodes pose un problème lorsque l'arité des contraintes augmente de manière significative. Conscients de l'efficacité des techniques du cadre CSP, nous avons proposé deux approches basées sur ces techniques pour résoudre le problème WCSP. Nos résultats expérimentaux montrent que notre première approche est compétitive avec l'état de l'art, tandis que la deuxième représente une approche alternative aux méthodes de résolution habituelles du problème WCSP.

L'algorithme de filtrage pour les contraintes tables souples de grande arité permet d'établir la cohérence d'arc souple généralisée GAC* sur des contraintes tables souples de grande arité. Les opérations de transferts de coûts permettant d'établir cette cohérence nécessitent de connaître pour chaque valeur le coût minimal des tuples contenant cette valeur dans chaque contrainte. Avec un schéma de filtrage classique, cela nécessite de parcourir tous les tuples valides et leur nombre peut être exponentiel en fonction de l'arité de la contrainte. Pour pallier cette difficulté, nous combinons la réduction tabulaire simple STR issue du cadre CSP et le principe du transfert de coûts. À chaque étape d'une recherche, les tuples explicites de la table sont parcourus et les tuples reconnus invalides sont supprimés, ce qui permet alors d'identifier les valeurs qui ne sont plus GAC*-cohérentes. Bien sûr, lors de ce parcours de tuples, les coûts minimaux trouvés dans les tuples explicites sont enregistrés. Ensuite, nous avons proposé une méthode polynomiale pour déterminer les coûts minimaux parmi les tuples implicites, c'est à dire les tuples qui ne sont pas présents dans la table et qui sont représentés par un coût par défaut. Cette méthode est spécifique selon la valeur du coût par défaut : égal à 0, égal au coût interdit k ou intermédiaire, c'est à dire entre 0 et k exclus. Une fois les coûts minimaux calculés pour chaque valeur, ils sont exploités pour réaliser les transferts de coûts. À noter que cette technique pour calculer les coûts minimaux associées aux valeurs a également été exploitée dans l'approche PFC-MPRDAC.

La résolution WCSP par l'extraction de noyaux insatisfaisables minimaux est une approche alternative aux approches de résolution les plus efficaces proposées pour le problème WCSP. Cette approche est basée uniquement sur des résolutions successives d'instances CSP obtenues depuis une instance WCSP d'origine. Aucune des techniques proposées dans le cadre WCSP (transferts de coûts) n'est

utilisée ici. À partir d'une instance CSP obtenue en durcissant au maximum l'instance WCSP d'origine, une résolution est effectuée. Si l'instance CSP est insatisfaisable, cela signifie que des contraintes doivent être relâchées. Pour déterminer ces contraintes, un noyau insatisfaisable minimal est extrait de l'instance CSP et englobe les contraintes à relâcher. Dès qu'une instance CSP est satisfaisable, la somme des coûts des relâchements de contraintes à l'origine de cette instance CSP représente alors le coût d'une solution pour l'instance WCSP à résoudre. Selon l'approche considérée, à savoir incomplète ou complète, ce coût correspond soit à une borne supérieure du coût d'une solution optimale, soit au coût d'une solution optimale.

Perspectives de travaux futurs Dans le chapitre 3, nous avons proposé un algorithme permettant d'établir la cohérence GAC* sur des contraintes tables souples de grande arité. Une perspective directe pourrait être de s'intéresser à établir d'autres cohérences locales souples, comme par exemple FDAC* dans un premier temps. Basée également sur les opérations de transferts de coûts, nous pourrions voir si, grâce au calcul de manière efficace de coûts minimaux rendu possible par notre algorithme, cette cohérence locale souple ne peut pas être établie plus efficacement sur les contraintes tables souples de grande arité. Dans le chapitre 4, l'un des inconvénients de notre approche complète est le nombre de fronts traités qui nécessitent chacun, potentiellement, une résolution et l'extraction d'un noyau insatisfaisable. L'enregistrement de patterns de noyaux insatisfaisables déjà rencontrés a certes permis d'améliorer nos résultats, mais cela n'est pas suffisant. Une perspective intéressante serait de continuer à réfléchir à des propriétés sur les fronts, comme par exemple des relations de dominance, pour déceler à moindre coût les fronts à ne pas considérer et ainsi réduire encore les efforts requis lors de l'exploration des fronts. Enfin, de manière générale, une perspective à ces deux travaux est bien sûr de continuer à explorer les techniques du cadre CSP et déterminer celles qui peuvent être adaptées à la résolution d'instances WCSP.

