UNIVERSITÉ D'ARTOIS

UNIVERSITÉ D'ARTOIS

ÉCOLE DOCTORALE SPI 072
SCIENCES POUR L'INGÉNIEUR

T H È S E

pour l'obtention du titre de

Docteur en Sciences

de l'Université d'Artois
Mention : INFORMATIQUE

Présentée par

Djamel-Eddine DEHANI

La substituabilité et la cohérence de tuples pour les réseaux de contraintes pondérées

soutenue publiquement le
13 Février 2014

Composition du jury :

Rapporteurs :	Simon DE GIVRY (INRA, Toulouse)	
	Arnaud LALLOUET (Université de Caen)	
Examinateurs :	Gilles AUDEMARD (Université d'Artois)	
	Remi COLETTA (Universite Montpellier 2)	
	Frédéric KORICHE (Université d'Artois)	
	Christophe LECOUTRE (Université d'Artois)	*directeur de thèse*
	Olivier ROUSSEL (Université d'Artois)	*co-directeur de thèse*
	Vincent VIDAL (ONERA, Toulouse)	

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS (CRIL 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.univ-artois.fr

La substituabilité et la cohérence de tuples pour les réseaux de contraintes pondérées

—

Djamel Eddine Dehani

À mes parents et à ma famille.

Remerciements

Un travail tel que celui-ci n'est pas seulement l'œuvre de son signataire. Je tiens à remercier tous ceux qui m'ont aidé et soutenu, sans qu'ils aient eu forcément pleinement conscience de l'importance que cela pouvait avoir pour moi.

Mes premiers remerciements vont à l'ensemble des membres du jury pour l'intérêt qu'ils ont porté à mes travaux, je remercie :

- M. Simon de Givry, Chargé de Recherche HDR, INRA Toulouse et
- M. Arnaud Lallouet, Professeur des universités à l'université de Caen de m'avoir fait l'honneur de rapporter ce manuscrit. Leurs remarques pertinentes ont inéluctablement permis de le rendre meilleur et de m'avoir aussi chaleureusement encouragé à poursuivre ce travail.
- M. Gilles Audemard, Professeur des universités à l'université d'Artois d'avoir accepter de présider le jury de cette thèse.
- M. Rémi Coletta, Maître de conférences à l'université Montpellier 2,
- M. Frederic Koriche, Professeur des universités à l'université d'Artois et
- M. Vincent Vidal, Chargé de Recherche, ONERA Toulouse d'avoir accepté de faire partie du jury.

Je remercie tout particulièrement mes deux co-directeurs : M. Christophe Lecoutre, Professeur des universités à l'université d'Artois, et M. Olivier Roussel, Maître de conférences à l'université d'Artois, d'avoir accepter d'encadrer cette thèse.

Un grand remerciement est adressé à Mme Souhila Kaci, Professeur des universités à l'université Montpellier 2, M. Salem Benferhat, M. Jean-François Condotta et M. Lakhdar Sais, Professeurs des universités à l'université d'Artois, qui m'ont toujours encouragé, m'ont régulièrement soutenue dans ce travail. Je tiens à leur exprimer ma reconnaissance et ma profonde gratitude.

Je n'oublie pas non plus tout ce que je dois à M. Yakoub Salhi Maître de conférences à l'université d'Artois, qui m'a toujours fait bon accueil. Son énergie et sa confiance ont été des éléments moteurs pour moi.

Je tiens à remercier aussi l'ensemble de mes collègues pour leur bonne humeur et leur sympathie. Je tiens à remercier tout particulièrement ceux avec qui j'ai partagé de nombreux moments de convivialité : Atef, Badran, Jerry, Issam, Karim, Michael, Said...

Je tiens aussi à remercier les deux personnes sans qui je ne serai sûrement pas entrain de finir d'écrire ces remerciements. Ils ont eu la patience de m'élever, de me supporter, de me conseiller et de me motiver jour après jour, année après année. Je veux bien évidement parler de mes parents, Zohra et Salah. Je n'oublie pas mon petit frère Oussama et ma petite sœurs Sarah dont leurs présence pendant toutes ces années a également contribué à faire de moi ce que je suis aujourd'hui.

Merci à tous les membres de ma grande famille Dehani et Kelfat, pour leur soutien durant toutes ces années d'études, je ne saurais être qu'infiniment reconnaissant quant aux sacrifices qu'ils ont consentis.

Enfin merci à ceux et celles que je n'ai pas pu citer, mes sincères amitiés et remerciements.

Table des matières

Liste des Figures

Liste des Tables

Liste des Algorithmes

Introduction générale

La *programmation par contraintes* (en anglais CP pour Constraint Programming) est un paradigme puissant utilisé pour la modélisation et la résolution des problèmes de recherche combinatoire qui s'appuie sur un large éventail de techniques de l'intelligence artificielle, recherche opérationnelle, théorie des graphes...etc. L'idée de base en programmation par contraintes est que l'utilisateur donne les contraintes et un solveur de contraintes cherche à les résoudre. Une contrainte est une relation sur un ensemble de variables ayant chacune un domaine de valeurs possibles. Elle exprime une restriction sur les combinaisons possibles de valeurs affectées aux variables. Un réseau de contraintes est défini par un ensemble de variables, et par un ensemble de contraintes.

Le *problème de satisfaction de contraintes* [Montanari, 1974] (en anglais *CSP* pour Constraint Satisfaction Problem), est un problème central de la CP. Il consiste à trouver une affectation pour chaque variable d'une manière à satisfaire toutes les contraintes. Le formalisme CSP permet de représenter naturellement de larges classes de problèmes qui s'expriment sous la forme de contraintes "dures" qui doivent absolument être satisfaites. Cependant, il existe une autre gamme de problèmes nécessitant l'introduction de contraintes "souples" (i.e. des contraintes que l'on peut violer). On utilise ces contraintes souples soit lorsque le problème est sur-contraint (et donc il n'existe pas de solution) auquel cas on recherche une solution qui viole le moins possible les contraintes, soit lorsque l'on veut exprimer des préférences sur les solutions possibles. Dans les deux cas, l'objectif est de trouver une solution qui satisfait au mieux les contraintes exprimées. Il s'avère nécessaire d'évaluer la qualité d'une instanciation pour choisir celles qui satisfont au mieux les contraintes. Plusieurs travaux ont été proposés en ce sens afin d'étendre le formalisme CSP pour permettre d'évaluer les assignations par exemple, selon le coût [Shapiro and Haralick, 1981; Freuder and Wallace, 1992], la priorité [Schiex, 1992b] et la probabilité [Fargier and Lang, 1993].

Le *problème de satisfaction de contraintes valuées* [Schiex et al., 1995] (en anglais VCSP pour Valued Constraint Satisfaction Problem), est un cadre d'optimisation général utilisé avec succès pour manipuler le concept de "contraintes souples" dans de nombreuses applications en intelligence artificielle et en recherche opérationnelle. Une instance de problème VCSP est appelée réseau de contraintes valuées (en anglais VCN pour Valued Constraints network). Elle est définie au moyen d'un ensemble de variables et d'un ensemble de fonctions de coût construites à partir d'une structure d'évaluation. Chaque contrainte souple détermine un degré de violation pour chaque instantiation possible d'un sous-ensemble de variables. Ces degrés (ou valuation) peuvent alors être combinés en utilisant l'opérateur d'agrégation $\oplus$ de la structure de valuation afin d'obtenir le coût global de toute instanciation complète. Le problème de satisfaction de contraintes valuées est de trouver, pour un VCN donné, une instanciation du réseau dont le coût est minimal.

Dans ce mémoire, nous nous intéressons au *problème de satisfaction de contraintes pondérées* [Larrosa, 2002] (en anglais WCSP pour Weighted Constraint Satisfaction Problem). Ce formalisme a été utilisé avec succès pour plusieurs domaines d'application tels que le problème d'affectation de liaison de fréquences radio [Cabon et al., 1999], les enchères combina-

toires [Sandholm, 2002], la bio-informatique et le raisonnement probabiliste [Pearl, 1988]. Le formalisme WCSP représente une extension du formalisme CSP et une spécialisation du formalisme VCSP. Il consiste à trouver une solution pour un réseau de contraintes pondérées (en anglais WCN pour Weighted Constraints network) ou prouver qu'il en existe pas. Un réseau de contraintes pondérées est constitué d'un ensemble de variables ayant chacune un domaine discret et fini, et d'un ensemble de contraintes pondérées ou fonction de coût. La notion de contrainte dans ce formalisme diffère par rapport à celle du formalisme CSP classique. Ici, chaque tuple de contrainte est étiqueté par un entier positif qui représente le coût attribué à ce tuple sur cette contrainte. Une instanciation qui atteint un coût égal à k (paramètre du WCSP représentant un coût prohibitif) est interdite. Ce paramètre k permet donc aussi d'interdire certains tuples. Il peut être soit un entier naturel ou $+\infty$. La tâche usuelle (NP-difficile) du problème de satisfaction de contraintes pondérées est de trouver, pour un WCN donné, une instanciation du réseau dont le coût est minimal.

Au cours de ces dernières années, de nombreux travaux ont étés menés pour adapter au cadre WCSP des algorithmes efficaces définis pour le cadre CSP, généralement dans le but de réduire l'espace de recherche, comme par exemple la cohérence de nœud (NC) ou la cohérence d'arc (AC) [Larrosa, 2002; Larrosa and Schiex, 2004]. D'autres algorithmes de filtrage plus sophistiqué connus aussi sous l'appellation cohérences souples locales ont été proposés tels que EDAC, VAC et OSAC. Une taxonomie partielle de ces cohérences souples peut être trouvée dans [Cooper et al., 2010]. Les algorithmes évoqués ci-dessus s'intègrent à l'algorithme de base de séparation et évaluation (en anglais B&B pour Branch and Bound). Le principe de l'algorithme de séparation et évaluation est de rechercher une solution par séparation du problème en petits sous-problèmes plus facile à résoudre, et ensuite, l'évaluation du coût minimal de chaque sous-problème en utilisant le principe de transfert de coût entre les portées de contraintes.

Dans ce mémoire, nous présentons la notion de substituabilité souple au voisinage pour le cadre WCSP. Nous étudions une première version de la substituabilité qui se base sur les valeurs de variables. Nous proposons un algorithme efficace qui permet de détecter et de supprimer les valeurs souple-substituables au voisinage. Nous présentons aussi deux nouvelles cohérences souples pour le cadre WCSP à savoir, la *cohérence de tuples* (en anglais TC pour Tuple Consistency) et la *cohérence de tuples optimale* (en anglais OTC pour Optimal Tuple Consistency). L'idée de base est d'étendre les deux notions de cohérence d'arc (AC) [Larrosa, 2002; Larrosa and Schiex, 2004] et de cohérence d'arc souple optimale (OSAC) [Cooper et al., 2007] aux tuples. Nous présentons donc un algorithme qui permet d'établir la propriété TC sur un WCN donné, ainsi que le problème de programmation linéaire qui permet de calculer une borne inférieure en établissant la propriété OTC. Ensuite, nous étendons la notion de substituabilité aux tuples de contraintes. Nous notons le fait que la substituabilité de tuples est une généralisation de la substituabilité de valeurs. Enfin nous proposons une généralisation de l'algorithme de substituabilité aux tuples.

Plan du manuscrit

Chapitre 1 : Dans ce chapitre, nous présentons l'état de l'art en satisfaction de contraintes (*CSP*) en introduisant quelques définitions et notions de base que nous utiliserons tout au long de ce manuscrit. Puis, nous présentons les méthodes de résolutions de base telles que le `Backtracking`, le `Forward-Checking` et le `Maintien de l'Arc-Cohérence`. Nous décrivons ensuite quelques algorithmes de filtrage qui se basent sur le concept de cohérence. Enfin, nous présentons quelques heuristiques qui sont utilisées pour guider la recherche sur les espaces de recherche les plus prometteurs.

Chapitre 2 : Dans ce chapitre, nous présentons brièvement le formalisme VCSP qui est utilisé pour modéliser les problèmes d'optimisation sous contraintes. Nous nous focalisons sur notre cadre de travail qui est le problème de satisfaction de contraintes pondérées WCSP. Nous montrons qu'il représente une extension du formalisme CSP et une spécialisation du formalisme VCSP. Ensuite, nous présentons l'état de l'art des différentes cohérences souples proposées pour le cadre WCSP. Enfin, nous faisons un comparatif entre ces cohérences et nous étudions les relations existantes entre elles.

Chapitre 3 : Dans ce chapitre, nous adoptons une perspective différente en revisitant la propriété bien connue de la substituabilité en l'adaptant au cadre souple. Tout d'abord, nous précisons les relations existant entre la substituabilité de voisinage souple (en anglais SNS pour Soft Neighbourhood Substitutability) et une propriété appelée *pcost* qui est basée sur le concept de surcoût de valeurs (par le biais de l'utilisation de paires de surcoût). Nous montrons que sous certaines hypothèses, *pcost* est équivalent à SNS, mais que dans le cas général, elle est plus faible que SNS prouvée être coNP-difficile. Ensuite, nous montrons que SNS conserve la propriété VAC, mais pas la propriété EDAC. Nous introduisons un algorithme optimisé et nous montrons sur diverses séries d'instances WCSP l'intérêt pratique du maintien de *pcost* avec AC*, FDAC ou EDAC, au cours de la recherche.

Chapitre 4 : Dans ce chapitre, nous présentons un nouveau type de cohérence pour les réseaux de contraintes pondérées. Il s'agit de la cohérence de tuples (TC) dont l'établissement sur un WCN est effectué grâce à une nouvelle opération appelée `TupleProject`. Nous proposons également une version "optimale" de cette propriété, OTC, qui peut être perçue comme une généralisation de OSAC (Optimal Soft Arc Consistency). Le principe sous-jacent à OTC est d'appliquer de manière itérative l'opération `TupleProject` afin de factoriser un coût qui maximise la borne inférieure $w_\emptyset$, sur la base de transferts de coûts entre tuples de différentes contraintes d'arité quelconque.

Chapitre 5 : Dans ce chapitre, nous étendons la notion de substituabilité souple aux tuples. Nous nous focalisons sur la notion de substituabilité de tuples au voisinage (SNST) dans le cadre WCSP. Nous proposons une généralisation de la propriété *pcost* pour permettre d'identifier les tuples souples-substituables au voisinage. Enfin, nous proposons un algorithme qui permet d'identifier et éliminer efficacement les tuples souples-substituables au voisinage.

Première partie

État de l’art

Chapitre 1

Problèmes de satisfaction de contraintes

Sommaire

Ce chapitre est constitué de trois sections. Dans la première section, nous présentons l'état de l'art du formalisme du problème de satisfaction de contraintes (*CSP*). Nous introduisons quelques définitions et notions de bases telles que *variable*, *contrainte dure* et *réseau de contraintes*. Nous définissons clairement le problème de satisfaction de contraintes et la notion de solution pour un réseau de contraintes donné. Nous décrivons les différents types de représentation de contraintes en donnant un exemple pour chaque type. En guise d'exemple, nous montrons comment on peut modéliser le problème de coloriage académique à l'aide du formalisme CSP. Enfin, nous présentons les contraintes globales qui représentent un outil puissant pour la modélisation dans ce formalisme.

Dans la seconde section, nous présentons deux méthodes de résolution de base à savoir `Backtracking` et `Forward-Checking`. Nous présentons l'algorithme `Backtracking` ainsi que les deux algorithmes `Look-Ahead` et `Look-Back` qui découlent de `Forward-Checking`.

Dans la troisième section, nous décrivons quelques algorithmes de filtrage qui se basent sur le concept de cohérences telles que AC, PC et SAC. Pour chaque cohérence, nous montrons l'établissement de cette dernière sur un réseau de contraintes donné, nous présentons aussi pour mémoire, l'ensemble des algorithmes proposés dans la littérature pour établir celle-ci ainsi que leur complexité spatiale et temporelle.

Dans la dernière section, nous présentons quelques heuristiques de recherche basées sur les variables et sur les valeurs de variables.

1.1 Le formalisme CSP

Le *problème de satisfaction de contraintes* (en anglais *CSP* pour Constraint Satisfaction Problem), est un formalisme initialement introduit par Montanari [Montanari, 1974]. D'une manière générale, une instance de ce problème est définie par un ensemble de *variables*, chacune ayant un *domaine* de valeurs, et par un ensemble de *contraintes* définies sur des sous-ensembles de variables. Dans tout ce qui suit, nous nous restreindrons aux problèmes de satisfaction de contraintes, dont les domaines de variables sont définis par des ensembles discrets et finis. Dans ce chapitre, nous introduisons quelques définitions et notations de base que nous utiliserons tout au long de ce mémoire.

1.1.1 Définitions et notations

Définition 1 (Variable)**.** *Une variable x est une inconnue qui peut prendre une valeur parmi celles présentes dans un domaine discret et fini noté $dom(x)$.*

Définition 2 (Instanciation)**.** *Une instanciation I d'un ensemble $S = \{x_1, \ldots, x_k\}$ de variables est un ensemble $\{(x_1, v_1), \ldots, (x_k, v_k)\}$ tel que $v_1 \in dom(x_1), \ldots, v_k \in dom(x_k)$; l'ensemble des variables apparaissant dans I est noté $vars(I)$ et l'ensemble des instanciations possibles de S est noté $l(S)$ (en anglais l pour labeling).*

Par exemple, pour l'ensemble de variables $\{x, y\}$, si nous supposons que le domaine des variables x et y contient deux valeurs a et b, l'ensemble des instanciations possibles $l(\{x, y\})$ est donné par la table suivante :

$l(\{x, y\})$
$\{(x, a), (y, a)\}$
$\{(x, a), (y, b)\}$
$\{(x, b), (y, a)\}$
$\{(x, b), (y, b)\}$

Définition 3 (Projection)**.** *La projection d'une instanciation I sur un ensemble de variables $S \subseteq vars(I)$ est une instanciation notée $I[S]$ qui est la restriction de I aux variables de S.*

Par abus de notation, $I[x]$ représente la valeur de la variable x dans l'instanciation I.

Avant de donner la définition classique d'une contrainte dure, nous introduisons la notion de relation sur un ensemble de variables.

Définition 4 (Relation)**.** *Pour tout ensemble de variables S, une relation sur l'ensemble S est un sous-ensemble de $l(S)$.*

Définition 5 (Contrainte dure)**.** *Une contrainte dure c_S porte sur l'ensemble de variables S, et elle est définie sémantiquement par une relation, noté $rel(c_S)$, représentant l'ensemble des instanciations autorisées pour l'ensemble de variables S.*

Dans tout ce qui suit, une instanciation d'un ensemble de variables sur lesquelles portent une contrainte est appelée "*tuple*". Pour une contrainte donnée c_S, on utilise alors le terme de tuple de la contrainte c_S pour désigner une instanciation de S.

Remarque 1. *Pour faire le lien avec le chapitre 2, nous adoptons la définition d'une contrainte dure comme suit : une contrainte dure c_S associe à chaque tuple t de $l(S)$ un booléen $c_S(t)$. Un tuple t est autorisé par la contrainte c_S si et seulement si $c_S(t)$ est vrai.*

Pour toute instanciation I d'un ensemble de variables Y et toute contrainte c_S tel que $S \subseteq Y$, $c_S(I)$ sera considéré comme égal à $c_S(I[S])$. Autrement dit, les projections seront implicites. Pour indiquer qu'un tuple t est autorisé par la contrainte c_S, nous pourrons simplement écrire que $c_S(t) = vrai$, que l'on peut simplifier selon le contexte en $c_S(t)$. À partir de maintenant et seulement dans ce chapitre, nous utiliserons le terme contrainte pour désigner une contrainte dure.

Définition 6 (Portée d'une contrainte)**.** *Pour toute contrainte c_S, l'ensemble de variables S représente la portée (ou le scope) de la contrainte c_S.*

Définition 7 (Arité d'une contrainte)**.** *L'arité d'une contrainte c_S représente le nombre de variables $|S|$ dans l'ensemble S.*

En se basant sur cette dernière définition, nous pouvons distinguer les contraintes dures selon leur arité comme suit :

- une contrainte est dite unaire si son arité est égale à un ;
- une contrainte est dite binaire si son arité est égale à deux ;
- une contrainte est dite n-aire si son arité est supérieure ou égale à trois.

Nous présentons maintenant la définition d'un *réseau de contraintes* pour le formalisme CSP. Un réseau de contraintes permet de définir une instance CSP.

Définition 8 (Réseau de contraintes)**.** *Un réseau de contraintes (en anglais CN pour Constraint Network) est un couple $P = (\mathscr{X}, \mathscr{C})$, où $\mathscr{X}$ représente l'ensemble des variables et $\mathscr{C}$ l'ensemble des contraintes dures.*

Étant donné un réseau de contraintes P, on note :

- n le nombre de variables du réseau P ;
- d la taille du plus grand domaine d'une variable de P ;
- e le nombre de contraintes du réseau P ;
- r l'arité maximale des contraintes de P.

1.1.2 Satisfaction de contraintes

Le problème de satisfaction de contraintes consiste à trouver une affectation pour les variables d'un réseau de contraintes donné de manière à satisfaire toutes les contraintes de ce réseau.

Définition 9 (Satisfaction d'une contrainte)**.** *Soient $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes et I une instanciation d'un sous-ensemble de $\mathscr{X}$. On dit que I satisfait la contrainte c_S si et seulement si $S \subseteq vars(I)$ et $c_S(I[S])$.*

En d'autres termes, une instanciation I satisfait une contrainte c_S si et seulement si I couvre la contrainte c_S ($S \subseteq vars(I)$) et la projection de I sur l'ensemble S produit un tuple autorisé par la contrainte c_S.