Bibliographie

[Aldous et Vazirani, 1994] ALDOUS, D. et VAZIRANI, U. (1994). “go with the winners” algorithms. *In Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 492–501. IEEE.

[Allouche *et al.*, 2012] ALLOUCHE, D., BESSIERE, C., BOIZUMAULT, P., de GIVRY, S., GUTIERREZ, P., LOUDNI, S., MÉTIVIER, J.-P. et SCHIEX, T. (2012). Decomposing global cost functions. *In Proceedings of AAAI'12*.

[Ansótegui *et al.*, 2010] ANSÓTEGUI, C., BONET, M. L. et LEVY, J. (2010). A new algorithm for weighted partial maxsat. *In AAAI*.

[Beacham *et al.*, 2001] BEACHAM, A., CHEN, X., SILLITO, J. et van BEEK, P. (2001). Constraint programming lessons learned from crossword puzzles. *In Advances in Artificial Intelligence*, pages 78–87. Springer.

[Bensana *et al.*, 1999] BENSANA, E., LEMAITRE, M. et VERFAILLIE, G. (1999). Earth observation satellite management. *Constraints*, 4(3):293–299.

[Bessiere, 1991] BESSIERE, C. (1991). Arc-consistency in dynamic constraint satisfaction problems. *In AAAI*, volume 91, pages 221–226.

[Bessiere, 1994] BESSIERE, C. (1994). Arc-consistency and arc-consistency again. *Artificial intelligence*, 65(1):179–190.

[Bessiere, 2006] BESSIERE, C. (2006). Constraint propagation. *Foundations of Artificial Intelligence*, 2:29–83.

[Bessière *et al.*, 1999] BESSIÈRE, C., FREUDER, E. C. et RÉGIN, J.-C. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107(1):125–148.

[Bessiere et Régin, 1996] BESSIERE, C. et RÉGIN, J.-C. (1996). Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems. *In Principles and Practice of Constraint Programming—CP96*, pages 61–75. Springer.

[Bessière et Régin, 2001] BESSIÈRE, C. et RÉGIN, J.-C. (2001). Refining the basic constraint propagation algorithm. *In IJCAI*, volume 1, pages 309–315.

[Bessière *et al.*, 2005] BESSIÈRE, C., RÉGIN, J.-C., YAP, R. H. et ZHANG, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185.

[Bistarelli *et al.*, 1996] BISTARELLI, S., FAXGIER, H., MONTANARI, U., ROSSI, F., SCHIEX, T. et VERFAILLIE, G. (1996). Semiring-based csps and valued csps : Basic properties and comparison. *In Over-constrained systems*, pages 111–150. Springer.

[Bistarelli *et al.*, 1995] BISTARELLI, S., MONTANARI, U. et ROSSI, F. (1995). Constraint solving over semirings. *In IJCAI (1)*, pages 624–630. Citeseer.

[Black, 1999] BLACK, P. E. (1999). Algorithms and theory of computation handbook.

[Boussemart *et al.*, 2004] BOUSSEMART, F., HEMERY, F., LECOUTRE, C. et SAIS, L. (2004). Boosting systematic search by weighting constraints. *In ECAI*, volume 16, page 146.

[Brélaz, 1979] BRÉLAZ, D. (1979). New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256.

[Cabon *et al.*, 1999] CABON, B., DE GIVRY, S., LOBJOIS, L., SCHIEX, T. et WARNERS, J. P. (1999). Radio link frequency assignment. *Constraints*, 4(1):79–89.

[Cohen *et al.*, 2006] COHEN, D. A., COOPER, M. C., JEAVONS, P. G. et KROKHIN, A. A. (2006). The complexity of soft constraint satisfaction. *Artificial Intelligence*, 170(11):983–1016.