Définition 10 (Instanciation localement cohérente)**.** *Soient $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes et I une instanciation d'un sous-ensemble de $\mathscr{X}$. On dit que I est localement cohérente si et seulement si pour toute contrainte c_S telle que $S \subseteq vars(I)$, I satisfait la contrainte c_S.*

Ainsi, I est localement cohérente si et seulement si toutes les contraintes c_S dont la portée est incluse dans $vars(I)$ sont satisfaites.

Définition 11 (Types d'instanciation)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. Une instanciation I est dite partielle pour P si $vars(I)$ est un sous-ensemble strict de l'ensemble $\mathscr{X}$. Elle est dite complète pour P si $vars(I) = \mathscr{X}$.*

Le problème de satisfaction de contraintes (CSP) consiste à trouver une solution pour un réseau de contraintes. La notion de solution pour un réseau de contraintes est donnée par la définition suivante :

Définition 12 (Solution d'un réseau de contraintes)**.** *Soient $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. Une instanciation complète I pour P est une solution de P si et seulement si $\forall c_S \in \mathscr{C}$, c_S est satisfaite par I. Autrement dit, une solution est une instanciation localement cohérente et complète.*

On peut aussi généraliser la définition de solution comme étant une instanciation partielle dont toutes les extensions complètes sont localement cohérentes. Cependant, cette définition n'est pas nécessaire dans nos travaux.

De cette dernière définition nous retenons qu'une solution pour un réseau de contraintes est une instanciation complète qui satisfait toutes les contraintes. Nous désignons par $sols(P)$ l'ensemble des solutions possibles pour le réseau de contraintes P. Si le réseau de contraintes P contient un domaine de variables vide ou s'il comporte une relation vide (i.e. une contrainte avec uniquement des tuples interdits) alors aucune solution ne peut satisfaire P ($sols(P) = \emptyset$), et celui-ci est trivialement insatisfaisable, noté $P = \bot$.

Dans la plupart des cas, pour résoudre un CN P, on doit le transformer en un autre CN P' équivalent et plus facile à résoudre. La définition suivante capture brièvement la notion d'équivalence entre réseaux de contraintes classiques.

Définition 13. *Soient $P = (\mathscr{X}, \mathscr{C})$ et $P' = (\mathscr{X}', \mathscr{C}')$ deux réseaux de contraintes. P et P' sont équivalents si et seulement s'ils ont le même ensemble de variables ($\mathscr{X} = \mathscr{X}'$) et le même ensemble de solutions ($sols(P) = sols(P')$).*

Nous verrons des exemples de réseaux de contraintes équivalents dans la section 1.3.

1.1.3 Arités des réseaux de contraintes

Selon l'arité des contraintes, nous distinguons deux grandes catégories de réseaux de contraintes, à savoir, les réseaux de contraintes binaires et les réseaux de contraintes n-aires.

Définition 14 (Les réseaux de contraintes binaires)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un réseau de contraintes. P est dit binaire si et seulement si $\forall c_S \in \mathscr{C}$, $|S| \leq 2$ et $\exists c_S \in \mathscr{C}$, tel que $|S| = 2$.*

En d'autre termes, un réseau de contraintes est binaire si toutes ses contraintes sont des contraintes binaires et/ou unaires et dont au moins une contrainte binaire.

Définition 15 (Les réseaux de contraintes n-aires)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN. P est dit n-aire si et seulement si $\exists c_S \in \mathscr{C}$, tel que $|S| > 2$.*

En d'autres termes, un réseau de contraintes est n-aire s'il existe au moins une contrainte d'arité strictement supérieure à deux.

La structure d'un réseau de contraintes peut être représentée par un graphe $G = (S, A)$, appelé le *graphe de contraintes* ou *macrostructure*, dont les sommets sont les variables de l'ensemble $\mathscr{X}$ et les arêtes sont les scopes des contraintes de l'ensemble $\mathscr{C}$. Si le réseau de contraintes est n-aire alors G est un hypergraphe.

Notons qu'il existe plusieurs techniques dite de binarisation pour transformer un réseau de contraintes n-aire en un réseau de contraintes binaires équivalent. Parmi ces dernières nous pourrons citer l'encodage dual (en anglais dual encoding) initialement appelé l'encodage du graphe dual [Dechter and Pearl, 1988] (en anglais dual graph encoding), l'encodage double (en anglais double encoding) [Stergiou and Walsh, 1999] et l'encodage des variables cachées (en anglais hidden variable encoding) [Rossi et al., 1990].

1.1.4 Représentations des contraintes

Les contraintes peuvent être représentées en intension ou en extension :

- **intension :** la contrainte est représentée sous forme de fonction mathématique. La relation entre les variables impliquées dans la contrainte est définie par une formule booléenne (*prédicat*).
- **extension :** la contrainte est représentée sous la forme d'une liste de tuples autorisés (ou interdits) pour les variables de sa portée. Trois représentations équivalentes sont couramment utilisées :
  - **une relation explicite :** dans ce cas, la relation représente soit l'ensemble des tuples autorisés par la contrainte, soit l'ensemble des tuples interdits par la contrainte.
  - **une table de vérité :** la contrainte est représentée par une table de vérité en listant explicitement l'ensemble des tuples possibles de sa portée avec le booléen correspondant (un tuple autorisé sera mentionné à vrai, tandis qu'un tuple interdit sera mentionné à faux).

- **un graphe de compatibilité :** la contrainte est représentée sous forme d'un graphe de compatibilité ou microstructure. L'ensemble des sommets représente les différents couples (*variable*, *valeur*). Les valeurs compatibles sont reliées entre elles avec une arête (une ligne continue). Dans le cas où nous voulons exprimer un graphe d'incompatibilité, les valeurs incompatibles sont reliées entre elles avec une ligne en pointillés.

Pour bien comprendre ces deux types de représentation de contraintes, considérons l'exemple 1 ci-dessous.

Exemple 1. *Supposons que nous voulions représenter une contrainte qui porte sur deux variables x et y avec deux valeurs a, b pour chaque variable. Cette contrainte doit exprimer le fait que les deux valeurs assignées à x et y doivent être différentes. Cette contrainte peut être exprimée :*

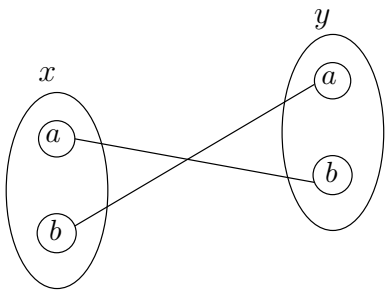
- ***en intension,*** *par la fonction mathématique :* $c_{xy} := (x \neq y)$,
- ***en extension,*** *par :*
  - ***une relation explicite :*** *par exemple, en listant l'ensemble des tuples autorisés comme suit :*

c_{xy}
$\{(x,a),(y,b)\}$
$\{(x,b),(y,a)\}$

  - ***une table de vérité :***

x	y	c_{xy}
a	a	$faux$
a	b	$vrai$
b	a	$vrai$
b	b	$faux$

  - ***un graphe de compatibilité :***



1.1.5 Un exemple académique : le problème du coloriage de carte

Le cadre CSP permet de représenter de nombreux problèmes de manière simple et efficace. Dans ce qui suit nous rappelons un problème académique connu sous l'appellation "problème du coloriage de carte". Ce problème consiste à colorier chaque région d'une carte donnée avec une couleur (l'ensemble de couleurs est donné) de telle sorte que deux régions adjacentes n'aient pas la même couleur. Pour bien illustrer ce problème, considérons l'exemple de la

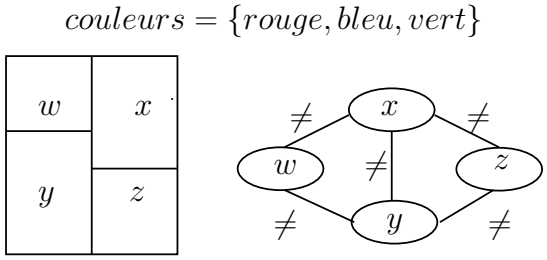


FIGURE 1.1 – La modélisation du problème du coloriage de carte en CSP

figure 1.1. Sur cet exemple, le but est d'affecter une couleur (*valeur*) pour chaque région (*variable*) de manière à respecter le fait que chaque couple de régions adjacentes n'aient pas la même couleur (*contrainte*). En terme CSP, ce problème peut être formalisé de la manière suivante :

- Le CN $P = (\mathscr{X}, \mathscr{C})$ tel que :
  - $\mathscr{X} = \{w, x, y, z\}$ est l'ensemble des variables avec $dom(w) = dom(x) = dom(y) = dom(z) = \{r, b, v\}$.
  - $\mathscr{C} = \{c_{wx}, c_{xz}, c_{zy}, c_{wy}, c_{xy}\}$ tel que $\forall c_{x_1x_2} \in \mathscr{C}$, $c_{x_1x_2} := (x_1 \neq x_2)$.

Une solution pour ce problème est : $S = \{(w, v), (x, r), (y, b), (z, v)\}$.

1.1.6 Contraintes globales

Une *contrainte globale* [Bessiere and van Hentenryck, 2003] [Régin, 2003] [Rossi et al., 2006] est un modèle (motif) de contraintes capturant une sémantique précise et pouvant porter sur un nombre quelconque de variables. La plupart du temps, un propagateur spécifique et très efficace est associé à une contrainte globale donnée. Elle exprime une relation entre un nombre non fixé de variables d'un réseau de contraintes. Parmi ces contraintes nous citons la contrainte `alldifferent`$(x_1, x_2, \ldots, x_n)$ [Régin, 1994], elle exprime le fait que toutes les valeurs des variables passées en paramètre doivent être deux à deux différentes. La modélisation classique d'un tel problème est faite avec des contraintes de différence binaire du type $x_i \neq x_j$ tel que $1 \leq i \leq n$ et $i < j \leq n$. Grâce à la contrainte `alldifferent` nous avons remplacé les $\frac{n(n-1)}{2}$ contraintes par une seule contrainte plus efficace en terme de suppression de valeurs qui ne peuvent pas mener à une solution, notion dite de *filtrage*. Pour bien comprendre ce point, supposons qu'on veut exprimer le fait que les valeurs des trois variables x, y et z soient différentes et supposons que le domaine de ces variables est constitué de deux valeurs a et b. La modélisation classique ($x_i \neq x_j$) ne va pas filtrer le domaine des variables car chaque valeur de la première variable peut avoir une valeur différente dans le domaine de la deuxième variable. En revanche, `alldifferent` va détecter trivialement que ce problème est incohérent. En effet, `alldifferent` identifie qu'il n'y a que deux valeurs possibles pour les trois variables, et que donc elles ne peuvent être toutes différentes. La contrainte `cumulative` [Aggoun and Beldiceanu, 1992] est un autre exemple de contrainte globale. Elle permet la modélisation des problèmes d'ordonnancement tels que l'affectation de tâches à des machines. Elle exprime le fait qu'à tout instant, le total des ressources utilisées par un en-

semble de tâches pour une machine donnée ne dépasse pas une certaine limite. La syntaxe de `cumulative` est `cumulative`(Débuts, Durées, Ressources, Limite, Fin) où Débuts est la liste des dates de début des tâches, Durées la liste de leurs durées, Ressources la liste des quantités de ressource qu'elles utilisent, Limite le maximum de ressources disponibles, et Fin la date de fin de toutes les tâches. Il est clair que `cumulative` simplifie la modélisation du problème d'ordonnancement. Par ailleurs, un algorithme de filtrage efficace de cette contrainte existe.

1.2 Méthodes de résolution des réseaux de contraintes

Il existe plusieurs requêtes classiques pour un réseau de contraintes :
- Déterminer si une solution existe.
- Trouver une solution.
- Trouver toutes les solutions.
- Compter le nombre de solutions.
- Trouver toutes les valeurs possibles pour une variable qui peuvent apparaître dans une solution.
- ...

La première requête est un problème de décision, son but est de savoir si le réseau de contraintes est cohérent ou non. Dans ce cas on parle de la résolution du réseau de contraintes. Ce problème est combinatoire et appartient à la classe des problèmes *NP-Complets*.

Un algorithme de base utilisé pour résoudre un réseau de contraintes est la procédure *`Backtracking`*. Dans ce qui suit nous détaillons cette méthode en analysant les aspects négatifs d'un tel algorithme.

1.2.1 L'algorithme `Backtracking` (BT)

La procédure `Backtracking` consiste à affecter des valeurs aux variables d'un réseau de contraintes dans un ordre prédéfini. À chaque affectation d'une variable à une valeur de son domaine, un test de compatibilité (de cohérence) est réalisé avec l'ensemble des valeurs correspondant aux variables déjà affectées, cet ensemble représente l'*instanciation partielle courante*. Ce test de compatibilité ou de cohérence consiste à s'assurer que l'instanciation courante ne viole aucune contrainte. Sinon, on obtient un conflit et dans ce cas il faut chercher une autre valeur dans le domaine de la variable en cours d'affectation. Si cette variable ne possède aucune valeur compatible (elles ont été toutes testées) avec les valeurs de la solution partielle courante, un retours arrière chronologique est effectué. L'algorithme procède ainsi jusqu'à ce que toutes les variables du réseau de contraintes soient affectées, ce qui constitue une solution du réseau de contraintes et une preuve de la satisfaisabilité de ce réseau. Dans le cas contraire, le réseau de contraintes est dit incohérent et il n'admet pas de solution. L'algorithme `Backtracking` est donné par l'algorithme 1. Dans cet algorithme, I représente l'instanciation partielle courante, i le niveau de la variable dans l'arbre de recherche et x_i une variable du problème. L'appel initial est `Backtracking`$(\emptyset, 1)$.

Cet algorithme peut être utilisé pour trouver une solution ou l'ensemble de toutes les solutions possibles pour un réseau de contraintes donné (après une modification mineure du code). Cependant, il présente plusieurs inconvénients, parmi lesquels nous citons :

Algorithme 1 : `Backtracking`

Entrées : I : Instanciation, i : Niveau
Sorties : Booléen

```
1  si I est localement cohérente alors
2      si i = n alors
3          retourner vrai;
4      sinon
5          pour chaque v ∈ dom(x_i) faire
6              si Backtracking(I ∪ {(x_i, v)}, i + 1) alors retourner vrai;
7          fin
8      fin
9  fin
10 retourner faux ;
```

- **Le retour arrière chronologique :** dans la plupart des cas, le retour arrière chronologique ne résout pas le problème d'incompatibilité entre les valeurs de variables. En effet, si l'avant dernière variable affectée n'est pas la cause du conflit, cela provoquera à nouveau le même problème. Nous verrons dans la section suivante, qu'il est possible de faire des retours en arrière plus intelligents.
- **Des tests de cohérence inutiles :** supposons que nous ayons détecté sur une branche de l'arbre de recherche une incohérence. Cette dernière n'est pas mémorisée et peut ainsi se reproduire sur une autre branche de l'arbre de recherche qui porte sur le même ensemble de valeurs.

Pour remédier aux inconvénients cités ci-dessus, des algorithmes qui sont des raffinements du `Backtracking` ont été proposés. Le principe de ces algorithmes est d'analyser ou d'anticiper les situations d'échec pendant la recherche. Ces algorithme sont divisées en deux approches : *Look-Ahead* et *Look-Back*.

1.2.2 Look-Ahead

Initialement introduit dans [Haralick and Elliott, 1980], le principe des algorithmes qui se basent sur l'approche Look-Ahead est d'effectuer un filtrage du domaine des variables afin d'anticiper les échecs et ainsi réduire l'espace de recherche.

Parmi ces algorithmes on trouve le `Partiel-Lookahead` et `Full-Lookahead`. Dans ce qui suit, nous prenons pour exemple l'algorithme `Forward-Checking`. Une première version de cet algorithme qui se limite aux réseaux binaires a été proposée dans [Haralick and Elliott, 1980]. Les étapes de la méthode `Forward-Checking` sont données par l'algorithme 2. La boucle *pour* de la ligne 4 de cet algorithme filtre les domaines de variables x_j qui ne sont pas encore assignées et qui sont liées à la variable x_i (dernière variable assignée). La ligne 8 vérifie si le domaine de l'une des variables x_j est vide, si c'est le cas l'algorithme retourne faux (le CN est incohérent). Un appel récursif de l'algorithme sur la variable suivante est effectué à la ligne 9. Si on arrive à assigner toutes les variables l'algorithme retourne vrai, ce qui signifie

que le CN est cohérent.

Algorithme 2 : `Forward-Checking`

Entrées : I : Instance, i : Niveau
Sorties : Booléen

```
1  si i = n alors
2      retourner vrai;
3  sinon
4      pour chaque v ∈ dom(x_i) faire
5          pour j ← i + 1 à n faire
6              si c_{x_i x_j} ∈ 𝒞 alors Supprimer les valeurs v ∈ dom(x_j) qui sont incompatibles
               avec (x_i, v) dans la contrainte c_{x_i x_j};
7          fin
8          si ∃j ∈ i + 1..n t.q dom(x_j) = ∅ alors
9              retourner faux;
10         sinon
11             si Forward-Checking(I ∪ (x_i, v), i + 1) alors retourner vrai;
12         fin
13     fin
14     retourner faux ;
15 fin
```

Il existe d'autres formes plus poussées de l'approche Look-Ahead basées sur des propriétés bien définies appelées cohérences locales. Nous détaillerons quelques-unes de ces formes en section 1.3.

1.2.3 Look-Back

Dans cette approche, les retours en arrière sont réalisés après une analyse du conflit contrairement au `Backtracking` chronologique qui fait des retours en arrière systématique sur la dernière variable. Dans ce type d'approche, on trouve par exemple :

- **Gaschnig-backjumping [Gaschnig, 1979] :** Pour expliquer brièvement cette méthode, supposons que la variable en cours d'affectation x_i n'ait pas de valeur compatible avec les valeurs de variables déjà affectées (instanciation courante). Dans ce cas, l'algorithme sélectionne pour chaque valeur a du domaine de x_i la variable dite "coupable" (de l'instanciation partielle courante) la plus récemment affectée et incompatible avec a. Ensuite, l'ensemble de ces variables coupables est utilisé dans un second temps pour effectuer un saut vers la variable coupable la plus récemment affectée.
- **Graph-based backjumping [Freuder, 1982; Dechter and Pearl, 1988] :** Dans cette méthode, lorsqu'une impasse est rencontrée, le retour arrière est réalisé en tenant compte de la cause de l'échec et en se basant essentiellement sur la structure du graphe de contraintes. Supposons que lors de l'instanciation de la variable courante x_i, plus aucune valeur dans le domaine de cette variable n'est cohérente avec l'instanciation courante.

Il s'agit certainement d'une incohérence entre la variable x_i et une variable voisine à celle-ci.

- **Conflict-directed backjumping [Prosser, 1993] :** La méthode de retour en arrière dirigée par les conflits (en anglais *CBJ* pour Conflict-directed BackJumping) se base sur le graphe de contraintes et sur des informations recueillies au cours de la recherche. Cette méthode associe à chaque variable une liste de variables déjà affectées, cet ensemble représente l'ensemble des saut induits.
- **Dynamic Backtracking [Ginsberg, 1993] :**
  Lorsqu'un conflit apparaît suite à une affectation, les techniques de retour-arrière que nous avons présentées précédemment effectuent un saut dans l'arbre de recherche sur la variable dite coupable en remettant en cause les décisions prises depuis l'affectation de cette variable. En procédant ainsi, ces techniques ignorent des informations qui peuvent être utilisées durant le processus de recherche. La méthode de retour arrière dynamique (en anglais *DBT* pour Dynamic BackTracking) commence par détecter l'affectation la plus récente impliquée dans le dernier conflit. Une fois que l'affectation responsable de la contradiction est détectée, DBT supprime uniquement l'information dépendante de cette décision et n'effectue pas de retour-arrière.

En résumé, nous avons présenté différentes formes d'algorithme qui se basent sur le principe du Backtracking. Nous avons mentionné le fait que des tests de *cohérences* (compatibilité) peuvent être réalisés. Dans la section suivante, nous nous focalisons sur ce type de propriétés appelées cohérences locales.

1.3 Cohérences locales

Dans cette section, nous présentons quelques formes de cohérences locales classiques proposées dans le cadre CSP à savoir la *cohérence d'arc*, la *cohérence de chemin* et la *cohérence d'arc singleton*. Pour chaque forme de cohérence, nous montrons avec un exemple la façon d'établir cette forme sur un réseau de contraintes.

1.3.1 La cohérence d'arc (AC)

Dans ce qui suit, nous présentons la notion de support pour une valeur de variable sur une contrainte donnée.

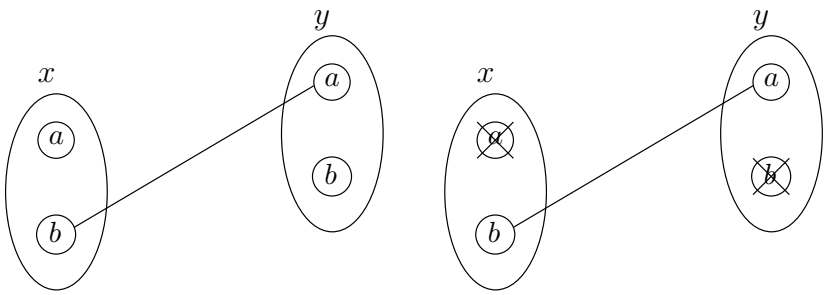
Définition 16 (Support d'une valeur)**.** *Soient x une variable, a une valeur de $dom(x)$ et c_S une contrainte telle que $x \in S$. Un support pour la valeur (x, a) sur la contrainte c_s est un tuple $t \in l(S)$ tel que $t[x] = a \wedge c_S(t)$.*

Si une valeur (x, a) ne possède pas de support sur l'une des contraintes qui portent sur la variable x, cette valeur peut être supprimée du domaine de la variable x. Notons que la suppression de la valeur (x, a) préserve l'équivalence car cette valeur ne peut pas apparaître dans une solution du CN. Cette suppression simplifie la résolution du problème.

En se basant sur la définition ci-dessus, la définition de la cohérence d'arc (en anglais *AC* pour Arc-Consistency) est donnée par :

Définition 17 (Cohérence d'arc)**.** *Une valeur* (x, a) *est dite arc-cohérente (AC) si elle possède un support sur chaque contrainte* c_S *qui porte sur la variable x. Une variable x est AC si son domaine n'est pas vide (*$dom(x) \neq \emptyset$*) et si toutes ses valeurs sont AC. Un CN P est AC si toutes ses variables sont AC.*

Dans le but d'illustrer ce type de cohérence, soit l'exemple du réseau de contraintes (CN) de la figure 1.2.



(a) P (b) P'

FIGURE 1.2 – L'établissement de la propriété *AC* sur un réseau de contraintes

Nous remarquons que le CN P de la figure 1.2 n'est pas *AC* car par exemple la valeur (x, a) n'a pas de support sur la contrainte c_{xy}. Pour établir la cohérence d'arc sur P, nous supprimons la valeur a (resp. b) du domaine de la variable x (resp. la variable y). Cette opération transforme P en P' qui lui est équivalent. Cette fois, le CN P' obtenu après la suppression de la valeur (x, a) et (y, b) est *AC*. L'établissement de la propriété *AC* sur le CN P de la figure 1.2 est assez simple et évidente. Dans le cas d'un réseau de contraintes complexe (avec de nombreuses variables et contraintes), la suppression d'une valeur d'une variable peut provoquer la perte d'un support pour une autre valeur d'une autre variable voisine, processus dit de *propagation de contraintes*. Pour établir la propriété de la cohérence d'arc plusieurs algorithmes ont été proposés. Parmi ces algorithmes, on trouve AC1, AC3 [Mackworth, 1977], AC4 [Mohr and Henderson, 1986], AC5 [van Hentenryck et al., 1992], AC6 [Bessiere, 1994], AC7 [Bessiere et al., 1999], AC8 [Chmeiss and Jégou, 1998], AC2001 [Bessiere and Régin, 2001] et AC3^{rm} [Lecoutre and Hemery, 2007]. Ces algorithmes diffèrent par leur complexité. Le tableau 1.1 donne la complexité en temps et en espace de chaque algorithme dans le cadre des réseaux de contraintes binaires cité ci-dessus.

À titre d'exemple, dans ce qui suit nous présentons l'algorithme AC3^{rm}. Pour établir AC3^{rm} sur un réseau de contraintes donné $P = (\mathscr{X}, \mathscr{C})$, il faut appeler l'algorithme 3. Dans un premier temps, l'algorithme initialise une queue (Q) avec tout couple possible (c_S, x) appelé arc composé d'une contrainte c_S et d'une variable x appartenant à sa portée. Une fois que la queue est initialisée, chaque arc (c_S, x) est révisé en effectuant un appel à la fonction `réviser` (algorithme 4). Cette fonction supprime les valeurs qui n'ont pas de support pour x sur la contrainte c_S et renvoie vrai si au moins une révision est effective. La fonction `réviser` utilise une structure de données $supp[c_S, x, a]$ pour enregistrer le support de la valeur (x, a) sur la contrainte c_S. Si lors d'un appel de la fonction `réviser` une valeur (x, a) n'a plus de support sur la contrainte c_S ($supp[c_S, x, a]$ n'est pas valide, cette vérification est

algorithme	complexité en temps	complexité en espace
AC1 [Mackworth, 1977]	$O(end^3)$	$O(n^2)$
AC3 [Mackworth, 1977]	$O(ed^3)$	$O(e + nd)$
AC4 [Mohr and Henderson, 1986]	$O(ed^2)$	$O(ed^2)$
AC5 [van Hentenryck et al., 1992]	$O(ed^2)$	$O(ed)$
AC6 [Bessiere, 1994]	$O(ed^2)$	$O(ed)$
AC7 [Bessiere et al., 1999]	$O(ed^2)$	$O(ed)$
AC8 [Chmeiss and Jégou, 1998]	$O(ed^2)$	$O(n)$
AC2001 [Bessiere and Régin, 2001]	$O(ed^2)$	$O(ed)$
AC3^{rm} [Lecoutre and Hemery, 2007]	$O(ed^3)$	$O(ed)$
AC3^{bit} [Lecoutre and Vion, 2008]	$O(ed^3)$	$O(ed)$

TABLE 1.1 – La complexité dans le pire cas des algorithmes qui établissent la cohérence d'arc sur des réseaux de contraintes binaires

Algorithme 3 : AC3^{rm}

Entrées : $P = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

```
1 Q ← {(c_S, x) | c_S ∈ C ∧ x ∈ S} ;
2 tant que Q ≠ ∅ faire
3     choisir puis éliminer (c_S, x) de Q ;
4     si revise(c_S, x) alors
5         si dom(x) = ∅ alors retourner faux ;
6         Q ← Q ∪ {(c_S', y) | c_S' ∈ C, c_S' ≠ c_S, y ≠ x, {x, y} ⊆ S'} ;
7     fin
8 fin
9 retourner vrai ;
```

Algorithme 4 : `réviser` : Réviser les supports des valeurs d'une variable donnée

Entrées : c_S : Contrainte, x : Variable
Sorties : Booléen

```
1 nbElements= |dom(x)| ;
2 pour chaque a ∈ dom(x) faire
3     si supp[c_S, x, a] est valide alors continue ;
4     t ← trouverSupport(c_S, x, a);
5     si t = ⊤ alors dom(x) = dom(x) \ {a} ;
6     sinon pour chaque y ∈ S faire supp[c_S, x, t[y]] ← t
7 fin
8 retourner nbElements≠ |dom(x)| ;
```

Algorithme 5 : `trouverSupport` : Trouver un support dans le cadre CSP

Entrées : c_S : Contrainte, x : Variable, a : Valeur
Sorties : Tuple

```
1 t ← ⊥;
2 tant que t ≠ ⊤ faire
3 |   si c_S(t) alors retourner t ;
4 |   t ← tupleSuivant(c_S, x, a, t) ;
5 fin
6 retourner ⊤ ;
```

faite sur la ligne 3 de l'algorithme 4), la fonction `réviser` fait appel (à la ligne 4) à une autre fonction qui s'appelle `trouverSupport`(c_S, x, a) et qui consiste à trouver un nouveau support pour la valeur (x, a). La fonction `trouverSupport` parcourt séquentiellement les tuples t de la contrainte c_S tels que $t[x] = a$ grâce à la fonction `tupleSuivant`(c_S, x, a, t) qui utilise un ordre sur les tuples où $\bot$ représente un tuple qui précède le premier tuple et $\top$ est un tuple non valide qui succède le dernier tuple selon l'ordre imposé par `tupleSuivant`. Si la fonction `trouverSupport`(c_S, x, a) renvoie un tuple valide t alors toutes les valeurs apparaissant dans t auront t comme un nouveau support sur la contrainte c_S. Sinon (si elle renvoie $\top$) la valeur a est supprimé du domaine de la variable x. Lorsque la révision d'un arc (c_S, x) est effective, autrement dit, lorsque le domaine d'une variable change, la queue Q est mise à jour en ajoutant tous les couple $(c_{S'}, y)$ tel que y est une variable voisine à x (il existe au moins une contrainte qui porte à la fois sur x et y) et $c_{S'}$ est toute contrainte (sauf la contrainte c_S) qui porte au moins sur les deux variables x et y. L'algorithme 3 s'arrête quand un domaine d'une variable devient vide ou quand l'ensemble Q devient vide.

Sur les réseaux de contraintes binaires, AC3^{rm} à été prouvé particulièrement efficace (voir les résultats des compétitions de solveurs). Lorsque les domaines sont grands, la version AC3^{bit+rm} est un choix judicieux.

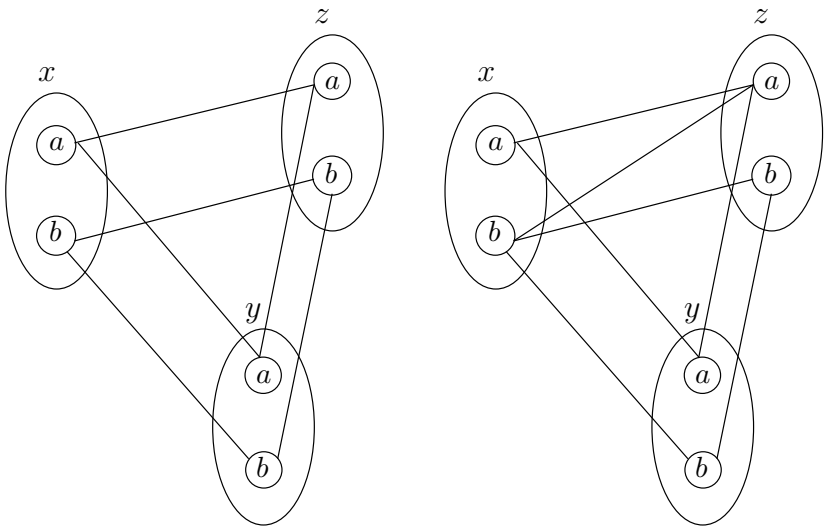
1.3.2 La cohérence de chemin (PC)

Dans cette section, nous présentons un autre type de cohérence connu sous l'appellation cohérence de chemin (en anglais *PC* pour Path Consistency). La cohérence de chemin est initialement introduite dans [Mackworth, 1977]. Par souci de simplicité, nous limitons notre présentation aux réseaux de contraintes binaires.

Définition 18 (cohérence de chemin)**.** *Soient $P = (\mathscr{X}, \mathscr{C})$ un CN et $I = \{(x, a), (y, b)\}$ une instanciation d'un couple de variables. On dit que I est chemin-cohérent si et seulement si $c_{xy}(I) = vrai$ et $\forall z \in \mathscr{X}$ tel que $c_{xz} \in \mathscr{C}$ et $c_{yz} \in \mathscr{C}$, $\exists c \in dom(Z)$ avec $c_{xz}(\{(x, a), (z, c)\}) = vrai$ et $c_{yz}(\{(y, b), (z, c)\}) = vrai$. On dit que la paire de variables $\{x, y\}$ est chemin-cohérente si et seulement si $\forall I \in l(\{x, y\})$, si I est localement cohérente alors, I est chemin-cohérent. On dit que P est chemin-cohérent si et seulement si tout couple de variables de P est chemin-cohérent.*

Autrement dit, un réseau de contraintes est PC-cohérent si et seulement si toute instanciation cohérente de deux variables peut être étendue à une troisième variable. Pour comprendre

la cohérence de chemin, considérons l'exemple des deux CNs P et P' de la figure 1.3.



(a) P (b) P'

FIGURE 1.3 – La cohérence de chemin sur un réseau de contraintes

Le CN P est *PC* car tout couple de variables à savoir $\{x, y\}$, $\{x, z\}$ et $\{y, z\}$ sont chemin-cohérents. En revanche, le CN P' n'est pas *PC* car le couple de variables $\{x, z\}$ n'est pas *PC* . En effet, pour l'instanciation $I = \{(x, b), (z, a)\}$ il n'existe pas de valeur dans le domaine de la variable y qui est compatible avec cette instanciation.

Pour établir la cohérence de chemin plusieurs algorithmes ont été proposés. Nous citons par exemple : PC1 [Mackworth, 1977], PC2 [Mackworth, 1977], PC3 [Mohr and Henderson, 1986] [1], PC4 [Han and Lee, 1988], PC5 [Singh, 1996], PC6[Bessiere, 1994] [2], PC7 [Chmeiss and Jégou, 1996], PC8 [Chmeiss and Jégou, 1998], PC2001 [Bessiere et al., 2005], sDC2 et sDC3 [Lecoutre et al., 2007]. Le tableau 1.2 résume la complexité temporelle est spatiale de chacun des algorithmes que nous avons cités ci-dessus. Notons que pour l'algorithme sDC2 la variable λ représente le nombre de tuples autorisés sur toutes les contraintes de $\mathscr{C}$.

Il faut également remarquer que sur les réseaux de contraintes binaires l'algorithme sDC2 s'avère être plus performant que les algorithmes antérieurs.

1.3.3 La cohérence d'arc singleton (SAC)

Le dernier type de cohérence que nous présentons est la cohérence d'arc singleton (en anglais *SAC* pour Singleton Arc-Consistency). Ce type est initialement introduit dans [Debruyne and Bessiere, 1997]. Dans ce qui suit, nous donnons la définition de ce type de cohérence et illustrons celle-ci. Puis, nous examinons les différents types d'algorithmes proposés dans la littérature afin d'établir SAC sur un réseau de contraintes donné.

1. Dans [Han and Lee, 1988], les auteurs ont montré que l'algorithme PC3 est incorrect et proposent une version corrigé qui est l'algorithme PC4

2. Une généralisation naturelle de l'algorithme AC6

algorithme	complexité en temps	complexité en espace
PC1 [Mackworth, 1977]	$O(n^5d^5)$	$O(n^3d^2)$
PC2 [Mackworth, 1977]	$O(n^3d^5)$	$O(n^3d^3)$
PC4 [Han and Lee, 1988]	$O(n^3d^3)$	$O(n^3d^3)$
PC5 [Singh, 1996]	$O(n^3d^3)$	$O(n^3d^2)$
PC6 [Bessiere, 1994]	$O(n^3d^3)$	$O(n^3d^2)$
PC7 [Chmeiss and Jégou, 1996]	$O(n^3d^4)$	$O(n^2d^2)$
PC8 [Chmeiss and Jégou, 1998]	$O(n^3d^4)$	$O(n^2d)$
PC2001 [Bessiere et al., 2005]	$O(n^3d^3)$	$O(n^3d^2)$
sDC2 [Lecoutre et al., 2007]	$O(\lambda n^3d^3)$	$O(n^2d^2)$
sDC3 [Lecoutre et al., 2007]	$O(n^3d^4)$	$O(n^2d^2)$

TABLE 1.2 – La complexité dans le pire cas des algorithmes qui établissent la cohérence de chemin pour les réseaux de contraintes binaires

Définition 19. *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN. Nous désignons par $AC(P)$ la* fermeture arc-cohérente *de P, qui consiste à établir AC d'une manière itérative jusqu'à l'obtention d'un point fixe.*

Notons que $AC(P)$ est un CN équivalent à P. Nous rappelons que $AC(P) = \perp$ implique que le CN P n'a pas pas de solution.

Définition 20. *Soient $P = (\mathscr{X}, \mathscr{C})$ un CN, $x \in \mathscr{X}$ une variable et $a \in dom(x)$ une valeur du domaine de x. Nous désignons par $P|_{x=a}$ le CN obtenu en réduisant le domaine de la variable x au singleton a.*

Définition 21 (Cohérence d'arc singleton)**.** *Soit $P = (\mathscr{X}, \mathscr{C})$ un CN. On dit que P est SAC si et seulement si $\forall x \in \mathscr{X}$, $\forall a \in dom(x)$, $AC(P|_{x=a}) \not\equiv \perp$.*

Pour comprendre cette propriété, considérons l'exemple du CN de la figure 1.4. Nous rappelons que les valeurs incompatibles sont reliées entre elles avec une ligne en pointillés.

Il est clair que le CN P de la figure 1.4 est AC. En revanche, ce dernier n'est pas SAC car la fermeture arc-cohérente de $P|_{x=b}$ est équivalente à $\perp$. En effet, la réduction du domaine de la variable x au singleton $\{b\}$ fait perdre à la fois le support de la valeur (z, b) sur la contrainte c_{xz} et le support de la valeur (y, a) sur la contrainte c_{xy}. Comme nous l'avons mentionné avant, une valeur qui n'a pas de support sur une contrainte peut être supprimée du domaine de sa variable. Nous supprimons donc les deux valeurs (y, a) et (z, b). La suppression de la valeur (z, b) fait perdre à son tour le support de la valeur (y, b) sur la contrainte c_{yz}. Après la suppression de la valeur (y, b), le domaine de la variable y devient vide, d'où $AC(P|_{x=b}) = \perp$ ($AC(P|_{x=b})$ est trivialement insatisfaisable). Pour rendre le CN P SAC, il faut supprimer la valeur b du domaine de la variable x. Le CN P' de la figure 1.4 est SAC et équivalent à P.

Comme pour AC et PC, il existe plusieurs algorithmes pour établir la cohérence d'arc singleton SAC. Nous citons par exemple, SAC1 [Debruyne and Bessiere, 1997], SAC2 [Bartak and Erben, 2004], SAC-Opt [Bessiere and Debruyne, 2004], SAC-SDS [Bessiere and Debruyne, 2005] et SAC3 [Lecoutre and Cardon, 2005]. Le tableau 1.3 donne la complexité en temps et en espace dans le cadre des réseaux binaires des différents algorithmes que nous

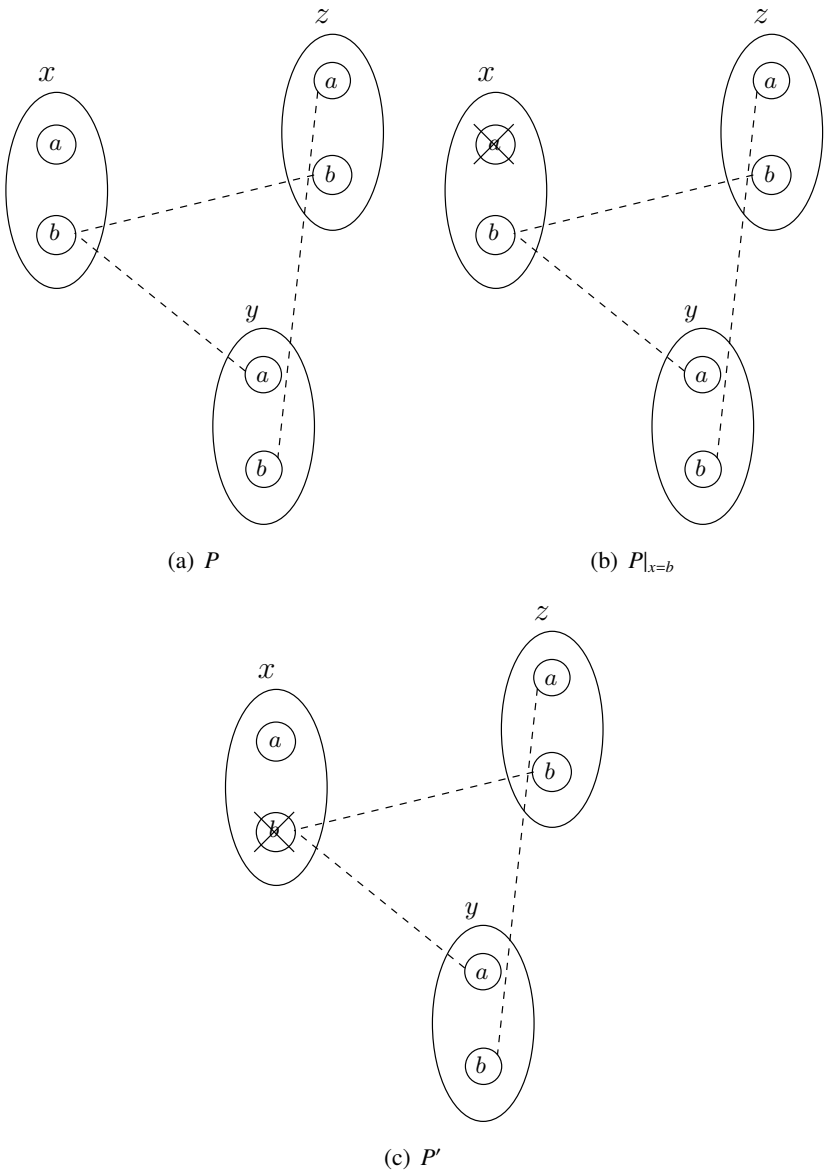


FIGURE 1.4 – La cohérence d'arc singleton sur un réseau de contraintes

avons cités. Pour l'algorithme SAC3-SDS la variable S représente la complexité spatiale de l'algorithme AC utilisé.

1.4 Le maintien de la cohérence d'arc pendant la recherche

Dans la section 1.3, nous avons présenté quelques propriétés de cohérence locale pour le cadre CSP. Pour améliorer l'efficacité de la recherche, ces propriétés peuvent être maintenues au cours de la recherche. L'algorithme 6 présente le maintien de la propriété d'une propriété de cohérence noté ϕ sur un réseau de contraintes P. À chaque étape de la recherche, le réseau

algorithme	complexité en temps	complexité en espace
SAC1 [Debruyne and Bessiere, 1997]	$O(en^2d^4)$	$O(ed)$
SAC2 [Bartak and Erben, 2004]	$O(en^2d^4)$	$O(n^2d^2)$
SAC-Opt [Bessiere and Debruyne, 2004]	$O(end^3)$	$O(end^2)$
SAC-SDS [Bessiere and Debruyne, 2005]	$O(end^4)$	$O(n^2d^2)$
SAC3 [Lecoutre and Cardon, 2005]	$O(en^2d^4)$	$O(ed)$
SAC3-SDS [Bessiere et al., 2011]	$O(end^4)$	$O(n^2d^2 + S)$

TABLE 1.3 – La complexité dans le pire cas des algorithmes qui établissent la cohérence d'arc singleton pour les réseaux de contraintes binaires

de contraintes est filtré avec la cohérence ϕ. Si le réseau obtenu après le filtrage n'est pas incohérent, une nouvelle paire (x, a) est sélectionnée où x est une variable qui n'est pas encore assignée et a une valeur appartenant au domaine de x. La recherche se poursuit sur la branche avec deux hypothèses $x = a$ et $x \neq a$ (réfutation).