[Cook, 1971] COOK, S. A. (1971). The complexity of theorem-proving procedures. *In Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM.

[Cooper et Schiex, 2004] COOPER, M. et SCHIEX, T. (2004). Arc consistency for soft constraints. *Artificial Intelligence*, 154(1):199–227.

[Cooper *et al.*, 2008] COOPER, M. C., de GIVRY, S., SÁNCHEZ, M., SCHIEX, T. et ZYTNICKI, M. (2008). Virtual arc consistency for weighted csp. *In AAAI*, volume 8, pages 253–258.

[Cooper *et al.*, 2010] COOPER, M. C., de GIVRY, S., SANCHEZ, M., SCHIEX, T., ZYTNICKI, M. et WERNER, T. (2010). Soft arc consistency revisited. *Artificial Intelligence*, 174(7):449–478.

[Cooper *et al.*, 2007] COOPER, M. C., de GIVRY, S. et SCHIEX, T. (2007). Optimal soft arc consistency. *In IJCAI*, volume 7, pages 68–73.

[Dantzig, 1982] DANTZIG, G. B. (1982). Reminiscences about the origins of linear programming. *Operations Research Letters*, 1(2):43–48.

[De Givry *et al.*, 2005] DE GIVRY, S., HERAS, F., ZYTNICKI, M. et LARROSA, J. (2005). Existential arc consistency : Getting closer to full arc consistency in weighted csps. *In IJCAI*, volume 5, pages 84–89.

[De Givry *et al.*, 2003] DE GIVRY, S., LARROSA, J., MESEGUER, P. et SCHIEX, T. (2003). Solving max-sat as weighted csp. *In Principles and Practice of Constraint Programming–CP 2003*, pages 363–376. Springer.

[De Givry *et al.*, 2006] DE GIVRY, S., SCHIEX, T. et VERFAILLIE, G. (2006). Exploiting tree decomposition and soft local consistency in weighted csp. *In AAAI*, volume 6, pages 1–6.

[De Saint-Cyr *et al.*, 1994] DE SAINT-CYR, F. D., LANG, J. et SCHIEX, T. (1994). Penalty logic and its link with dempster-shafer theory. *In Proceedings of the Tenth international conference on Uncertainty in artificial intelligence*, pages 204–211. Morgan Kaufmann Publishers Inc.

[de Siqueira et Puget, 1988] de SIQUEIRA, J. et PUGET, J. (1988). Explanation-based generalisation of failures. *In Proceedings of ECAI'88*, pages 339–344.

[Dechter, 1990] DECHTER, R. (1990). Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312.

[Dechter, 2003] DECHTER, R. (2003). *Constraint processing*. Morgan Kaufmann.

[Dechter et Frost, 2002] DECHTER, R. et FROST, D. (2002). Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136(2):147–188.

[Dechter et Meiri, 1989] DECHTER, R. et MEIRI, I. (1989). *Experimental evaluation of preprocessing techniques in constraint satisfaction problems*. UCLA, Computer Science Department.

[Dechter et Pearl, 1989] DECHTER, R. et PEARL, J. (1989). Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366.

[Dehani, 2014] DEHANI, D.-E. (2014). La substituabilité et la cohérence de tuples pour les réseaux de contraintes pondérées. pages 62–70.

[Delisle et Bacchus, 2013] DELISLE, E. et BACCHUS, F. (2013). Solving weighted csps by successive relaxations. *In Principles and Practice of Constraint Programming*, pages 273–281. Springer.

[Dimitriou et Impagliazzo, 1996] DIMITRIOU, T. et IMPAGLIAZZO, R. (1996). Towards an analysis of local optimization algorithms. *In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 304–313. ACM.

[Dorigo *et al.*, 1996] DORIGO, M., MANIEZZO, V. et COLORNI, A. (1996). Ant system : optimization by a colony of cooperating agents. *Systems, Man, and Cybernetics, Part B : Cybernetics, IEEE Transactions on*, 26(1):29–41.