Algorithme 6 : $\phi-$solve : Résoudre un CN en maintenant la cohérence ϕ

Entrées : $P_{init} = (\mathscr{X}, \mathscr{C})$: CN
Sorties : Booléen

1 $P = \phi(P_{init})$;
2 **si** $P \equiv \bot$ **alors retourner** $faux$;
3 **si** $\forall x \in \mathscr{X}, |dom(x)| = 1$ **alors retourner** $vrai$;
4 choisir un couple (x, a) avec $|dom(x)| > 1 \wedge a \in dom(x)$;
5 **retourner** (ϕ-$solve(P|_{x=a})$ ou ϕ-$solve(P|_{x\neq a})$) ;

1.5 Heuristiques de recherche

Une heuristique de recherche est une fonction/procédure qui a pour but de guider la recherche vers les espaces les plus prometteurs. Dans la littérature, on note la présence de deux grands types. Le premier est fondé sur le choix de variables. Par exemple, une heuristique de ce type peut correspondre à fixer un ordre sur les variables suivant leur degré (nombre d'apparition dans les contraintes). La seconde est fondée sur le choix de valeurs lors d'une instanciation d'une variable.

1.5.1 Le choix de variables

Nous distinguons trois types d'heuristiques de choix de variables :

1.5.1.1 Heuristique statique (en anglais *SVO* pour Static Variable Ordering)

Dans ce type, l'ordre d'assignation des variables est prédéfini avant le processus de recherche. Les critères d'ordonnancement sont basés sur des informations statiques qui ne dé-

pendant pas du processus de recherche. Par exemple, le dégrée d'une variable qui représente le nombre d'apparition de cette dernière dans l'ensemble des contraintes. Deux heuristiques classiques fondées sur les degrés de variables sont :

- **max degree :** Dans cette heuristique, les variables sont ordonnées selon l'ordre décroissant de leur degré d'apparition dans les contraintes.
- **min degree :** Cette heuristique est construite on inversant le principe de max degree, elle consiste à ordonner les variables selon l'ordre croissant de leurs degrés d'apparition dans les contraintes.

1.5.1.2 Heuristique dynamique (en anglais *DVO* pour Dynamic Variable Ordering)

Dans ce type d'heuristique, le choix de la variable s'effectue en se basant sur l'état courant du réseau. Par exemple, la taille du domaine courant, la dernière variable affectée ou une combinaison entre une information statique et une information sur l'état courant du réseau. Parmi les heuristiques de recherches dynamique, on peut citer par exemple :

- **Domaine et degré (dom+deg) [Frost and Dechter, 1994]** Choisir la variable qui a le plus petit domaine courant. Dans le cas d'égalité (deux variables avec la même taille du domaine), choisir la variable avec le plus grand degré.
- **Domaine sur degré ($\frac{dom}{deg}$) [Bessiere and Régin, 1996]** Choisir la variable qui donne le rapport minimal entre la taille du domaine courant et le degré de la variable.
- **Domaine sur le futur degré ($\frac{dom}{degfut}$) [Bessiere and Régin, 1996]** Choisir la variable qui donne le rapport minimal entre la taille du domaine courant et le futur degré de la variable.

1.5.1.3 Heuristique adaptative (en anglais *AVO* Adaptive Variable Ordering)

Dans ce type d'heuristique, le choix de la variable s'effectue en se basant sur l'état courant du réseau et sur des informations sur les états passés collectées pendant la recherche. Dans cette catégorie, on trouve par exemple :.

- **Degré pondéré(wdeg) [Boussemart et al., 2004a]** Son principe est d'associer à chaque contrainte un poids qu'on incrémente chaque fois que la contrainte est violée pendant la recherche. Les variables liées aux contraintes qui conduisent le plus souvent à des situations de conflit sont les plus prioritaires.
- **Impact [Refalo, 2004]** Cette heuristique se base sur l'impact d'une assignation. Ce dernier est évalué par la réduction moyenne de l'espace de recherche après cette assignation. Au cours de la recherche, la mise à jour de cet impact est nécessaire après chaque assignation.

1.5.2 Le choix de valeurs

Ce type d'heuristique consiste à guider la recherche lors d'une assignation d'une variable pour réduire l'espace de recherche. Dans la littérature, il existe plusieurs heuristiques sur le choix de valeurs. Nous pouvons citer par exemple, l'heuristique de minimisation de conflits [Minton et al., 1992]. Pour expliquer brièvement cette heuristique, supposons qu'une instanciation représente un conflit (viole au moins une contrainte). Dans ce cas, l'heuristique

choisit arbitrairement une variable parmi celles qui sont en conflit. Puis, parmi toute les valeurs possibles pour cette variable, l'heuristique affecte à cette dernière la valeur qui minimise le nombre de conflit.

CHAPITRE 2

Les contraintes souples

Sommaire

Ce chapitre est divisé en trois sections principales décrivant le problème de satisfaction de contraintes valuées, le problème de satisfaction de contraintes pondérées et les différents types de cohérences pour le problème de satisfaction de contraintes pondérées. Dans la première section, nous présentons le formalisme du problème de satisfaction de contraintes valuées (*VCSP*). Nous introduisons d'abord la notion de structure de valuation qui permettra de définir les contraintes souples ainsi que les réseaux de contraintes valuées (*VCN*). Nous examinons brièvement différents formalismes de problèmes qu'on peut présenter à l'aide de la structure de valuation des VCSPs. Enfin, nous présentons un axiome de la structure de valuation que nous utiliserons pour faire des transformations sur des réseaux de contraintes valuées pour mieux faciliter la tâche de résolution.

Dans la seconde section, nous détaillons le cadre de notre thèse qui est le problème de satisfaction de contraintes pondérées (*WCSP*). Nous rappelons la notion de contrainte souple

(fonction de coût) et la manière d'agréger les coûts. Nous établissons le lien existant entre ce formalisme et le formalisme CSP classique. Enfin, nous présentons un algorithme de base dit "séparation et évaluation" qui est utilisé dans plusieurs algorithmes de résolution.

En dernière section, nous présentons les différentes formes de cohérences locales proposées dans la littérature. Pour chaque cohérence, nous illustrons les étapes d'établissement de cette dernière avec un exemple. Enfin, nous étudions la relation existante entre ces dernières.

2.1 Problème de satisfaction de contraintes valuées

Le *problème de satisfaction de contraintes valuées* (en anglais *VCSP* pour Valued Constraint Satisfaction Problem) est un formalisme introduit dans [Schiex et al., 1995], qui permet de modéliser des problèmes d'optimisation sous contraintes. Notons que cette classe de problèmes peut être aussi modélisée avec le formalisme SCSP (Semiring Constraint Satisfaction Problem) [Bistarelli et al., 1995]. Une comparaison entre les deux formalismes (SCSP vs VCSP) est faite dans [Bistarelli et al., 1999]. Dans un premier temps, nous utilisons le formalisme VCSP pour présenter les différentes variantes du problèmes de satisfaction de contraintes souples et aboutir au problème WCSP qui est le cœur de notre étude.

Le formalisme VCSP étend le formalisme CSP classique en introduisant une *structure de valuation* que nous notons $\mathscr{S}$. Cette structure permet d'exprimer une graduation dans la violation des contraintes dites "souples". Autrement dit, elle autorise la violation de certaines contraintes avec une pénalité.

Définition 22 (Structure de valuation)**.** *Une structure de valuation $\mathscr{S}$ est un triplet $(E, \circledast, \geq)$ tel que :*

- *E est un ensemble de valuations totalement ordonné par $\geq$, avec un élément $\top$ qui représente le plus grand élément et un élément $\bot$ qui représente le plus petit élément.*
- *E est soumis à une loi de composition interne binaire $\circledast$ appelée aussi opérateur d'agrégation qui vérifie :*
  - *commutativité : $\forall a, b \in E, a \circledast b = b \circledast a$*
  - *associativité : $\forall a, b, c \in E, (a \circledast b) \circledast c = a \circledast (b \circledast c)$*
  - *monotonie : $\forall a, b, c \in E, a \geq c \Rightarrow a \circledast b \geq c \circledast b$.*
  - *élément absorbant : $\forall a \in E, a \circledast \top = \top$.*
  - *élément neutre : $\forall a \in E, a \circledast \bot = a$.*

Définition 23 (Contrainte souple)**.** *Une contrainte souple c_S associe à chaque tuple t de $l(S)$ une valuation $v \in E$. L'élément v représente la valuation du tuple t sur la contrainte c_S.*

Dans tout ce qui suit, nous utiliserons le terme "contrainte" pour désigner une "contrainte souple". Le problème de satisfaction de contraintes valuées consiste à trouver une instanciation complète de valuation minimale pour un réseau de contraintes valuées :

Définition 24 (Réseau de contraintes valuées)**.** *Un réseau de contraintes valuées (en anglais* VCN *pour Valued Constraint Network) est un triplet $P = (\mathscr{X}, \mathscr{C}, \mathscr{S})$, où $\mathscr{X}$ représente l'ensemble des variables, $\mathscr{C}$ l'ensemble des contraintes souples et $\mathscr{S}$ la structure de valuation.*

La valuation d'une instanciation I d'un sous-ensemble de variables $Y \subseteq \mathscr{X}$ est calculée en combinant à l'aide de $\circledast$ les valuations de toutes les contraintes c_S du réseau couvertes par I.

Définition 25 (Valuation d'une instanciation)**.** *Soient $P = (\mathscr{X}, \mathscr{C}, \mathscr{S})$ un VCN, $Y \subseteq \mathscr{X}$ et I une instanciation de Y. La valuation de I par rapport à P est :*

$$V_P(I) = \circledast_{c_S \in \mathscr{C}, S \subseteq vars(I)} c_S(I[S])$$

Une *solution* pour un réseau de contraintes valuées est une instanciation complète de valuation différente de $\top$. L'objectif du VCSP est de produire une instanciation complète de valuation minimale, c'est ce que nous appelons une *solution optimale*. La définition suivante capture les deux notions de *solution* et *solution optimale*.

Définition 26 (Solution)**.** *Soit $P = (\mathscr{X}, \mathscr{C}, \mathscr{S})$ un VCN. Une solution optimale de P est une instanciation complète I de $\mathscr{X}$ telle que :*

- $V_P(I) \neq \top$ *(I est une solution)*
- $\forall I'$ *instanciation complète de* $\mathscr{X}, V_P(I') \geq V_P(I)$ *(I est optimale)*

Un VCN P peut avoir un ensemble de solutions S mais seul un sous-ensemble de solutions $S^* \subseteq S$ est dit optimal. Si $S = \emptyset$ alors le VCN P est incohérent. Notons que le problème de satisfaction de contraintes valuées est un problème NP-dur [Schiex et al., 1995]. Pour bien comprendre ce formalisme considérons l'exemple suivant :

Exemple 2. *Soit $P = (\mathscr{X}, \mathscr{C}, \mathscr{S})$, tel que :*

- $\mathscr{X} = \{x, y, z\}$ *avec* $dom(x) = dom(y) = dom(z) = \{a, b\}$.
- $\mathscr{C} = \{c_{xy}, c_{yz}, c_{xz}\}$.
- $\mathscr{S} = (\{0, 1, 2, 3\}, \oplus, \geq)$*, tel que* $\bot = 0$*,* $\top = 3$*,* $\forall a, b \in E$*,* $a \oplus b = \min(a + b, \top)$ *et* $\geq$ *est l'ordre usuel sur les entiers naturels.*
- *Les contraintes souples de $\mathscr{C}$ sont définies par les tables suivantes :*

x	y	c_{xy}
a	a	1
a	b	0 ($\bot$)
b	a	3 ($\top$)
b	b	2

y	z	c_{yz}
a	a	0 ($\bot$)
a	b	3 ($\top$)
b	a	3 ($\top$)
b	b	0 ($\bot$)

x	z	c_{xz}
a	a	0 ($\bot$)
a	b	3 ($\top$)
b	a	3 ($\top$)
b	b	0 ($\bot$)

- *La requête est : "Trouver une instanciation complète I de valuation minimale et différente de $\top$"*

Nous supposons que les valeurs sont ordonnées selon l'ordre lexicographique ($a > b$). La contrainte souple c_{xy} est complètement satisfaite uniquement quand $x < y$. En cas d'égalité, on préfère le couple $\{(x, a), (y, a)\}$ par rapport au couple $\{(x, b), (y, b)\}$). Le cas $x > y$ est totalement interdit. En revanche, les deux contraintes c_{yz} et c_{xz} représentent la contrainte d'égalité classique. Notons que la contrainte d'égalité classique est une contrainte dure car elle ne contient que des tuples autorisés et interdits. Un couple de valeurs (v_1, v_2) satisfait totalement les contraintes c_{yz} et c_{xz} si et seulement si $v_1 = v_2$ sinon le couple est rejeté et il ne peut pas apparaître dans une solution. Il est clair que cette instance n'admet pas d'instanciation de valuation

0. Par conséquent, cette instance n'aurait pas de solution dans le formalisme CSP. L'unique solution optimale dans cet exemple est l'instanciation complète $I = \{(x, a), (y, a), (z, a)\}$. En effet, cette instanciation représente une solution car $V_P(I) \neq \top$ et toute autre instanciation complète a une valuation supérieure ou égale à $V_P(I)$.

2.1.1 Classes de VCSP

Le formalisme VCSP nous offre un mécanisme puissant pour la modélisation des différents problèmes sous contraintes d'une manière générale. Dans cette section, nous examinons différents formalismes de problèmes qui existent dans la littérature et nous montrons que grâce à la structure de valuation des VCSPs, nous pouvons exprimer ces types de problèmes en spécifiant l'opérateur de composition $\circledast$, l'ensemble de valuation E et l'ordre $\geq$ utilisé sur E.

- **CSP classique ($\wedge$-VCSP)[Montanari, 1974]** : Dans le chapitre 1, nous avons détaillé ce problème. Nous avons montré qu'une *solution* à un tel problème est une instanciation complète qui satisfait toutes les contraintes. Un CSP classique est équivalent à un VCSP avec une structure $\mathscr{S}$ tel que, $E = \{\bot = vrai, \top = faux\}$, l'opérateur d'agrégation $\circledast = \wedge$ représente le ET logique, la relation d'ordre sur E est $faux \geq vrai$.
  On peut remarquer que, dans le cas CSP, toute solution est une solution optimale.
- **Max-CSP [Freuder and Wallace, 1992]** : Dans un problème *Max-CSP*, chaque contrainte c_S associe à un tuple t de $l(S)$ un coût 0 si t satisfait c_S ou 1 sinon. Une *solution optimale* pour un Max-CSP consiste à trouver une instanciation qui maximise (resp. minimise) le nombre de contraintes satisfaites (resp. violées). La structure de valuation qui correspond à ce type de problème est : $E = \{\bot = 0, 1, 2, \ldots, \top = +\infty\}$, $\circledast = +$, $\geq$ est l'ordre usuel utilisé sur les entiers naturels.
- **CSP additif ou CSP à pénalité ($\sum$-VCSP) [Freuder and Wallace, 1992]** : Ce formalisme permet d'exprimer les préférences sur les contraintes. Il associe à chaque contrainte c_S un entier positif $p(c_S)$, qui représente une pénalité (coût) de la violation de c_S. Pour pouvoir exprimer les contraintes dures dans ce formalisme, la valeur de $p(c_S)$ doit être égale à $+\infty$, dans ce cas la contrainte c_S doit être satisfaite. La valuation d'une instanciation est donnée par la somme des pénalités de contraintes violées par cette instanciation. Une *solution* pour un $\sum$-VCSP est une instanciation complète avec une pénalité différente de $+\infty$. Une *solution optimale* est une solution avec une pénalité minimale. Le problème équivalent à ce formalisme est un VCSP avec la structure $\mathscr{S}$ telle que, $E = \{\bot = 0, 1, 2, \ldots, \top = +\infty\}$, $\circledast = +$ et $\geq$ est l'ordre usuel utilisé sur les entiers naturels.
- **CSP possibiliste (Max-VCSP) [Schiex, 1992a]** : Le formalisme CSP possibiliste associe à chaque contrainte c_S une priorité sous forme d'un réel entre [0,1]. Cette priorité désigne l'importance de satisfaction de la contrainte c_S. Une contrainte avec une priorité égale à 1 est une contrainte dure qui doit être satisfaite. La priorité 0 représente les contraintes les moins importantes dans le problème. L'objectif de ce formalisme est de rechercher l'instanciation qui minimise le maximum des priorités des contraintes violées. Un CSP possibiliste est équivalent à un problème VCSP avec la structure $\mathscr{S}$ telle que, $E = \{[0 = \bot, 1 = \top]\}$, $\circledast = \max$ et l'ordre sur l'ensemble E est $1 \geq 0$.
- **CSP flou [Dubois et al., 1993]** : Dans un CSP flou, chaque contrainte est représentée par

un ensemble de tuples, tel que chaque tuple est annoté avec une valuation réelle $v \in [0, 1]$ qui représente son degré d'acceptation. Quand un tuple est valué à 1, cela revient à dire que ce tuple est totalement autorisé. En revanche, un tuple avec un coût 0 représente un tuple interdit et il ne peut pas apparaître dans une solution. Une *solution* pour un CSP flou est une instanciation complète avec un degré différent de 0. Une *solution optimale* est une solution avec un degré maximal. La structure de valuation correspondante à ce formalisme est $\mathscr{S}$ telle que, $E = \{[0 = \top, 1 = \bot]\}$, $\circledast = \min$ et l'ordre sur l'ensemble E est $0 \geq 1$.

– **CSP lexicographique (*lex*-VCSP) [Fargier et al., 1995]** : Dans le cadre CSP lexicographique, chaque contrainte c_S est annotée avec une priorité sous forme d'un *multi-ensemble* [1] de nombres réels entre 0 et 1 noté $[0, 1]^*$. Quand une contrainte c_S est annotée avec un multi-ensemble qui contient deux fois le nombre a (noté $\{a^2\}$), la contrainte c_S sera aussi importante que deux contraintes annotées avec un seul élément $\{a\}$, strictement plus importante que toute contrainte dont la valuation contient des éléments strictement inférieurs à $\{a\}$ et strictement moins importante que toute contrainte dont la valuation contient des éléments strictement supérieurs à $\{a\}$. La structure de valuation du formalisme CSP lexicographiques est $\mathscr{S}$ tel que, l'ensemble E est formé de l'ensemble des multi-ensembles de nombres réels entre 0 et 1, $\bot = \emptyset$, $\top = \{+\infty\}$, $\circledast = \cup$, et la relation d'ordre sur l'ensemble E est $\geq^*$ définie par :
Définition 27. *Soit E un ensemble ordonné avec l'ordre $\geq$ utilisé dans la structure de valuation du CSP possibiliste. Soit E^* l'ensemble des multi-ensembles des éléments de E. Soient a, b deux multi-ensembles de E^*, m l'élément maximum de $a \cup b$ selon l'ordre $\geq$, et a_m (resp. b_m) le sous-ensemble de a (resp. b) constitué des éléments égaux à m. $a \geq^* b \Leftrightarrow \{|a_m| \geq |b_m|\} \vee \{(|a_m| = |b_m|) \wedge (a - a_m \geq^* b - b_m)\}$. $a - a_m$ contient tous les éléments de a à l'exception de m.*
Pour comprendre cette définition, considérons l'exemple suivant :
Exemple 3. *Soit $\{a, b\} \subset E^*$ tel que $a = \{0.2, 0.3, 0.4^2\}$ et $b = \{0.2, 0.3^2\}$.*
Pour pouvoir comparer les deux multi-ensembles a et b selon l'ordre $\geq^*$, il faut calculer d'abord l'élément maximum m de l'ensemble $a \cup b = \{0.2, 0.3, 0.4\}$ selon l'ordre $\geq$. Ensuite, il faut définir les deux sous-ensemble a_m et b_m. Notons que sur cet exemple $m = 0.4$, $a_{0.4} = \{0.4^2\}$ et $b_{0.4} = \emptyset$. Il est clair que $|a_{0.4}| \geq |b_{0.4}|$, d'où $a \geq^* b$.

– **CSP probabiliste ($\prod$-VCSP) [Fargier and Lang, 1993]** : Dans le cadre CSP probabiliste, chaque contrainte c_S est annotée avec une probabilité $pr(c)$ d'appartenance au CSP. La valuation d'une instanciation I représente la probabilité qu'au moins une contrainte violée par I appartienne au CSP. La valuation de cette instanciation est donnée par $V_P(I) = 1 - \prod_{c \in C} (1 - pr(c))$. Une *solution* est une instanciation complète de coût minimal. La structure de valuation associée à ce problème est $\mathscr{S}$ tel que, $E = [0 = \bot, 1 = \top]$, l'opérateur de combinaison $\circledast$ est défini par $\forall a, b \in E, a \circledast b = 1 - (1 - a).(1 - b)$ et l'ordre $\geq$ sur l'ensemble E est l'ordre utilisé sur les entiers naturels.
– **CSP pondéré (WCSP)[Larrosa, 2002]** : Ce formalisme représente le cadre de notre travail. Il sera abordé en détail dans la section 2.2. D'une manière générale, un *problème de satisfaction de contraintes pondérées* est représenté par un ensemble de contraintes

1. Un ensemble dans lequel un élément peut apparaître plusieurs fois

qui à chaque tuple associe un coût dans $[0..k]$, où k est un paramètre qui représente le seuil à partir duquel les coûts sont jugés prohibitifs. Ce paramètre peut être un entier positif ou $+\infty$. Une *solution* est une instanciation complète avec un coût strictement inférieur à k. Une *solution optimale* est une solution avec un coût minimal. Le cadre WCSP est une instanciation du cadre VCSP avec la structure de valuation $\mathscr{S}$ telle que, $E = \{0 = \bot, 1, \ldots, k = \top\}$, $\circledast = \oplus$ est défini par $\forall a, b \in E, a \oplus b = \min(a + b, k)$ et l'ordre $\geq$ sur l'ensemble E est l'ordre utilisé sur les entiers naturels.

Dans le tableau 2.1 nous avons résumé les différentes classes VCSP. Notons que dans ce tableau les symboles :

- $\overline{N}$ représente l'ensemble des entiers naturels union $+\infty$.
- $\overline{N}^*$ l'ensemble des multi-ensembles de $\overline{N}$.
- $>$ représente l'ordre usuel sur les réels ou les entiers naturels.
- $\geq^*$ l'ordre sur les multi-ensembles.
- id est la propriété d'idempotence donnée par $\forall a \in E, a \circledast a = a$.
- m.stricte : la propriété de monotonie stricte donnée par $\forall a, b, c \in E, [(a \succ b) \wedge (c \neq \top)] \Rightarrow [(a \circledast c) \succ (b \circledast c)]$.

Classe VCSP	**Notation**	E	$\geq$	$\bot$	$\top$	$\circledast$	**Propriétés de $\circledast$**
CSP classique	$\wedge$-VCSP	$\{v, f\}$	$f \geq v$	$\bot = v$	$\top = f$	$\wedge$	id+m.stricte
Max CSP	Max-CSP	$\overline{N}$	$>$	$\bot = 0$	$\top = +\infty$	$+$	m.stricte
CSP additif	$\sum$-VCSP	$\overline{N}$	$>$	$\bot = 0$	$\top = +\infty$	$\sum$	m.stricte
CSP possibiliste	Max-VCSP	[0,1]	$>$	$\bot = 0$	$\top = 1$	max	id
CSP flous	CSP flous	[0,1]	$<$	$\bot = 1$	$\top = 0$	min	id
CSP lexico	*lex*-VCSP	$\overline{N}^*$	$\geq^*$	$\bot = \emptyset$	$\top = \{+\infty\}$	$\cup$	m.stricte
CSP probabiliste	$\prod$-VCSP	[0,1]	$<$	$\bot = 1$	$\top = 0$	1-(1-a).(1-b)	m.stricte
CSP pondérée	WCSP	[0,k]	$>$	$\bot = 0$	$\top = k$	$\min(a+b, k)$	-

TABLE 2.1 – Classes VCSP et les structures associées

Il est à noter que dans le cas WCSP, la structure de valuation $\mathscr{S}(k)$ vérifie la propriété de monotonie stricte seulement quand $k = +\infty$. Nous verrons dans la section 2.2.3 que le problème CSP peut être défini comme le problème WCSP sur $\mathscr{S}(1)$. Dans ce cas, la structure de valuation $\mathscr{S}(1)$ est équivalente à celle des CSPs classiques et vérifie les mêmes propriétés de l'idempotence et de la monotonie stricte.

2.1.2 Structure de valuation juste

Dans certains cas, la résolution d'un VCN nécessite la transformation de ce problème vers un autre problème équivalent et plus facile résoudre. D'une manière générale, nous disons que deux VCNs sont équivalents si et seulement s'ils ont le même ensemble de variables, et ils ont la même valuation pour chaque instanciation complète. Nous verrons plus en détail l'équivalence des problèmes dans le cadre WCSP dans le chapitre 2.2. Pour pouvoir faire ces transformations, Schiex et al. [Cooper and Schiex, 2004] ont défini un nouvel axiome.

Définition 28 (Différence entre valuations)**.** *Soient $\mathscr{S} = (E, \oplus, \geq)$ une structure de valuation et $u, v \in E$ avec $u \geq v$. une valuation w est appelée la différence entre u et v si $v \oplus w = u$.*

Définition 29 (Structure de valuation juste)**.** *Une structure de valuation $\mathscr{S}$ est dite "juste" (en anglais fair) si $\forall u, v \in E$ avec $u \geq v$, il existe une différence unique et maximale $w \in E$ notée $u \ominus v$.*

La majorité des structures de valuation liées aux VCSPs satisfont cet axiome. Dans ce qui suit, nous étudions cet axiome sur certains formalismes VCSP.

– La structure de valuation d'un CSP classique ($\wedge$-VCSP) satisfait cet axiome. Nous rappelons que dans cette structure $E = \{vrai = \bot, faux = \top\}$, $\oplus = \wedge$ et l'ordre $faux \geq vrai$. En effet, pour $a \ominus b = max(a, b)$, $\forall u, v \in E$ avec $u \geq v$, $\exists w \in E$ tel que $u \ominus v = w$. Ce qui permet de conclure que cette structure est juste. L'opérateur $\ominus$ pour cette structure est donné par la table suivante :

u	v	$u \ominus v$
$vrai$	$vrai$	$vrai$
$faux$	$vrai$	$faux$
$faux$	$faux$	$faux$
$vrai$	$faux$	non défini

Remarque 2. *Le cas $u \ominus v$ avec $u = vrai$ et $v = faux$ n'intervient pas pour déterminer si la structure de valuation du VCSP est juste car la définition impose que $u \geq v$.*

– Dans le cadre Max-CSP, nous rappelons que sa structure $S = (E, \oplus, \geq)$ où $E = N \cup \{+\infty\}$, $\oplus = +$ et $\geq$ représente l'ordre usuel sur les entiers naturels. L'axiome défini ci-dessus est satisfait pour l'opérateur $\ominus$ qui est défini comme suit :

$$u \ominus v = \begin{cases} u - v : u \neq +\infty \\ +\infty : u = +\infty \end{cases}$$

– La structure $\mathscr{S}(k)$ du cadre WCSP vérifie aussi cet axiome. La définition de l'opérateur $\ominus$ pour cette structure est la suivante :

$$u \ominus v = \begin{cases} u - v : u \neq k \\ k : u = k \end{cases}$$

Cette définition spécifie $k \ominus k = k$. Cette convention ne convient pas nécessairement dans tous les calculs. Nous abordons avec plus de détail ce problème dans le chapitre "la substituabilité souple au voisinage dans le cadre WCSP".

2.2 Problème de satisfaction de contraintes pondérées

Le *problème de satisfaction de contraintes pondérées* (en anglais *WCSP* pour Weighted Constraint Satisfaction Problem) est un problème d'optimisation introduit dans [Larrosa, 2002]. Dans cette section, nous montrons que le formalisme WCSP est une spécialisation du formalisme VCSP et une extension du cadre CSP classique et nous décrivons un algorithme de base pour résoudre ce type de problème. Enfin, nous examinons et nous établissons des relations entre les différentes cohérences souples (NC*, AC*, DAC^* ... etc.).

En section 2.1, nous avons présenté le formalisme VCSP comme étant une généralisation du cadre CSP qui permet de définir différents types de contraintes souples sur la base de différentes structures de valuation. Parmi les différentes spécialisations de VCSP se trouve le problème de satisfaction de contraintes pondérées (WCSP) qui est défini par la structure de valuation suivante :

Définition 30 (Structure de valuation du formalisme WCSP)**.** *La structure de valuation du formalisme WCSP est une structure $\mathscr{S}(k)$ paramétrée par k qui est soit un entier strictement positif, soit $+\infty$. $\mathscr{S}(k)$ est le triplet $(E, \oplus, \geq)$ défini par :*

- $E = [0..k]$ *lorsque* $k \neq +\infty$ *et* $E = \mathbb{N} \cup \{+\infty\}$ *quand* $k = +\infty$*. De ce fait,* $\bot = 0$ *et* $\top = k$*.*
- $\oplus$ *l'opérateur d'agrégation sur E défini par :*

$$a \oplus b = \min(a + b, k)$$

- $\geq$ *est l'ordre standard sur les entiers naturels*

La définition formelle d'un WCSP est la suivante :

Définition 31 (Problème de satisfaction de contraintes pondérées)**.** *Le problème de satisfaction de contraintes pondérées (en anglais* WCSP *pour Weighted Constraint Satisfaction Problem) est le problème VCSP défini sur la structure de valuation $\mathscr{S}(k)$.*

Pour le cadre WCSP, nous utiliserons la notation $\mathscr{W}$ pour désigner l'ensemble des fonctions de coût w_s (contraintes souples). Le problème WCSP consiste à trouver une instanciation de coût minimale pour un réseau de contraintes pondérées :

Définition 32 (Réseau de contraintes pondérées)**.** *Un réseau de contraintes pondérées (en anglais* WCN *pour Weighted Constraint Network) est défini par un triplet $P = (\mathscr{X}, \mathscr{W}, k)$, où $\mathscr{X}$ est l'ensemble des variables, $\mathscr{W}$ l'ensemble des fonctions de coût (contraintes) et k est le coût $\top$ (le coût interdit).*

2.2.1 Fonctions de coût (contraintes souples)

Comme dans le cadre CSP classique, chaque contrainte $w_S \in \mathscr{W}$ porte sur un ensemble fini de variables $S \subseteq \mathscr{X}$. L'arité de la contrainte w_S est $|S|$. La contrainte w_S représente une fonction de coût qui associe à chaque tuple t de $l(S)$ un coût entier entre 0 et k.

$$w_S : l(S) \rightarrow [0..k]$$

Quand w_S assigne un coût k à un tuple t de $l(S)$, cela veut dire que t est interdit, sinon il est autorisé avec le coût correspondant. Notons qu'un WCN est dit “binaire" si et seulement si chaque fonction de coût a une arité inférieure ou égale à 2 et comporte au moins une fonction de coût binaire.

On peut définir une partition de l'ensemble des fonctions de coût comme suit :

- **La fonction de coût d'arité nulle :** Comme son nom l'indique, cette fonction de coût d'arité 0 ne porte sur aucune variable. Elle représente une constante dans la fonction de coût globale. La contrainte d'arité nulle est notée $w_\emptyset$. Sans perte de généralité, on peut supposer que chaque WCN contient une fonction $w_\emptyset$ (éventuellement de coût 0).

- **Les fonctions de coût unaires :** Ce sont les fonctions de coût d'arité 1, qui portent donc sur une seule variable. Sans perte de généralité, on peut supposer que chaque WCN contient une fonction de coût unaire w_x pour chaque variable x du réseau (éventuellement associant le coût 0 à chaque valeur de la variable ($\forall a \in dom(x), w_x(a) = 0$)).
- **Les fonctions de coût binaires :** C'est l'ensemble des fonctions de coût d'arité deux.
- **Les fonctions de coût n-aires :** C'est l'ensemble des fonctions de coût d'arité strictement supérieure à deux.

Remarque 3. *L'ensemble des fonctions de coût $\mathscr{W}$ d'un WCN est décomposé en sous-ensembles $\mathscr{W} = W_0 \cup W_1 \cup W_2 \cup \cdots \cup W_r$ (où W_i représente l'ensemble des contraintes d'arité i et r représente l'arité maximale des contraintes du problème).*

Les fonctions de coût peuvent être représentées en extension sous forme de tables de coût ou sous forme de graphes. Elles peuvent aussi être représentées en intension sous forme de fonction mathématique. Pour des raisons de simplicité, nous représentons nos exemples dans cette thèse en extension.

Sur l'exemple 4, nous avons exprimé la contrainte binaire w_{xy} de deux manières différentes :

1. **sous forme d'une table de coûts :** en listant explicitement l'ensemble des tuples possibles et en associant à chaque tuple son coût correspondant.

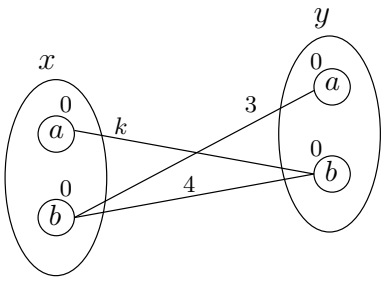
2. **sous forme d'un graphe de coûts :** où l'ensemble des sommets du graphe représente les différents couples (*variable*, *valeur*) tel que chaque couple est annoté avec son coût unaire correspondant. Les sommets sont reliés entre eux avec une arête de sorte que chaque arête est annotée avec le coût du tuple qu'elle représente. Pour assurer la lisibilité des graphes, quand un tuple t a un coût égal à 0, les sommets qui constituent t ne seront pas reliées avec une arête (comme dans le cas du tuple $\{(x, 0), (y, 0)\}$).

Notons que sur cet exemple le tuple $\{(x, 0), (y, 0)\}$ satisfait totalement la contrainte w_{xy}, c'est un tuple totalement autorisé. En revanche, le tuple $\{(x, 0), (y, 1)\}$ a un coût égal à k, il est donc interdit et il ne peut apparaître dans aucune solution.

Exemple 4. *Soit w_{xy} une fonction de coût binaire portant sur les variables x et y. w_{xy} peut se représenter sous forme de table de coût comme suit :*

x	y	w_{xy}
a	a	0
a	b	k
b	a	3
b	b	4

De manière équivalente, w_{xy} peut se représenter sous forme de graphe :



L'exemple 5 montre la façon de représenter les contraintes ternaires sous forme d'un graphe de coûts. Cependant, la représentation graphique des contraintes n-aires s'avère compliquées lorsque nous utilisons des contraintes portant sur un grand nombre de tuples. C'est pourquoi nous adoptons dans tout ce qui suit la représentation par tables de coût pour les contraintes n-aires.

Exemple 5. *Une fonction de coût ternaire représentée sous forme d'hypergraphe :*

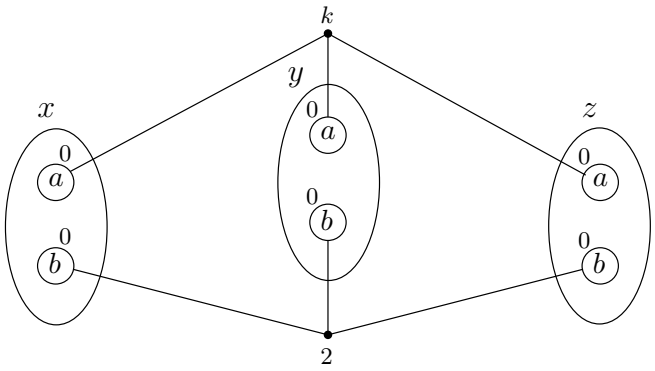


FIGURE 2.1 – Exemple de graphe d'une fonction de coût ternaire

L'exemple de la contrainte ternaire ci-dessus se présente sous forme de table de coût comme suit :

x	y	z	w_{xyz}
a	a	a	k
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	2

2.2.2 Calcul de coût

Dans le cadre WCSP, les coûts se combinent en utilisant l'opérateur d'agrégation $\oplus$ de la structure $\mathscr{S}(k)$. La requête classique pour le cadre WCSP consiste à produire une solution optimale (de coût minimal et strictement inférieur à k) ou à prouver qu'il n'en existe pas. Dans ce qui suit, nous introduisons le calcul de coût sur des instanciations quelconques, ensuite nous donnons les définitions de solution et solution optimale pour un problème WCSP.

Définition 33 (Coût d'une instanciation)**.** *Soient $P = (\mathscr{X}, \mathscr{W}, k)$ un WCN, et I une instanciation complète ($I \in l(\mathscr{X})$). Le coût de l'instanciation I noté $cost(I)$ est donné par :*

$$cost(I) = \bigoplus_{w_S \in \mathscr{W}} w_S(I)$$

Dans le cas d'une instanciation partielle I sur un ensemble $Y \subseteq \mathscr{X}$ ($I \in l(Y)$), on peut définir le coût courant de cette instanciation par :

$$cost(I) = \bigoplus_{\substack{w_S \in \mathscr{W}, \\ S \subseteq Y}} w_S(I)$$

Définition 34 (Solution)**.** *Une solution optimale d'un WCN P est une instanciation complète I de P telle que :*

- *$cost(I) \neq k$* *(I est une solution)*
- *pour toute instanciation complète I' de P, $cost(I') \geq cost(I)$* *(I est optimal)*

2.2.3 Lien entre CSP et WCSP

On peut remarquer que le formalisme WCSP est une extension du formalisme CSP classique. En effet, le problème CSP peut être défini comme le problème WCSP sur $\mathscr{S}(1)$. Ci-dessous, nous établissons le lien existant entre la structure $\wedge$-CSP du cadre CSP et la structure $\mathscr{S}(k)$, et en déduisons quelques résultats.

Rappelons que $\mathscr{S}(1)$ est la structure de valuation $E = \{0 = \bot, 1 = \top\}$. Les tuples sont donc soit autorisés, soit interdits.

Dans le tableau 2.1 nous avons indiqué que l'opérateur d'agrégation $\wedge$ associé à la structure de valuation de la classe $\wedge$-CSP vérifie les deux propriétés d'idempotence et de la monotonie stricte. En revanche, l'opérateur d'agrégation $\oplus$ de la structure $\mathscr{S}(k)$ ne vérifie que la propriété de monotonie stricte dans le cas général. Cependant, quand l'ensemble $E = \{0, 1\}$, la propriété d'idempotence est vérifiée dans ce cas. En effet, $\forall a \in E, a \oplus a = a$. De plus, dans la structure de valuation $\mathscr{S}(1)$, l'opérateur $\oplus$ est équivalent à l'opérateur $\wedge$. Ceci est montré dans la table suivante (nous assumons que $0 = vrai$ et $1 = faux$) :

Il en résulte que le cadre WCSP est plus général que le cadre CSP classique. Et donc, tout CN peut être vu comme un WCN particulier.

Dans ce qui suit, nous présentons un algorithme de base utilisé pour résoudre des problèmes d'optimisations notamment ceux correspondant à des WCNs.

		WCSP	**CSP**
a	b	$a \oplus b$	$a \wedge b$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

TABLE 2.2 – Comparaison des opérateurs $\oplus$ et $\wedge$

2.2.4 Séparation et évaluation (B & B)

Comme les domaines sont finis, l'ensemble des instanciations pour les problèmes d'optimisation combinatoire est fini. Il est donc possible d'énumérer toutes les instanciations, et ensuite choisir celle qui satisfait au mieux nos critères de recherche. L'inconvénient majeur de cette approche est le temps de calcul utilisé. En effet, la méthode énumère toutes les instanciations possibles du problème sans éliminer celles qui trivialement ne peuvent pas apparaître dans une solution. La procédure par séparation et évaluation(en anglais *B & B* pour Branch and Bound) est l'une des méthodes les plus utilisées pour la résolution de problèmes d'optimisation [Lawler and Wood, 1966]. Il s'agit d'énumérer l'ensemble des solutions d'une manière intelligente en éliminant les instanciations partielles qui ne mènent pas à la solution finale que l'on recherche. Pour ce faire, cette méthode détermine une borne inférieure (en anglais *lb* pour Lower Bound) qui représente le coût de l'instanciation courante plus, un minorant du coût des instanciations qui étendent l'instanciation courante. Quand ce coût dépasse le coût de la meilleure solution connue (en anglais *ub* pour Upper Bound), il est inutile de poursuivre la recherche en étendant l'instanciation courante. L'approche B & B reste bien sûr exponentielle, mais la complexité en moyenne est bien plus faible que pour une énumération complète. Notons que dans le pire des cas, on retombe sur l'énumération explicite de toutes les instanciations du problème mais un bon algorithme par séparation et évaluation est normalement capable de détecter et éliminer les mauvaises instanciations partielles. Dans ce qui suit, nous détaillons l'algorithme général de cette méthode et nous montrons sur un exemple le bénéfice de l'utilisation de cette approche.

La méthode branch-and-bound est représentée par un arbre de recherche, où chaque branche de l'arbre constitue une instanciation partielle du problème. Pour pouvoir appliquer cet algorithme sur les problèmes de minimisation (le cadre de notre travail), on doit disposer d'un moyen de calcul d'une borne inférieure pour la solution partielle, d'une technique de décomposition du problème en plusieurs sous-problèmes plus petit pour réduire l'espace de recherche, et enfin d'une borne supérieure (en anglais *ub* pour Upper Bound) qui ne doit pas être dépassée par la solution trouvée. Dans les méthodes par séparation et évaluation (branch and bound), la *séparation* permet de décomposer le problème en sous-problèmes plus faciles à résoudre, tandis que *l'évaluation* évite l'énumération systématique de toutes les solutions.

– **Séparation :** c'est la décomposition d'un problème P en plusieurs sous-problèmes $\{P_1, \ldots, P_m\}$ dans le but de réduire l'espace de recherche. Chaque sous-problème P_i constituent une partie de l'ensemble des solutions réalisables de P. La meilleure solu-

tion du problème original P est obtenue après la résolution de tous ses sous-problèmes, ce principe est appliqué de manière récursive sur chacun des sous-ensembles contenant plusieurs solutions. La séparation d'un problème donné se fait à l'aide d'un arbre de recherche (arbre de décision) tel que la racine de l'arbre représente le problème de départ, chaque nœud autre que la racine représente un sous-problème et les feuilles représentent soit une solution réalisable, soit un échec (une instanciation partielle qui ne peut pas mener à une solution réalisable).

– **Évaluation :** dans un problème de minimisation, l'évaluation d'un nœud de l'arbre de recherche consiste à trouver une borne inférieure de l'ensemble des instanciations associées au sous-arbre de ce nœud. Si la borne inférieure est supérieure au coût de la meilleure solution connue, on en conclut alors que le sous-arbre de ce nœud ne peut contenir de meilleure solution, sinon le processus de recherche continue à partir de ce nœud. Si une nouvelle solution apparaît, elle sera retenue pour la suite de la recherche et son coût utilisé comme une borne supérieure.