[Fargier et Lang, 1993] FARGIER, H. et LANG, J. (1993). Uncertainty in constraint satisfaction problems : a probabilistic approach. *In Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 97–104. Springer.

[Fargier *et al.*, 1993] FARGIER, H., LANG, J. et SCHIEX, T. (1993). Selecting preferred solutions in fuzzy constraint satisfaction problems. *In Proc. 1st European Congress on Fuzzy and Intelligent Technologies (EUFIT)*.

[Favier *et al.*, 2011] FAVIER, A., de GIVRY, S., LEGARRA, A. et SCHIEX, T. (2011). Pairwise decomposition for combinatorial optimization in graphical models. *In Proceedings of IJCAI'11*, pages 2126–2132.

[Fontaine *et al.*, 2013] FONTAINE, M., LOUDNI, S. et BOIZUMAULT, P. (2013). Exploiting tree decomposition for guiding neighborhoods exploration for vns. *RAIRO-Operations Research*, 47(02):91–123.

[Freuder, 1978] FREUDER, E. C. (1978). Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966.

[Freuder et Wallace, 1992] FREUDER, E. C. et WALLACE, R. J. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58(1):21–70.

[Frost et Dechter, 1994] FROST, D. et DECHTER, R. (1994). Dead-end driven learning. *In AAAI*, volume 94, pages 294–300.

[Frost et Dechter, 1995] FROST, D. et DECHTER, R. (1995). Look-ahead value ordering for constraint satisfaction problems. *In IJCAI (1)*, pages 572–578. Citeseer.

[Fu et Malik, 2006] FU, Z. et MALIK, S. (2006). On solving the partial max-sat problem. *In Theory and Applications of Satisfiability Testing-SAT 2006*, pages 252–265. Springer.

[Ginsberg, 1993] GINSBERG, M. L. (1993). Dynamic backtracking. *J. Artif. Intell. Res. (JAIR)*, 1:25–46.

[Ginsberg *et al.*, 1990] GINSBERG, M. L., FRANK, M., HALPIN, M. P. et TORRANCE, M. C. (1990). Search lessons learned from crossword puzzles. *In AAAI*, pages 210–215. Citeseer.

[Glover, 1989] GLOVER, F. (1989). Tabu search—part i. *ORSA Journal on computing*, 1(3):190–206.

[Goldberg, 1989] GOLDBERG, D. (1989). Genetic algorithms in optimization, search and machine learning. *Addison Wesley, New York. Eiben AE, Smith JE (2003) Introduction to Evolutionary Computing. Springer. Jacq J, Roux C (1995) Registration of non-segmented images using a genetic algorithm. Lecture notes in computer science*, 905:205–211.

[Gregoire *et al.*, 2013] GREGOIRE, E., LAGNIEZ, J.-M. et MAZURE, B. (2013). Questioning the importance of wcore-like minimization steps in muc-finding algorithms. *In Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*, pages 923–930. IEEE.

[Haralick et Elliott, 1980] HARALICK, R. M. et ELLIOTT, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial intelligence*, 14(3):263–313.

[Harvey et Ginsberg, 1995] HARVEY, W. D. et GINSBERG, M. L. (1995). Limited discrepancy search. *In IJCAI (1)*, pages 607–615.

[Hemery *et al.*, 2006] HEMERY, F., LECOUTRE, C., SAIS, L., BOUSSEMART, F. *et al.* (2006). Extracting mucs from constraint networks. *In ECAI*, volume 6, pages 113–117.

[Heras et Larrosa, 2006] HERAS, F. et LARROSA, J. (2006). Intelligent variable orderings and re-orderings in dac-based solvers for wcsp. *Journal of heuristics*, 12(4-5):287–306.

[Holland, 1975] HOLLAND, J. H. (1975). *Adaptation in natural and artificial systems : An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

[Hwang et Mitchell, 2005] HWANG, J. et MITCHELL, D. G. (2005). 2-way vs. d-way branching for csp. *In Principles and Practice of Constraint Programming-CP 2005*, pages 343–357. Springer.

[Jégou, 1993] JÉGOU, P. (1993). Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. *In AAAI*, volume 93, pages 731–736.