Pour bien comprendre le principe de cette méthode, considérons l'exemple du WCN présenté sur la figure 2.2. Pour résoudre ce WCN avec la méthode branch and bound, nous commençons

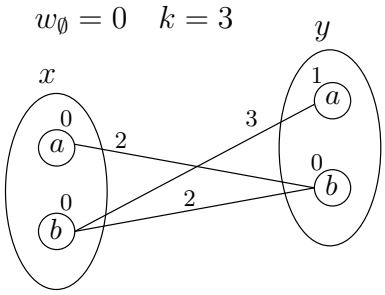


FIGURE 2.2 – Exemple d'un WCN

par définir le coût de la borne supérieure. Pour cet exemple, toute solution doit être strictement inférieure au coût interdit qui est de 3. D'où, une première borne supérieure de 3. Une fois que la borne supérieure est définie, nous décomposons d'une manière récursive le problème en plusieurs sous-problèmes. Nous commençons par décomposer le problème sur la variable x, cette opération permet d'obtenir deux nouveaux sous-problèmes $P|_{x=a}$ et $P|_{x=b}$ où $P|_{x=v}$ est le WCN P quand nous réduisons le domaine de la variable x au singleton v. La même opération est refaite pour la variable y sur les deux sous-problèmes $P|_{x=a}$ et $P|_{x=b}$. Chaque nœud de l'arbre de recherche est évalué en calculant une borne inférieure à l'ensemble des solutions réalisables associées à ce nœud. Une manière simple pour calculer cette borne est de considérer le coût de l'instanciation courante plus le coût minimal de tous les tuples qui étend cette dernière. Le premier nœud $P|_{x=a}$ a une borne inférieure de 0 ($lb = w_x(a) \oplus \min_{t \in l(xy), t[x]=a} w_{xy}(t)$), le second nœud $P|_{x=a,y=a}$ a un coût égal à 1. La première branche $P|_{x=a,y=a}$ va constituer une première solution du problème P et son coût de 1 représente une nouvelle borne supérieure. La seconde branche $P|_{x=a,y=b}$ a un coût égal à 2, elle représente un échec de l'arbre de recherche car son coût est strictement supérieur à la borne supérieure. L'exploration du nœud $P|_{x=b}$ est inutile car l'évaluation de ce nœud donne une borne inférieure de 2 (pour tout tuple $t \in l(\{x, y\})$ tel

que $t[x] = b$, le coût de t sur la contrainte w_{xy} est au moins égal à 2) ce qui dépasse notre borne supérieure. L'arbre de recherche de ce WCN est donné par la figure 2.3. Dans cet exemple,

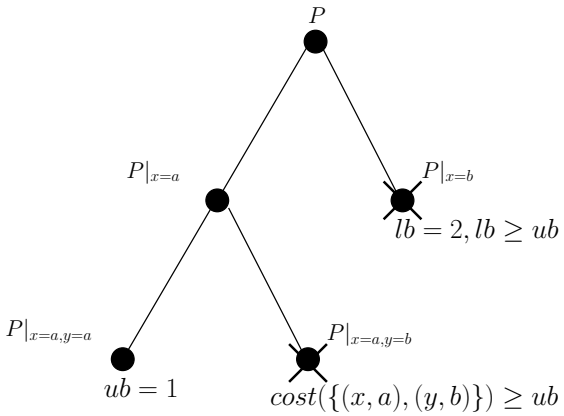


FIGURE 2.3 – L'arbre de recherche du WCN de la figure 2.2

nous avons présenté une méthode naïve pour calculer les bornes inférieures. Dans la section suivante, nous présentons des propriétés dites de *cohérences souples* qui permettent de calculer des bornes inférieures plus précises avec des algorithmes plus sophistiqués.

2.3 Types de cohérences pour WCSP

Durant ces dernières années, des bornes inférieures basées sur l'établissement de cohérences locales, connues aussi sous l'appellation d'arc-cohérence souple (en anglais soft arc-consistency) pour le cadre WCSP, ont été définies. Ces cohérences locales sont établies via l'application répétée de *transformations préservant l'équivalence* (en anglais *EPT* pour Equivalence Preserving Transformations [Cooper and Schiex, 2004]) qui étendent les opérations de cohérences locale utilisées dans le cadre CSP classique. Dans cette section, nous définissons clairement la notion d'EPT. Puis nous examinons différents types de cohérences locales qui existent dans la littérature. Pour chaque type, nous détaillerons les opérations de transformation avec un petit exemple illustratif.

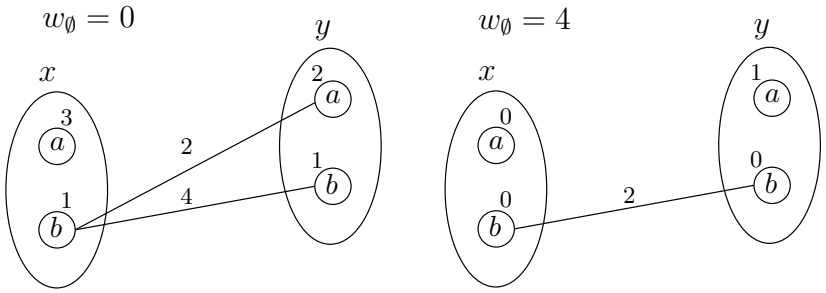
2.3.1 Les transformations préservant l'équivalence (EPT)

Définition 35. *[Équivalence entre WCNs] Soient $P = (\mathscr{X}, \mathscr{W}, k)$ et $P' = (\mathscr{X}', \mathscr{W}', k')$ deux WCNs. L'équivalence entre P et P′ est vérifiée si et seulement si $\mathscr{X} = \mathscr{X}'$ et pour toute instanciation complète I on a :*

$$(cost_P(I) = k \wedge cost_{P'}(I) = k') \vee (cost_P(I) \neq k \wedge cost_{P'}(I) \neq k' \wedge cost_P(I) = cost_{P'}(I))$$

Cette définition montre que l'équivalence entre les WCNs se base sur la distribution de coût sur l'ensemble des instanciations complètes et non pas sur les fonctions de coûts individuelles. Pour illustrer cette définition, considérons l'exemple des deux WCNs P et P' de la

figure 2.4. Nous remarquons que l'ensemble des variables et leur domaine associé est identique pour les deux WCNs. En revanche, la distribution des fonctions de coût de P est différente de celle de P'. Le coût des instanciations complètes ainsi que le coût des contraintes sur ces instanciations sont donnés par la table 2.3.



(a) P (b) P'

FIGURE 2.4 – Équivalence entre WCNs

l'instanciation complète I	$w_\emptyset$	$w_x(I)$	$w_y(I)$	$w_{xy}(I)$	$cost(I)$	$w_\emptyset$	$w_x(I)$	$w_y(I)$	$w_{xy}(I)$	$cost(I)$
$\{(x,a),(y,a)\}$	0	3	2	0	**5**	4	0	1	0	**5**
$\{(x,a),(y,b)\}$	0	3	1	0	**4**	4	0	0	0	**4**
$\{(x,b),(y,a)\}$	0	1	2	2	**5**	4	0	1	0	**5**
$\{(x,b),(y,b)\}$	0	1	1	4	**6**	4	0	0	2	**6**
	les contraintes du WCN P					**les contraintes du WCN P'**				

TABLE 2.3 – Comparaison de la distribution des fonctions de coût pour les deux WCNs P et P'

À partir du tableau 2.3, nous remarquons que le coût des instanciations complètes est identique pour P et P', ce qui permet de conclure à l'équivalence entre ces deux WCNs. Pour passer de P vers P' ou inversement, il faut appliquer des transformations préservant l'équivalence (EPTs). Ces EPTs déplacent les coûts entre les fonctions de coût de portées différentes afin généralement d'obtenir un nouveau WCN équivalent avec une borne inférieure plus grande que celle du WCN original. Notons que ce nouveau WCN obtenu est plus facile à résoudre car la nouvelle borne inférieure va permettre de couper des branches dans des niveaux plus haut de l'arbre de recherche. Pour déplacer les coûts, il faut être capable de les soustraire de leurs origines. Cela se fait à l'aide de l'opérateur $\ominus$ défini comme suit :

$$\forall u, v \in E, u \geq v : u \ominus v = \begin{cases} u - v : u \neq k \\ k : u = k \end{cases}$$

Comme nous l'avons mentionné dans la section 2.1, la structure de valuation $\mathscr{S}(k)$ est juste ; aussi cet opérateur $\ominus$ est-il bien défini. Donc, nous avons la garantie que pour chaque paire de valuations $u, v \in E$, il existe une valuation unique $w \in E$ qui représente la plus grande différence entre u et v. Cette différence est calculée à l'aide de l'opérateur $\ominus$ défini ci-dessus.

Dans ce qui suit, nous présentons trois EPTs élémentaires dont nous aurons besoin tout au long de ce mémoire. Ces EPTs sont la projection (en anglais `Project`), la projection unaire (en anglais `UnaryProject`) et l'extension (en anglais `Extend`). Elles sont utilisées comme des opérations de base par plusieurs algorithmes qui utilisent le principe de transformation d'un réseau WCN en un autre réseau WCN équivalent et plus facile à résoudre.

Définition 36 (Projection)**.** *Soient $w_S \in \mathscr{W}$ une contrainte, $x \in S$ une variable qui est dans la portée de w_S et a une valeur de $dom(x)$. La projection d'un coût $0 < \alpha \leq \min_{t \in l(S), t[x]=a} w_S(t)$ depuis les tuples $t \in l(S)$ tels que $t[x] = a$ vers la valeur (x, a) consiste à :*

- *Ajouter α à $w_x(a)$, à l'aide de l'opérateur d'agrégation $\oplus$ de la structure $\mathscr{S}(k)$.*
- *Soustraire α de $w_S(t)$ pour tout $t \in l(S)$ tel que $t[x] = a$, à l'aide de l'opérateur $\ominus$.*

Les étapes de la projection d'un coût α depuis les tuples de la contrainte w_S vers la valeur a de la variable x sont données par l'algorithme 7. Rappelons que $\ominus$ est l'opérateur qui nous a permis de montrer que la structure $\mathscr{S}(k)$ est juste. D'où la condition $\alpha \leq \min_{t \in l(S), t[x]=a} w_S(t)$. En effet, pour que l'opération de ligne 4 soit valide selon l'axiome de structure de valuation juste pour les VCSPs, α doit être inférieur ou égal à $w_S(t)$.

Algorithme 7 : `Project`

Entrées : w_S : Contrainte, x : Variable, a : Valeur, α : Coût

1 **pré-condition :** $0 < \alpha \leq \min_{t \in l(S), t[x]=a} w_S(t)$;
2 $w_x(a) \leftarrow w_x(a) \oplus \alpha$;
3 **pour chaque** $t \in l(S)$ *t.q.* $t[x] = a$ **faire**
4 $\quad w_S(t) \leftarrow w_S(t) \ominus \alpha$;

L'exemple de la figure 2.5 montre les étapes de `Project` sur une contrainte binaire. Sur cet exemple nous déplaçons un coût $\alpha = 3$ depuis les tuples t de la contrainte binaire w_{xy} tel que $t[x] = a$ vers $w_x(a)$. Notons que sur cet exemple $\alpha = \min_{t \in l(S), t[x]=a} w_S(t)$, par conséquent, la ligne 1 de l'algorithme est vérifiée. Le WCN P' obtenu après ces transferts est équivalent à P.

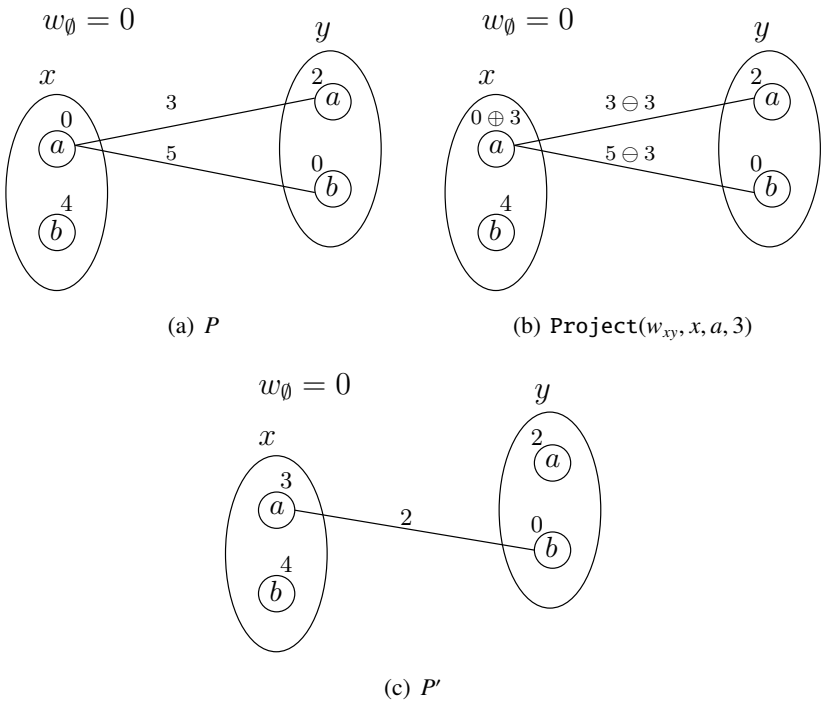
Après cette transformation, nous remarquons que toutes les valeurs de la variable x ont un coût supérieur ou égal à 3. Cet remarque va permettre d'augmenter la borne inférieure $w_\emptyset$ de 3. Cela peut se faire à l'aide d'une nouvelle EPT `UnaryProject` définie comme suit :

Définition 37 (Projection unaire)**.** *Soit $w_x \in W$ une contrainte unaire. La projection unaire d'un coût $0 < \alpha \leq \min_{a \in dom(x)} w_x(a)$ depuis les valeurs de la variable x vers la contrainte d'arité nulle $w_\emptyset$ consiste à :*

- *Ajouter α à $w_\emptyset$, à l'aide de l'opérateur d'agrégation $\oplus$ de la structure $\mathscr{S}(k)$.*
- *Soustraire α de $w_x(a)$ tel que $a \in dom(x)$, à l'aide de l'opérateur $\ominus$.*

Les étapes de la projection unaire d'un coût α depuis la contrainte unaire w_x sont données par l'algorithme 8.

En figure 2.6, nous avons repris le WCN P' de la figure 2.5. Sur cet exemple, nous avons déplacé un coût égal à 3 depuis la contrainte unaire w_x vers la contrainte d'arité nulle $w_\emptyset$.



(a) P

(b) $\texttt{Project}(w_{xy}, x, a, 3)$

(c) P'

FIGURE 2.5 – Illustration de l'EPT `Project` sur une contrainte binaire

Algorithme 8 : `UnaryProject`

Entrées : w_x : Contrainte, α : Coût

1 **pré-condition :** $0 < \alpha \leq \min_{a \in dom(x)} w_x(a)$;

2 $w_\emptyset \leftarrow w_\emptyset \oplus \alpha$;

3 **pour chaque** $a \in dom(x)$ **faire**

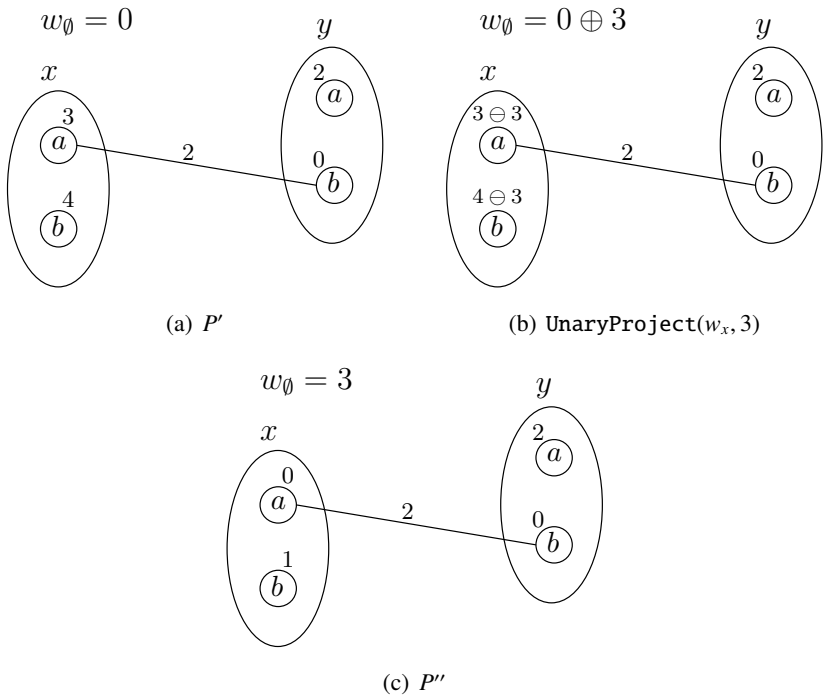
4 $\quad w_x(a) \leftarrow w_x(a) \ominus \alpha$;

Notons que d'un point de vue pratique, le WCN P'' de la figure 2.6 est préférable à P et P' de la figure 2.5. En effet, la borne inférieure $w_\emptyset$ du WCN P'' est supérieure à celle de P et P'. Comme $w_\emptyset$ est un minorant du coût des instanciations à venir, cette borne peut être exploitée pour obtenir la borne lb et couper des branches dans l'arbre de recherche Branch and Bound (comme nous l'avons mentionné dans la sous-section 2.2.4).

La dernière opération est l'extension (`Extend`). La définition de cette EPT est capturée par la définition suivante :

Définition 38 (Extension)**.** *Soient w_x une contrainte unaire et w_S une contrainte telle que $x \in S$ et $|S| \geq 2$. L'extension d'un coût $0 < \alpha \leq w_x(a)$ depuis la valeur (x, a) vers les tuples de la contrainte w_S tel que $x \in S$ consiste à :*

- *Ajouter α à $w_S(t)$ pour tout tuple $t \in l(S)$ tel que $t[x] = a$, à l'aide de l'opérateur d'agrégation $\oplus$ de la structure $\mathcal{S}(k)$.*
- *Soustraire α de $w_x(a)$, à l'aide de l'opérateur $\ominus$.*



(a) P' (b) `UnaryProject`$(w_x, 3)$

(c) P''

FIGURE 2.6 – Illustration de l'EPT `UnaryProject`

L'EPT `Extend` effectue l'opération inverse de la projection en transférant un coût α depuis une valeur d'une variable donnée vers les tuples d'une contrainte qui implique cette variable comme le montre l'algorithme 9.

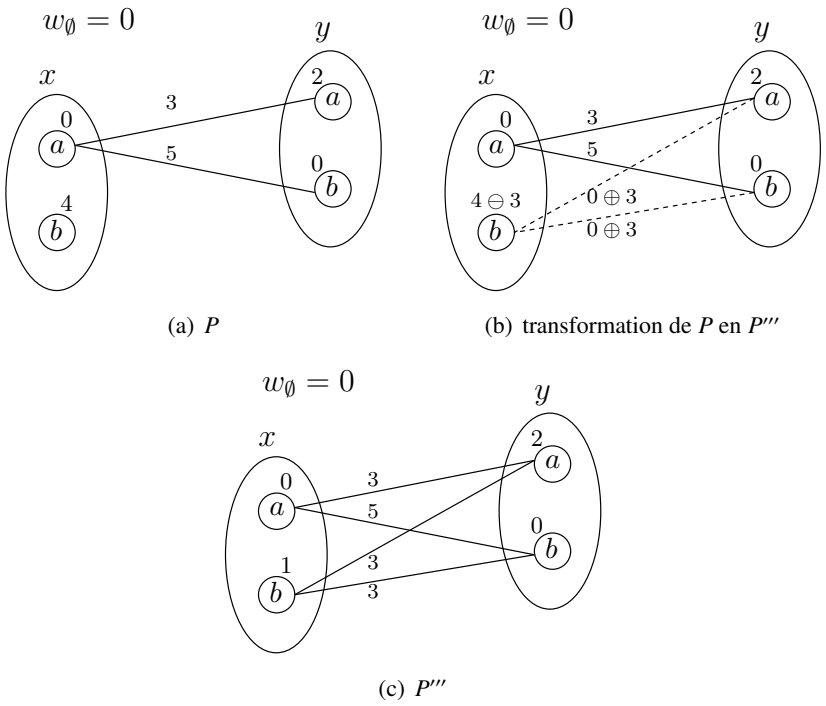
Algorithme 9 : `Extend`

Entrées : w_x : Contrainte, a : Variable, W_S : Contrainte, α : Coût

1 **pré-condition :** $0 < \alpha \leq w_x(a)$;
2 $w_x(a) \leftarrow w_x(a) \ominus \alpha$;
3 **pour chaque** $t \in l(S)$ *t.q.* $t[x] = a$ **faire**
4 $\quad w_S(t) \leftarrow w_S(t) \oplus \alpha$;

Reprenons l'exemple du WCN P de la figure 2.5. Pour montrer l'EPT `Extend`, nous transférons un coût égal à 3 de la valuation $w_x(b)$ vers les valuations $w_{xy}(t)$ où $t \in l(\{x, y\})$ tel que $t[x] = b$. Le résultat de cette transformation est illustré par la figure 2.7

Notons que le WCN P''' obtenu est équivalent à P, P' et P''. Dans ce qui suit, nous présentons les différentes cohérences souples et nous montrons qu'elles peuvent être obtenues par l'application des opérations `Project`, `UnaryProject` et `Extend`. Pour chaque cohérence souple, nous donnons un exemple qui montre clairement ce type de transformations.