[Jégou et Terrioux, 2004] JÉGOU, P. et TERRIOUX, C. (2004). Decomposition and good recording for solving max-csps. *In Proceedings of ECAI'2004*, pages 196–200.

[Jeroslow et Wang, 1990] JEROSLOW, R. G. et WANG, J. (1990). Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187.

[Junker, 2004] JUNKER, U. (2004). Quickxplain : preferred explanations and relaxations for over-constrained problems. *In AAAI*, volume 4, pages 167–172.

[Jussien *et al.*, 2002] JUSSIEN, N., LHOMME, O. et ILOG, S. (2002). Unifying search algorithms for csp. Rapport technique, Citeseer.

[Karmarkar, 1984] KARMARKAR, N. (1984). A new polynomial-time algorithm for linear programming. *In Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM.

[Kirkpatrick *et al.*, 1983] KIRKPATRICK, S., JR., D. G. et VECCHI, M. P. (1983). Optimization by simmulated annealing. *science*, 220(4598):671–680.

[Krzysztof, 2003] KRZYSZTOF, R. (2003). *Apt. Principles of constraint programming*. Cambridge University Press Cambridge.

[Larrosa, 2002] LARROSA, J. (2002). Node and arc consistency in weighted csp. *In AAAI/IAAI*, pages 48–53.

[Larrosa et Meseguer, 1996] LARROSA, J. et MESEGUER, P. (1996). Exploiting the use of dac in max-csp. *In Principles and Practice of Constraint Programming—CP96*, pages 308–322. Springer.

[Larrosa et Meseguer, 1999] LARROSA, J. et MESEGUER, P. (1999). Partition-based lower bound for max-csp. *In Principles and Practice of Constraint Programming–CP'99*, pages 303–315. Springer.

[Larrosa *et al.*, 1999] LARROSA, J., MESEGUER, P. et SCHIEX, T. (1999). Maintaining reversible dac for max-csp. *Artificial Intelligence*, 107(1):149–163.

[Larrosa et Schiex, 2003] LARROSA, J. et SCHIEX, T. (2003). In the quest of the best form of local consistency for weighted csp. *In IJCAI*, volume 3, pages 239–244. Citeseer.

[Larrosa et Schiex, 2004] LARROSA, J. et SCHIEX, T. (2004). Solving weighted csp by maintaining arc consistency. *Artificial Intelligence*, 159(1):1–26.

[Lecoutre, 2009] LECOUTRE, C. (2009). Constraint networks : Techniques and algorithms.

[Lecoutre, 2011] LECOUTRE, C. (2011). Str2 : optimized simple tabular reduction for table constraints. *Constraints*, 16(4):341–371.

[Lecoutre *et al.*, 2003] LECOUTRE, C., BOUSSEMART, F. et HEMERY, F. (2003). Exploiting multidirectionality in coarse-grained arc consistency algorithms. *In Principles and Practice of Constraint Programming–CP 2003*, pages 480–494. Springer.

[Lecoutre et Hemery, 2007] LECOUTRE, C. et HEMERY, F. (2007). A study of residual supports in arc consistency. *In In Proceedings of IJCAI'07*. Citeseer.

[Lecoutre *et al.*, 2012] LECOUTRE, C., PARIS, N., ROUSSEL, O. et TABARY, S. (2012). Propagating soft table constraints. *In Principles and Practice of Constraint Programming*, pages 390–405. Springer.

[Lecoutre *et al.*, 2013] LECOUTRE, C., PARIS, N., ROUSSEL, O. et TABARY, S. (2013). Solving wcsp by extraction of minimal unsatisfiable cores. *In Tools with Artificial Intelligence, 2013. ICTAI'13. 25th International Conference on*, pages 915–922. IEEE.

[Lecoutre *et al.*, 2006] LECOUTRE, C., SAIS, L., TABARY, S. et VIDAL, V. (2006). Last conflict based reasoning. *In Proceedings of ECAI'2006*, pages 133–137.

[Lecoutre *et al.*, 2007] LECOUTRE, C., SAIS, L., TABARY, S., VIDAL, V. *et al.* (2007). Nogood recording from restarts. *In IJCAI*, volume 7, pages 131–136.

[Lecoutre et Szymanek, 2006] LECOUTRE, C. et SZYMANEK, R. (2006). Generalized arc consistency for positive table constraints. *In Principles and Practice of Constraint Programming-CP 2006*, pages 284–298. Springer.

[Lee et Leung, 2009] LEE, J. et LEUNG, K. (2009). Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. *In Proceedings of IJCAI'09*, pages 559–565.

[Lee et Leung, 2010] LEE, J. et LEUNG, K. (2010). A stronger consistency for soft global constraints in weighted constraint satisfaction. *In Proceedings of AAAI'10*, pages 121–127.

[Levasseur, 2008] LEVASSEUR, N. (2008). *Heuristiques de recherche pour la résolution des WCSP*. Thèse de doctorat, Caen.

[Levasseur *et al.*, 2007] LEVASSEUR, N., BOIZUMAULT, P. et LOUDNI, S. (2007). A value ordering heuristic for weighted csp. *In Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on*, volume 1, pages 259–262. IEEE.

[Loudin et Boizumault, 2001] LOUDIN, S. et BOIZUMAULT, P. (2001). Vns/lds+ cp : A hybrid method for constraint optimization in anytime contexts. *In Proceedings of MIC*, pages 761–765. Citeseer.

[Loudni et Boizumault, 2008] LOUDNI, S. et BOIZUMAULT, P. (2008). Combining vns with constraint programming for solving anytime optimization problems. *European Journal of Operational Research*, 191(3):705 – 735.

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

[Marques-Sila et Planes, 2011] MARQUES-SILA, J. et PLANES, J. (2011). Algorithms for maximum satisfiability using unsatisfiable cores. *In Advanced Techniques in Logic Synthesis, Optimizations and Applications*, pages 171–182. Springer.

[Minton *et al.*, 1992] MINTON, S., JOHNSTON, M. D., PHILIPS, A. B. et LAIRD, P. (1992). Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58(1):161–205.

[Mladenović et Hansen, 1997] MLADENOVIĆ, N. et HANSEN, P. (1997). Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100.

[Mohr et Henderson, 1986] MOHR, R. et HENDERSON, T. C. (1986). Arc and path consistency revisited. *Artificial intelligence*, 28(2):225–233.

[Neveu et Trombettoni, 2003] NEVEU, B. et TROMBETTONI, G. (2003). Incop : An open library for incomplete combinatorial optimization. *In Principles and Practice of Constraint Programming–CP 2003*, pages 909–913. Springer.

[Neveu et Trombettoni, 2004] NEVEU, B. et TROMBETTONI, G. (2004). Hybridation de gww avec de la recherche locale. *9èmes Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets, JNPC*, 3.

[Nguyen *et al.*, 2013] NGUYEN, H., SCHIEX, T. et BESSIERE, C. (2013). Dynamic virtual arc consistency. *In Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 98–103. ACM.

[Papadimitriou, 1994] PAPADIMITRIOU, C. H. (1994). *Computational complexity*. Addison-Wesley.

[Pinkas, 1991] PINKAS, G. (1991). *Propositional non-monotonic reasoning and inconsistency in symmetric neural networks*. Washington University, Department of Computer Science.

[Pisinger et Ropke, 2010] PISINGER, D. et ROPKE, S. (2010). Large neighborhood search. *In Handbook of metaheuristics*, pages 399–419. Springer.

[Prosser, 1993] PROSSER, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational intelligence*, 9(3):268–299.

[Puget, 1998] PUGET, J.-F. (1998). A fast algorithm for the bound consistency of alldiff constraints. *In AAAI/IAAI*, pages 359–366.

[Refalo, 2004] REFALO, P. (2004). Impact-based search strategies for constraint programming. *In Principles and Practice of Constraint Programming–CP 2004*, pages 557–571. Springer.

[Régin, 1994] RÉGIN, J.-C. (1994). A filtering algorithm for constraints of difference in csps. *In AAAI*, volume 94, pages 362–367.