(a) P

(b) transformation de P en P'''

(c) P'''

FIGURE 2.7 – Exemple de l'EPT `Extend` sur une fonction de coût binaire

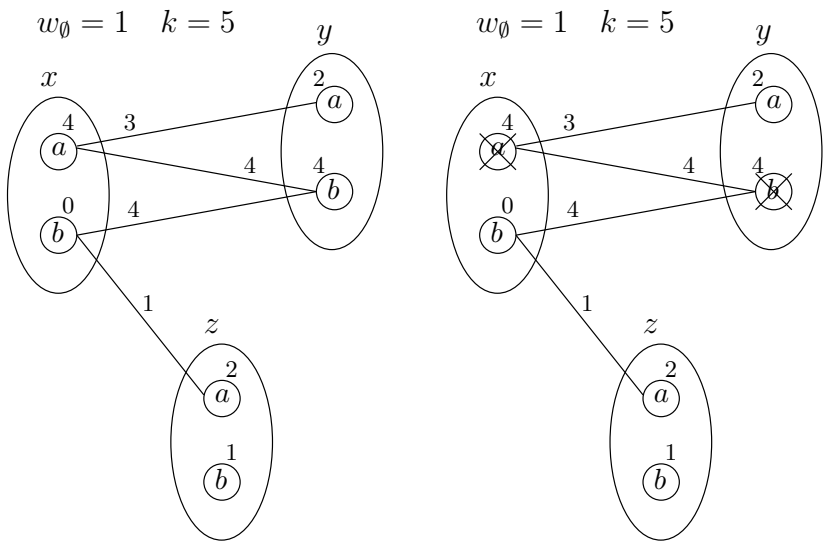
2.3.2 La cohérence de nœud (NC, NC*)

La cohérence de nœud (en anglais *NC*, *NC** pour Node Consistency) est introduite pour le formalisme WCSP par Larrosa [Larrosa, 2002]. Son principe consiste à supprimer toutes les valeurs a de $dom(x)$ telle que $w_\emptyset \oplus w_x(a) = k$, et d'avoir au moins une valeur a de $dom(x)$ dont le coût unaire est nul ; autrement dit, $\exists a \in dom(x)$ tel que $w_x(a) = 0$. Avant de définir formellement la cohérence de nœud, nous introduisons la notion de *support unaire* pour une variable sur sa contrainte unaire.

Définition 39 (Support unaire)**.** *La valeur (x, a) est un support unaire pour la variable x sur la contrainte unaire w_x si et seulement si $w_x(a) = 0$.*

Définition 40 (NC, NC*)**.** *Une valeur (x, a) est dite nœud-cohérente (NC) si $w_\emptyset \oplus w_x(a) < k$. Une variable x est NC si toutes ses valeurs sont NC. Une variable x est NC* si toutes ses valeurs sont NC et si elle possède un support unaire sur la contrainte w_x. Un WCN P est NC si toutes ses variables sont NC. Il est NC* si toutes ses variables sont NC*.*

Pour bien comprendre la définition précédente, considérons l'exemple du WCN P représenté par la figure 2.8. Sur cette figure, il est clair que P n'est pas NC*. En effet, d'une part les valeurs (x, a), (y, b) ne sont pas NC ($w_\emptyset + w_x(a) = w_\emptyset + w_y(b) = k = 5$), et d'autre part, les variables y et z n'ont pas de supports unaires. Pour transformer P en un autre WCN P' NC* et équivalent à P, nous supprimons les deux valeurs (x, a) et (y, b). Le but de cette opération est de rendre toutes les valeurs de P NC. Ensuite, nous créons des supports unaires pour les variables y et z à l'aide des EPTs que nous avons déjà présenté précédemment. Pour la



(a) P (b) P'

FIGURE 2.8 – Exemple de la cohérence NC*

variable y, nous transférons un coût égal à 2 depuis cette dernière vers la contrainte d'arité nulle $w_\emptyset$. Cela se fait à l'aide de l'EPT `UnaryProject`$(w_y, 2)$. De même, pour la variable z, nous transférons un coût égal à 1 avec un `UnaryProject`$(w_z, 1)$. À la fin, nous obtenons une nouvelle borne $w_\emptyset = 4$. L'algorithme qui transforme le WCN P en un WCN P' nœud-cohérent est donné par l'algorithme 10. Notons que la complexité temporelle de l'algorithme est $O(nd)$ et sa complexité spatiale est $O(nd)$ [Larrosa, 2002].

Algorithme 10 : NC*

Entrées : $P = (\mathscr{X}, \mathscr{C}, k)$: WCN

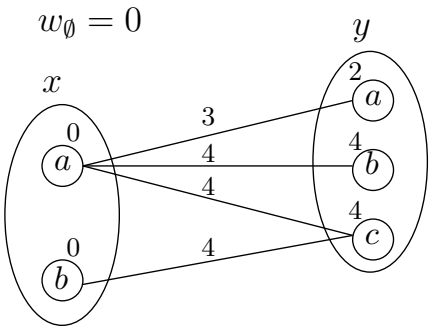
pour chaque $x \in \mathscr{X}$ **faire**
  $\alpha \leftarrow \min_{a \in dom(x)} w_x(a)$;
  `UnaryProject`(w_x, α);
pour chaque $x \in \mathscr{X}$ **faire**
  pour chaque $a \in dom(x)$ **faire**
    si $w_\emptyset \oplus w_x(a) = k$ **alors** $dom(x) \leftarrow dom(x) \setminus \{a\}$;

2.3.3 La cohérence d'arc souple (AC, AC*)

La cohérence d'arc souple (en anglais *AC*, *AC** pour Arc Consistency) est une généralisation de la cohérence d'arc déjà présentée pour le cadre CSP classique au chapitre 1. La notion de cohérence d'arc souple est introduite dans [Schiex, 2000] pour le cadre VCSP. Dans ce travail, l'auteur a proposé un algorithme qui établit la cohérence d'arc souple pour la famille des contraintes souples binaires. Un algorithme d'établissement d'AC* pour le cadre WCSP

binaire avec une complexité temporelle en $O(n^2d^2 + ed^3)$ et spatiale en $O(ed)$ est proposé par Larrosa dans [Larrosa, 2002]. Notons que pour la structure de valuation $\mathscr{S}(1)$, AC* est équivalent à la cohérence d'arc classique dans le cadre CSP. Un autre algorithme qui établit la cohérence d'arc souple généralisé pour les contraintes d'arité quelconque (en anglais GAC* pour Generalized Arc Consistency) a été proposé dans [Lee and Leung, 2010]. Avant de donner la définition de la cohérence d'arc souple pour le cadre WCSP, nous introduisons d'abord la notion de *support simple* sur une contrainte. Pour des raisons de simplicité, nous définissons la notion de support simple sur des contraintes binaires. Notons que cette définition peut être généralisée au cas des contraintes d'arité quelconque.

Définition 41 (Support simple)**.** *Soient w_{xy} une contrainte binaire et $a \in dom(x)$. (y, b) est dite support simple de la valeur (x, a) sur la contrainte w_{xy} si et seulement si $w_{xy}(a, b) = 0$*



(a) P

FIGURE 2.9 – Exemple de support simple sur une contrainte

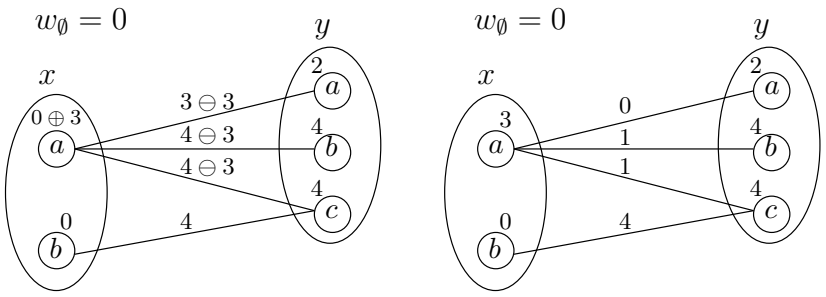
Par exemple, sur la figure 2.9 la valeur (x, a) n'a pas de support simple sur la contrainte w_{xy}. En revanche, la valeur (x, b) a deux supports simples sur la contrainte w_{xy} à savoir (y, a) et (y, b).

Si nous souhaitons établir un support pour la valeur (x, a), il suffit de projeter un coût 3 qui représente le coût minimal des tuples $t \in l(\{xy\})$ tels que $t[x] = a$ depuis les tuples de la contrainte w_{xy} contenant (x, a) vers la valeur (x, a). Cette projection se fait à l'aide de l'EPT `Project`$(w_{xy}, x, a, 3)$. La figure 2.10 montre les étapes de création d'un support simple pour la valeur (x, a) du WCN P de la figure 2.9.

Dans ce qui suit, nous donnons la définition formelle de AC ainsi que l'algorithme permettant de l'établir.

Définition 42 (AC)**.** *Une valeur (x, a) est dite arc-cohérente (AC) sur la contrainte w_{xy} si elle possède un support simple sur la contrainte w_{xy}. Une variable x est dite arc-cohérente si toutes ses valeurs sont arc-cohérentes sur toutes les contraintes qui portent sur la variable x. Un WCN P est arc-cohérent si toutes ses variables sont arc-cohérentes.*

Le principe de la définition ci-dessus est le même que celui de la définition de la cohérence d'arc pour le cadre CSP classique. D'une manière générale, il faut établir un support pour



(a) `Project`$(w_{xy}, x, a, 3)$ (b) P'

FIGURE 2.10 – Établir un support simple sur une contrainte

chacune des valeurs du WCN. La seule différence entre les deux formalismes (CSP et WCSP) réside dans la définition du support d'une valeur sur une contrainte. Nous rappelons que dans le cadre CSP classique, un support d'une valeur sur une contrainte est un tuple autorisé (libellé à vrai) contenant cette valeur, tandis que, dans le cadre WCSP un support d'une valeur dans une contrainte est un tuple totalement autorisé (un tuple dont le coût est égal à 0) contenant cette valeur.

L'algorithme 13 [Larrosa, 2002], représente les étapes de la transformation d'un WCN binaire en un autre WCN arc-cohérent équivalent. Ce dernier se base sur l'algorithme AC2001 [Bessiere and Régin, 2001] proposé dans le cadre CSP classique. Il utilise une structure de donnée $Last(x, a, y)$ (initialisée à null) pour stocker le support simple courant de la valeur (x, a) sur la contrainte w_{xy}. L'algorithme utilise deux procédures `modifierVariable` et `trouverSupportsAC`. La procédure `modifierVariable` supprime les valeurs nœud-incohérentes du domaine de la variable passée en paramètre et retourne vrai si le domaine de la variable a changé. La procédure `trouverSupportsAC` rétablit un nouveau support simple pour les valeurs qui ont perdu leur support. L'algorithme principal `W-AC2001` utilise une queue Q (initialisée avec l'ensemble $\mathscr{X}$) contenant l'ensemble des variables dont le domaine a été changé. Pour chaque variable x appartenant à Q l'algorithme établit un support simple pour toutes les valeurs a dans $dom(s)$ sur toute contrainte qui contient dans son scope la variable x.

Algorithme 11 : `modifierVariable`

Entrées : x : Variable
Sorties : Booléen

```
1 change ← faux;
2 pour chaque a ∈ dom(x) faire
3     si w_∅ ⊕ w_x(a) = k alors
4         dom(x) ← dom(x) − {a};
5         change ← vrai;
6 retourner change;
```

Considérant l'exemple du WCN P' de la figure 2.10. Ce WCN est AC mais pas NC*, car

Algorithme 12 : `trouverSupportsAC`

Entrées : x, y : Variable

```
1 pour chaque a ∈ dom(x) faire
2     si Last(x, a, y) ∉ dom(y) alors
3         v ← argmin_{b∈dom(y)} w_xy(a, b);
4         α ← w_xy(a, v);
5         Last(x, a, y) ← v;
6         Project(w_xy, x, a, α);
```

Algorithme 13 : `W-AC2001`

Entrées : $P = (\mathscr{X}, \mathscr{C}, k)$: WCN

Sorties : Booléen

```
1 Q ← 𝒳;
2 tant que Q ≠ ∅ faire
3     x ← pop(Q);
4     pour chaque w_xy ∈ 𝒲 faire
5         trouverSupportsAC(x, y);
6         si modifierVariable(x) alors Q ← Q ∪ {x};
```

toute affectation de la variable y a un coût minimal de 2. Par conséquent, on peut augmenter la borne inférieure $w_\emptyset$ de 2. Donc, la combinaison de ces deux cohérences (AC et NC*) peut augmenter la borne inférieure $w_\emptyset$. D'où, la définition de la cohérence d'arc AC* [Larrosa, 2002] :

Définition 43 (AC*)**.** *Une variable x est AC* si elle est AC et NC*. Un WCN binaire P est AC* si toutes ses variables sont AC*.*

La propriété AC* représente une extension de la propriété AC. En effet, l'établissement de la propriété AC* sur un WCSP P consiste à établir la cohérence d'arc AC ainsi que la cohérence de nœud NC*. La relation entre la propriété AC* et les deux propriété AC et NC* est une relation d'*implication*. La définition formelle de la relation d'implication est la suivante :

Définition 44. *Soient deux cohérences ϕ et ψ. On dit que ϕ* implique *ψ si tout WCN P qui vérifie ϕ vérifie aussi ψ.*

Nous remarquons immédiatement que AC* implique NC* et AC. En effet, par définition tout WCN qui est AC* est à la fois NC* et AC. Notons qu'il existe une autre relation entre les cohérences qui s'appuie sur la comparaison des bornes inférieures obtenues après l'établissement de ces dernières (voir la sous-section 2.3.11).

2.3.4 La cohérence d'arc complète (FAC, FAC*)

Avant de donner la définition la cohérence d'arc complète (en anglais Full Arc Consistency pour *FAC*, *FAC**) [Cooper, 2003] , nous introduisons d'abord la notion d'un support complet pour une valeur sur une contrainte. Pour des raisons de simplicité, nous définissons la notion de support complet sur des contraintes binaires.

Définition 45 (Support complet)**.** *Soient w_{xy} une contrainte et $a \in dom(x)$. (y, b) est un support complet de la valeur (x, a) sur la contrainte w_{xy} si et seulement si $w_{xy}(a, b) \oplus w_y(b) = 0$*

L'établissement d'un support complet pour une valeur (x, a) sur une contrainte w_{xy} est effectué en trois étapes :

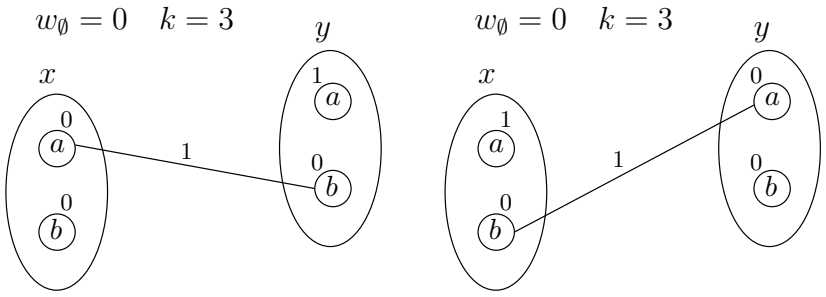
1. Définir la valeur $b \in dom(y)$ qui sera le support complet de la valeur (x, a) sur la contrainte w_{xy}, telle que $b = argmin_{v \in dom(y)}\{w_{xy}(a, v) \oplus w_y(v)\}$.
2. Exécuter la fonction $Extend(w_y, b, w_{xy}, \alpha)$ avec $\alpha = w_y(b)$. Cette opération effectue une extension d'un coût $w_y(b)$ depuis la valeur b de la contrainte unaire w_y vers les tuples de la contrainte w_{xy} pour forcer le coût de la valeur (y, b) à 0.
3. Exécuter la fonction $\texttt{Project}(w_{xy}, x, a, \alpha)$ avec $\alpha = w_{xy}(a, b) \oplus w_y(b)$. Cette opération effectue une projection d'un coût $w_{xy}(a, b) \oplus w_y(b)$ depuis les tuples de la contrainte w_{xy} vers la valeur a de la contrainte w_x pour forcer le tuple $\{(x, a), (y, b)\}$ à 0.

Définition 46 (FAC)**.** *Une valeur (x, a) est dite arc-cohérente complète (FAC) sur la contrainte w_{xy} si elle possède un support complet sur cette dernière. Une valeur (x, a) est dite arc-cohérente complète si elle possède un support complet sur toutes les contraintes impliquant x. Une variable x est dite arc-cohérente complète si toutes ses valeurs sont arc-cohérentes complètes. Un WCN P est dit arc-cohérent complet si toutes ses variables sont arc-cohérentes complètes.*

Comme le montre la définition ci-dessus, la cohérence d'arc complète consiste à établir un support complet pour chaque valeur de chaque variable d'un WCN P. Notons que la combinaison des deux propriétés FAC et NC* donne une autre cohérence qui s'appelle FAC*. La définition de FAC* est comme suit :

Définition 47 (FAC*)**.** *Une variable x est FAC* si elle est FAC et NC*. Un WCN P est FAC* s'il est à la fois FAC et NC*.*

Tout comme AC*, la propriété FAC* implique la propriété NC*. En revanche, les deux propriétés FAC et FAC* ne peuvent être établies pour tous les WCNs. Pour prouver ce résultat, prenons un exemple. Soit le WCSP P de la figure 2.11. Ce dernier n'est pas arc-cohérent complet car la valeur (x, a) n'a pas de support complet sur la contrainte w_{xy}. L'établissement d'un support pour la valeur (x, a) transforme P en un autre WCSP P' équivalent. Cette transformation établit un support complet pour la valeur (x, a), ce qui rend la variable x arc-cohérente complète. En revanche, la valeur (y, a) n'est plus arc-cohérente complète car elle perd son support complet sur la contrainte w_{xy} à cause de la transformation de P en P'. Cependant, l'établissement d'un nouveau support complet pour la valeur (y, a) transforme P' en P.



(a) P (b) P'

FIGURE 2.11 – Exemple d'impossibilité d'établir un support complet sur tout le réseau

2.3.5 La cohérence d'arc dirigée (DAC, DAC*)

Comme nous l'avons précisé dans la section précédente, la cohérence d'arc complète pose un problème lorsque le WCN sur lequel elle est appliquée ne possède pas une fermeture par arc-cohérence complète. Pour résoudre ce problème, un travail a été introduit dans [Cooper, 2003] et raffiné dans [Larrosa and Schiex, 2003] qui consiste à vérifier la cohérence d'arc complète pour une variable donnée sur seulement un sous-ensemble de contraintes impliquant cette variable. L'idée est d'imposer un ordre sur les variables, ensuite, pour chaque variable sélectionner un sous-ensemble de contraintes impliquant cette dernière et uniquement des variables qui sont strictement plus grandes qu'elle selon l'ordre qui a été choisi. Dans cette section, nous examinons la propriété de la cohérence d'arc dirigée (en anglais *DAC*,*DAC** pour Directed Arc Consistency) et nous montrons avec un exemple les étapes d'établissement de cette propriété sur un WCN. Dans ce qui suit, nous supposons que l'ensemble des variables $\mathscr{X}$ est totalement ordonné par >. La définition de la propriété de la cohérence d'arc dirigée [Larrosa and Schiex, 2003] est donnée par :

Définition 48. *[DAC, DAC*] Une valeur* (x, a) *est arc-cohérente dirigée* (*DAC*) *sur une contrainte* w_{xy} *avec* $y > x$ *si elle possède un support complet sur cette dernière. Une variable x est DAC si toutes ses valeurs sont DAC sur toutes les contraintes* w_{xy} *telle que* $y > x$*. Un WCN P est DAC* si toutes ses variables sont DAC et NC*.*

Établir la cohérence d'arc dirigée revient à créer un support complet pour les valeurs de chaque variable sur un sous-ensemble de contraintes qui respecte l'ordre défini sur les variables. L'algorithme 15 (proposé dans [Larrosa and Schiex, 2003]) établit la cohérence d'arc dirigée pour le cas binaire avec une complexité temporelle en $O(ed^2)$ et une complexité spatiale en $O(ed)$. Cet algorithme est conçu pour être utilisé seul pour établir DAC* ou en conjonction avec l'algorithme AC* pour établir la propriété FDAC* que nous présenterons dans la sous-section 2.3.6. Par conséquent, l'algorithme 15 utilise deux queues Q et R initialisées à $\mathscr{X}$. Si une valeur est supprimée du domaine d'une variable, cette variable est insérée dans Q pour informer l'algorithme AC* de la suppression. La queue R est utilisée pour stocker les variables dont au moins le coût d'une valeur dans leur domaine passe de 0 à un entier strictement supérieur à 0, dans ce cas certaines valeurs dans les domaines des variables plus petites peuvent perdre leur support complet et un nouveau support complet doit être établi. La fonc-

tion $trouverSupportsComplets(x, y)$ (avec $x < y$) établit les supports complets des valeurs $a \in dom(x)$ sur la contrainte w_{xy}. Si le coût d'une valeur (x, a) passe de 0 à un entier strictement supérieur à 0 la fonction retourne vraie et la variable x est insérée dans la queue R. Dans cette fonction, la structure $P[a]$ permet de stocker le coût qui va être projeté vers la valeur a, la structure $E[b]$ permet de stocker le coût qui va être étendu depuis la valeur b et la structure $Last(x, a, y)$ stocke le support de la valeur (x, a) sur la contrainte w_{xy}.