[Robertson et Seymour, 1986] ROBERTSON, N. et SEYMOUR, P. D. (1986). Graph minors. ii. algorithmic aspects of tree-width. *Journal of algorithms*, 7(3):309–322.

[Rossi *et al.*, 2006] ROSSI, F., VAN BEEK, P. et WALSH, T. (2006). *Handbook of constraint programming*. Access Online via Elsevier.

[Roussel et Lecoutre, 2009] ROUSSEL, O. et LECOUTRE, C. (2009). Xml representation of constraint networks : Format xcsp 2.1. *arXiv preprint arXiv :0902.2362*.

[Sabin et Freuder, 1994] SABIN, D. et FREUDER, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. *In Principles and Practice of Constraint Programming*, pages 10–20. Springer.

[Sánchez *et al.*, 2009] SÁNCHEZ, M., ALLOUCHE, D., de GIVRY, S. et SCHIEX, T. (2009). Russian doll search with tree decomposition. *In Proceedings of IJCAI'09*, pages 603–608.

[Sanchez *et al.*, 2008] SANCHEZ, M., de GIVRY, S. et SCHIEX, T. (2008). Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1-2):130–154.

[Schiex, 1992] SCHIEX, T. (1992). Possibilistic constraint satisfaction problems or how to handle soft constraints ? *In Proceedings of the Eighth international conference on Uncertainty in artificial intelligence*, pages 268–275. Morgan Kaufmann Publishers Inc.

[Schiex, 2000] SCHIEX, T. (2000). Arc consistency for soft constraints. *In Principles and Practice of Constraint Programming–CP 2000*, pages 411–425. Springer.

[Schiex *et al.*, 1995] SCHIEX, T., FARGIER, H., VERFAILLIE, G. *et al.* (1995). Valued constraint satisfaction problems : Hard and easy problems. *IJCAI (1)*, 95:631–639.

[Shaw, 1998] SHAW, P. (1998). Using constraint programming and local search methods to solve vehicle routing problems. *In Principles and Practice of Constraint Programming—CP98*, pages 417–431. Springer.

[Smith, 1996] SMITH, B. M. (1996). Succeed-first or fail-first : A case study in variable and value ordering.

[Teghem, 2012] TEGHEM, J. (2012). La recherche opérationnelle tome 1 : Les méthodes d'optimisation.

[Turing, 1936] TURING, A. M. (1936). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265.

[Ullmann, 2007] ULLMANN, J. R. (2007). Partition search for non-binary constraint satisfaction. *Information Sciences*, 177(18):3639–3678.

[van Dongen, 2002] van DONGEN, M. R. (2002). Ac-3d an efficient arc-consistency algorithm with a low space-complexity. *In Principles and Practice of Constraint Programming-CP 2002*, pages 755–760.

[Verfaillie *et al.*, 1996] VERFAILLIE, G., LEMAÎTRE, M. et SCHIEX, T. (1996). Russian doll search for solving constraint optimization problems. *In AAAI/IAAI, Vol. 1*, pages 181–187. Citeseer.

[Wieringa, 2012] WIERINGA, S. (2012). Understanding, improving and parallelizing mus finding using model rotation. *In Principles and Practice of Constraint Programming*, pages 672–687. Springer.

[Wilson et Nash, 2003] WILSON, R. et NASH, C. (2003). Four colours suffice : How the map problem was solved. *The Mathematical Intelligencer*, 25(4):80–83.

[Xu *et al.*, 2007] XU, K., BOUSSEMART, F., HEMERY, F. et LECOUTRE, C. (2007). Random constraint satisfaction : easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9):514–534.

[Zhang et Yap, 2001] ZHANG, Y. et YAP, R. H. (2001). Making ac-3 an optimal algorithm. *In IJCAI*, volume 1, pages 316–321.

Résumé

Cette thèse se situe dans le contexte de la programmation par contraintes (CP). Plus précisément, nous nous sommes intéressés au problème de satisfaction de contraintes pondérées (WCSP). De nombreuses approches ont été proposées pour traiter ce problème d'optimisation. Les méthodes les plus efficaces utilisent des cohérences locales souples sophistiquées comme par exemple la cohérence d'arc directionnelle complète FDAC*, la cohérence d'arc directionnelle existentielle EDAC*, etc. Établies grâce à des opérations de transferts de coût préservant l'équivalence des réseaux, l'utilisation de ces cohérences permet généralement d'accélérer la résolution en réduisant l'espace de recherche via la suppression de valeurs et le calcul de bornes inférieures utiles en pratique. Cependant, l'utilisation de ces méthodes pose un problème lorsque l'arité des contraintes augmente de manière significative. L'efficacité des techniques du cadre du problème de satisfaction de contraintes (CSP) étant avérée, nous pensons que l'intégration de techniques CSP peut être très utile à la résolution d'instances WCSP. Dans cette thèse, nous proposons tout d'abord un algorithme de filtrage établissant la cohérence d'arc souple généralisée GAC* sur des contraintes tables souples de grande arité. Cette approche combine la technique de réduction tabulaire simple (STR), issue du cadre CSP, et le principe de transfert de coûts. Notre approche qui est polynomiale calcule efficacement pour chaque valeur les coûts minimaux dans les tuples explicites et implicites des contraintes tables souples. Ces coûts minimaux sont ensuite utilisés pour transférer les coûts afin d'établir GAC*. Dans un second temps, nous proposons une approche alternative aux méthodes de résolution habituelles du problème WCSP. Elle consiste à résoudre une instance WCSP en résolvant une séquence d'instances CSP classiques obtenues depuis cette instance WCSP. À partir d'une instance CSP dans laquelle toutes les contraintes de l'instance WCSP d'origine sont durcies au maximum, les instances CSP suivantes correspondent à un relâchement progressif de contraintes de l'instance WCSP déterminées par l'extraction de noyaux insatisfaisables minimaux (MUC) depuis les réseaux insatisfaisables de la séquence. Nos résultats expérimentaux montrent que notre première approche est compétitive avec l'état de l'art, tandis que la deuxième représente une approche alternative aux méthodes de résolution habituelles d'instances WCSP.

Mots-clés: intelligence artificielle, programmation par contraintes, problème de satisfaction de contraintes pondérées, problème de satisfaction de contraintes, réduction tabulaire simple, noyau insatisfaisable minimal

Abstract

This thesis is in the context of constraint programming (CP). Specifically, we are interested in the Weighted Constraint Satisfaction Problem (WCSP). Many approaches have been proposed to handle this optimization problem. The most effective methods use sophisticated soft local consistencies such as, for example, full directional arc consistency FDAC*, existential directional arc consistency EDAC*, etc. Established by equivalence preserving transformations (cost transfer operations), the use of these consistencies generally allows both to accelerate the resolution by reducing the search space through the elimination of values and to compute lower bounds useful in practice. However, these methods reach their limits when the arity of constraints increases significantly. The techniques of the Constraint Satisfaction Problem framework (CSP) having proved efficienty, we believe that the integration of CSP techniques can be very useful for solving WCSP instances. In this thesis, we first propose a filtering algorithm to

enforce a soft version of generalized arc consistency (GAC*) on soft table constraints of large arity. This approach combines the techniques of simple tabular reduction (STR), from the CSP framework, with the techniques of cost transfer. Our approach, proved polynomial, efficiently calculates for each value the minimum cost of the explicit and implicit tuples from soft table constraints. The minimum costs are then used to transfer costs to establish GAC*. In a second step, we propose an alternative approach to the usual techniques to solve WCSP. The principle is to solve a WCSP instance by solving a sequence of classical CSP instances obtained from this WCSP instance. From a CSP instance containing all the constraints hardened to the maximum from the WCSP instance, the next CSP instances correspond to a progressive relaxation of constraints defined by extraction of minimal unsatisfiable cores (MUC) from unsatisfiable networks of the sequence. Our experimental results show that our first approach is competitive with the state-of-the-art, whereas the second one represents an alternative approach to the usual methods to solve WCSP instances.

Keywords: artificial intelligence, constraint programming, weighted constraint satisfaction problem, constraint satisfaction problem, simple tabular reduction, minimal unsatisfiable core