Algorithme 14 : `trouverSupportsComplets`

Entrées : x, y : Variable
Sorties : Booléen

$changer \leftarrow faux$;
pour chaque $a \in dom(x)$ $t.q$ $w_{xy}(a, Last(x, a, y)) \oplus w_y(Last(x, a, y)) > 0$ **faire**
  $Last(x, a, y) \leftarrow argmin_{b \in dom(y)} w_{xy}(a, b) \oplus w_y(b)$;
  $P[a] \leftarrow w_{xy}(a, Last(x, a, y)) \oplus w_y(Last(x, a, y))$;
  si $P[a] > 0 \wedge w_x(a) = 0$ **alors** $changer \leftarrow vrai$;
pour chaque $b \in dom(y)$ **faire**
  $Last(y, b, x) \leftarrow argmax_{a \in dom(x)} P[a] \ominus w_{xy}(a, b)$;
  $E[b] \leftarrow P[Last(y, b, x)] - w_{xy}(Last(y, b, x), b)$;
pour chaque $b \in dom(y)$ **faire** `Extend`$(w_y, b, w_{xy}, E[b])$;
pour chaque $a \in dom(x)$ **faire** `Project`$(w_{xy}, x, a, P[a])$;
`UnaryProject`$(w_x, \min_{a \in dom(x) w_x(a)})$;
retourner $changer$;

Algorithme 15 : `DAC`*

Entrées : $P = (\mathscr{X}, \mathscr{C}, k)$: WCN

$Q \leftarrow \mathscr{X}$;
$R \leftarrow \mathscr{X}$ **tant que** $R \neq \emptyset$ **faire**
  $y \leftarrow pop(R)$;
  si $modifierVariable(y)$ **alors** $Q \leftarrow Q \cup \{y\}$;
  pour chaque $w_{xy} \in \mathscr{W}$ $t.q$ $x < y$ **faire**
    si $trouverSupportsComplets(x, y)$ **alors** $R \leftarrow R \cup \{x\}$;
  pour chaque $x \in \mathscr{X}$ **faire**
    si $modifierVariable(x)$ **alors** $Q \leftarrow Q \cup \{x\}$;

Pour comprendre la notion de la cohérence d'arc dirigée, soit l'exemple du WCN P donné par la figure 2.11. Nous remarquons qu'avec l'ordre $x > y$, le WCN P est DAC car chaque valeur de la variable y a un support complet sur la contrainte w_{xy}. Puisque l'ordre défini est $x > y$, nous vérifions uniquement les supports complets de la variable y sur la contrainte w_{xy}. La vérification des supports complets pour la variable x sur la contrainte w_{xy} est ignorée en respectant cet ordre (voir la def. 48). De plus, P est DAC* car toutes ses variables sont

également NC*. En revanche, avec l'ordre $y > x$, le WCN P n'est plus DAC. En effet, avec l'ordre $y > x$ il faut vérifier l'existence d'un support complet pour les valeurs de la variable x sur la contrainte w_{xy}. Ainsi, nous remarquons que la valeur (x, a) n'a pas de support complet sur la contrainte w_{xy}. L'établissement de la propriété DAC sur P résulte en un WCN P' qui vérifie la propriété DAC et qui est en même temps équivalent à P. Notons aussi que P' est DAC* car il est également NC*. Ainsi, pour un WCN P, il existe plusieurs fermetures par arc-cohérence dirigée pour le même ordre total utilisé sur les variables (voir la sous-section 2.3.11.1).

2.3.6 La cohérence d'arc dirigée complète (FDAC*)

La cohérence d'arc dirigée complète (en anglais *FDAC** pour Full Directed Arc Consistency) est introduite dans [Larrosa and Schiex, 2003]. C'est une forme de cohérence d'arc qui maintient simultanément la cohérence d'arc dirigée DAC* dans un ordre total défini sur les variables, et la cohérence d'arc AC*. Pour bien expliquer la manière d'établir la propriété FDAC* sur un WCN, considérons l'exemple du WCN P de la figure 2.12, et soit $z < x < y$ l'ordre défini sur ses variables. Selon cet ordre, il est clair que le WCN P est DAC*. En revanche, il est n'est pas AC* car les valeurs (y, a) (resp. (y, b)) n'ont pas de support simple sur les contraintes w_{xy} (resp. w_{yz}). D'où une première observation : *DAC* n'implique pas forcément AC**. Pour établir la propriété AC* sur P, nous créons un support simple pour (y, a) et (y, b) de la manière suivante :

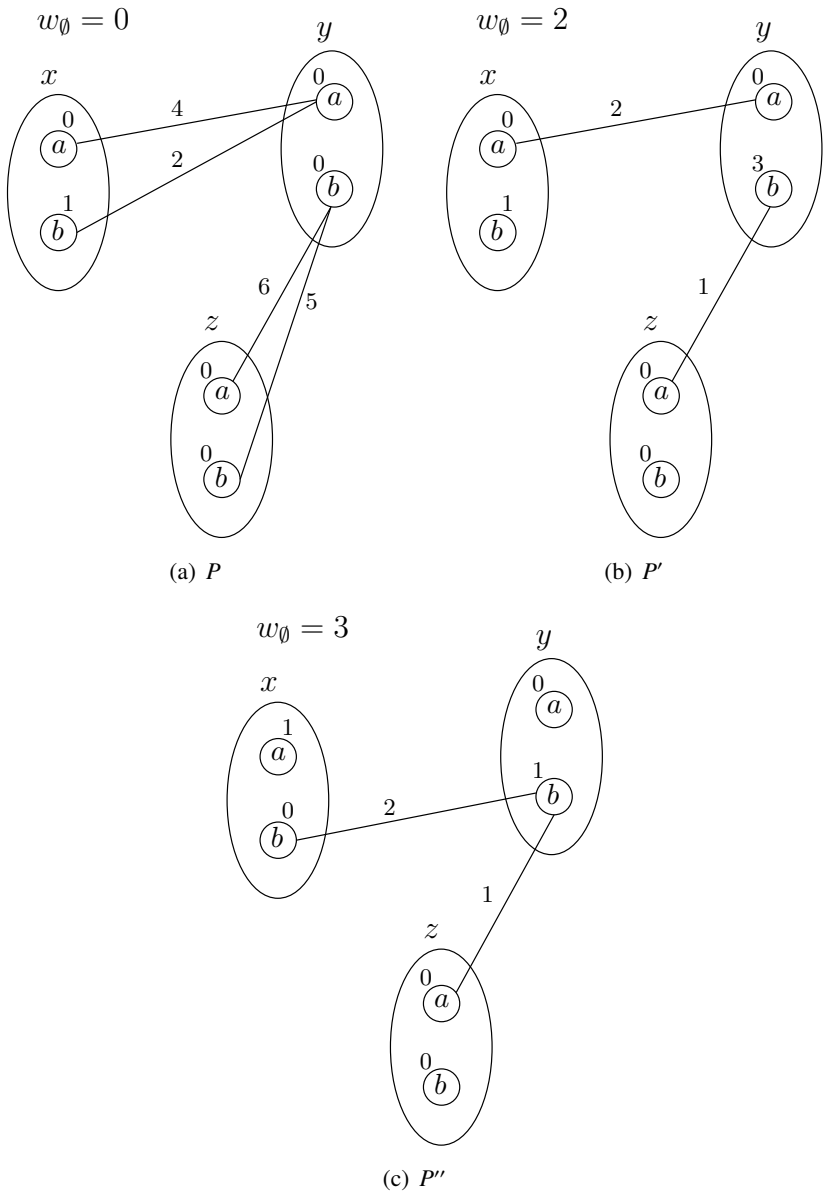
1. Pour (y, a), nous projetons un coût égal à 2 depuis les tuples de la contrainte w_{xy} vers la valeur (y, a) (avec la fonction `Project`).
2. De même, pour (y, b), nous projetons un coût égal à 5 depuis les tuples de la contrainte w_{yz} vers la valeur (y, b) (avec la fonction `Project`). Notons que la création de ces deux supports fait perdre la propriété NC* à la variable y. Pour rétablir la propriété NC* sur la variable y, il faut :
3. projeter un coût 2 depuis les valeurs de y vers la borne inférieure $w_\emptyset$ (avec la fonction `UnaryProject`).

L'établissement de la propriété AC* sur P l'a transformé en un autre WCN P' équivalent, AC* et avec une nouvelle borne $w_\emptyset$ plus grande que la première. Notons que le WCN P' obtenu n'est plus DAC* car la valeur (x, a) a perdu son support complet sur la contrainte w_{xy}. Donc, l'établissement de la propriété AC* sur un WCN P DAC* peut lui faire perdre cette propriété. D'où une seconde observation : *AC* n'implique pas forcément DAC**.

Pour rétablir le support complet de la valeur (x, a) il faut :

1. Étendre un coût 2 depuis la valeur (y, b) sur les tuples de la contrainte w_{xy} (avec la fonction `Extend`).
2. Projeter un coût 2 depuis les tuples de la contrainte w_{xy} vers la valeur (x, a) (avec la fonction `Project`).
3. Projeter un coût 1 depuis les valeurs de x vers la borne inférieure $w_\emptyset$ pour rétablir la propriété NC* sur la variable x (avec la fonction `UnaryProject`).

De cette façon, nous avons transformé le WCN P' en un autre WCN P'' équivalent, DAC* et AC* et avec une nouvelle borne $w_\emptyset$ plus grande que celle de P'. Le WCN P'' représente un point fixe par rapport aux deux propriétés AC* et DAC*, ce WCN est dit FDAC*-cohérent.



(a) P (b) P' (c) P''

FIGURE 2.12 – Comparaison entre AC* et DAC* : "AC* n'implique pas DAC* sur le WCN P' et DAC* n'implique pas AC* sur le WCN P"

En pratique, lors de l'établissement de la propriété FDAC*, le maintien de la propriété AC* peut se faire uniquement dans l'ordre inverse défini sur les variables. Cela est justifié par le fait que dans l'ordre total défini sur les variables, DAC* maintient déjà la propriété AC*.

Définition 49 (FDAC*)**.** *Un WCN P est FDAC* s'il est AC* et DAC*.*

Nous remarquons que la propriété FDAC* implique à la fois la propriété AC* et la propriété DAC*. Un algorithme avec une complexité temporelle en $O(end^3)$ et une complexité spatiale en $O(ed)$ pour les réseaux de contraintes binaires qui établit la propriété FDAC* est proposé

dans [Larrosa and Schiex, 2003]. Dans le même article, les auteurs ont montré expérimentalement que l'algorithme qui maintient la propriété FDAC* pendant la recherche (MFDAC*) est plus performant que MAC* et MDAC*.

Malgré l'amélioration de la borne inférieure $w_\emptyset$ par rapport à AC* et DAC*, la valeur de $w_\emptyset$ dépend fortement de l'ordre total défini sur les variables. Dans les travaux de Schiex et al. [Cooper and Schiex, 2004], les auteurs ont montré que définir un ordre total sur les variables qui maximise la borne inférieure $w_\emptyset$ est un problème NP-dur. Pour remédier à ce problème, De givry et al. [de Givry et al., 2005] ont proposé une autre forme de cohérence d'arc EAC* que nous détaillons dans la section suivante.

2.3.7 La cohérence d'arc existentielle (EAC*)

La cohérence d'arc existentielle (en anglais *EAC* pour Existential Arc Consistency) est proposée dans [de Givry et al., 2005]. Son principe est d'établir pour au moins une valeur de chaque variable un support complet dans chaque contrainte impliquant cette variable. Le but de cette forme de cohérence est de se rapprocher au mieux de la cohérence d'arc complète FAC*.

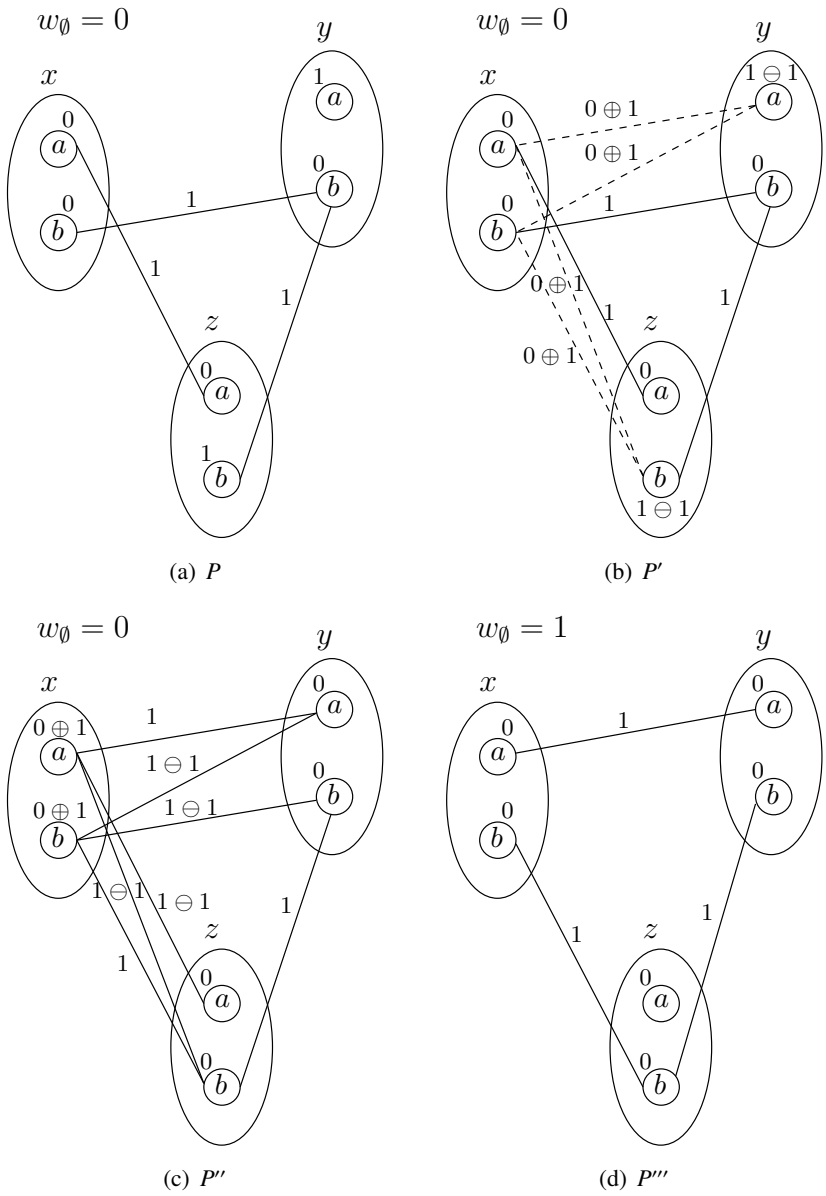
Avant de définir EAC* dans le cadre des réseaux de contraintes binaires, nous introduisons la notion du voisinage d'une variable.

Définition 50. *Pour toute variable x, nous désignons par $\Gamma(x)$ l'ensemble des contraintes portant sur la variable x.*

Définition 51 (EAC)**.** *Une variable x est arc-cohérente existentielle si et seulement si $\exists a \in dom(x)$ tel que $w_x(a) = 0$ et $\forall w_{xy} \in \Gamma(x), \exists b \in dom(y)$ tel que $w_{xy}(a,b) \oplus w_y(b) = 0$. Un WCN P est EAC si toutes ses variables sont EAC.*

Dans le but d'illustrer cette forme de cohérence, considérons l'exemple du WCN P donné par la figure 2.13 et soit $x > z > y$ l'ordre total défini sur les variables. Dans cette figure, notons que P est FDAC*-cohérent.

Cependant, il n'est pas EAC-cohérent car la variable x ne possède pas une valeur qui a en même temps un coût nul et un support complet sur chacune des contraintes où x apparaît. En effet, la valeur (x,a) n'a pas de support complet sur la contrainte w_{xz}. De même, la valeur (x,b) n'a pas de support complet sur la contrainte w_{xy}. Par conséquent, le coût unaire des deux valeurs de la variable x va pouvoir être augmenté de 1 en établissant un support complet pour celles-ci. L'établissement d'un support complet pour la valeur (x,a) (resp. (x,b)) peut se faire en deux étapes. Premièrement, il faut étendre un coût 1 depuis la valeur (z,b) (resp. (x,a)). Cet première étape permet d'obtenir le WCN P'. Deuxièmement, il faut projeter un coût 1 depuis les tuples de la contrainte w_{xz} (resp. w_{xy}) vers la valeur (x,a) (resp. (x,b)). Nous remarquons que le WCN P'' résultant de ces opérations n'est pas NC* car les valeurs du domaine de x sont strictement supérieures à 0. L'établissement de NC* sur la variable x augmente la borne inférieure $w_\emptyset$ de 1. Le WCN P''' obtenu est maintenant EAC et équivalent à P. Il faut noter que cet exemple indique clairement que la propriété *FDAC* n'implique pas la propriété EAC*. De même, comme le montre la figure 2.14, la propriété EAC n'implique pas la propriété FDAC*. En effet, le WCN P donné par la figure 2.14 est clairement EAC-cohérent. En revanche, selon



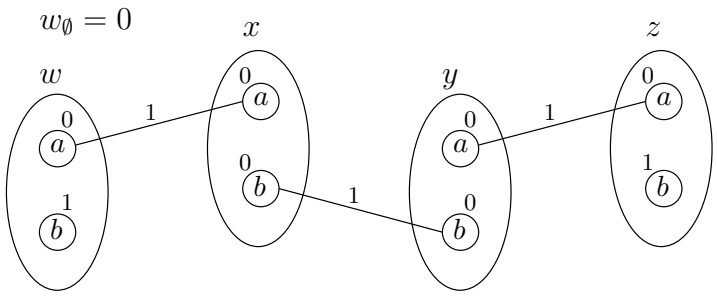
(a) P (b) P'

(c) P'' (d) P'''

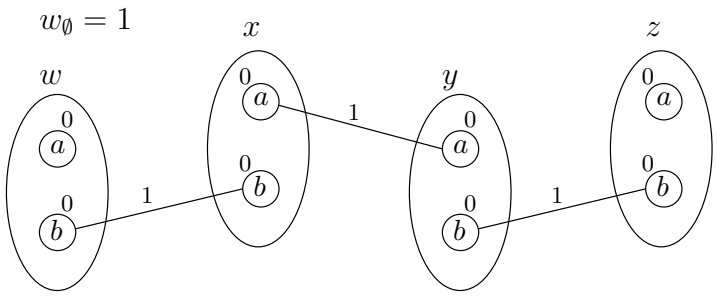
FIGURE 2.13 – Établissement de la cohérence d'arc existentielle EAC

l'ordre total ($w < x < y < z$) défini sur les variables, nous remarquons que la valeur (y, a) n'a pas de support complet sur la contrainte w_{yz}. Par conséquent, le WCN P n'est pas FDAC*-cohérent. Après l'établissement de la propriété FDAC*, nous obtenons le WCN P' qui est à la fois FDAC* et EAC avec une nouvelle borne inférieure $w_\emptyset = 1$ plus grande que celle de P. Le WCN P' est un point fixe par rapport au deux propriétés FDAC* et EAC. L'établissement simultané de FDAC* et EAC sur un WCN donné est une cohérence qui s'appelle EDAC* et qui représente le sujet de la section suivante.

Remarque 4. *Dans la figure 2.14, nous donnons uniquement le résultat final de l'établisse-*



(a) P



(b) P'

FIGURE 2.14 – FDAC* vs EAC

ment de la propriété FDAC sur P. Ainsi, nous ignorons toutes les étapes intermédiaires de la transformation de P en P′. Un exemple détaillé de l'établissement de la propriété FDAC* sur un WCN est illustré dans la sous-section 2.3.6.*

2.3.8 La cohérence d'arc existentielle dirigée (EDAC*)

La cohérence d'arc existentielle dirigée [de Givry et al., 2005] (en anglais *EDAC** pour Existential Directed Arc Consistency) est basée sur la combinaison des deux propriétés FDAC* et EAC. La définition formelle de la cohérence d'arc existentielle dirigée EDAC* est donnée par :

Définition 52 (EDAC*)**.** *Un WCN P est EDAC* s'il est FDAC* et EAC.*

Un algorithme avec une complexité temporelle en $O(ed^2 max\{nd, k\})$ et une complexité spatiale en $O(ed)$ établissant la propriété EDAC* sur les réseaux de contraintes binaires est proposé dans [de Givry et al., 2005]. La comparaison des résultats entre l'algorithme MEDAC* qui maintient la propriété EDAC* pendant la recherche et les autres algorithmes MAC*, MDAC*... (en terme de temps et de nœuds visités), montre que dans le cas général, MEDAC* est plus performant. Toutefois, il existe une autre cohérence plus performant que EDAC*. Ce type de cohérence est appelé VAC. Nous présentons cette cohérence dans la prochaine section.

2.3.9 La cohérence d'arc virtuelle (VAC)

Dans cette section, nous présentons la cohérence d'arc virtuelle (en anglais *VAC* pour Virtual Arc Consistency) proposée dans [Cooper et al., 2008]. D'une manière générale, VAC consiste à planifier et appliquer itérativement des séquences d'opérations de transferts de coûts *fractionnaires* qui garantissent de transformer un WCN P en un autre WCN P' équivalent avec une borne inférieure améliorée. Pour la cohérence VAC ainsi que la cohérence OSAC que nous présentons dans la section suivante, nous n'exigeons plus que les coûts soient des entiers et utilisons des valeurs réelles. Dans ce qui suit, nous définissons la cohérence d'arc virtuelle et nous illustrons celle-ci.

La définition de la cohérence d'arc virtuelle se base sur la transformation d'un WCN P en un CN noté $Bool(P)$. Le CN $Bool(P)$ est défini comme suit :

Définition 53. *Étant donné un WCN $P = (\mathscr{X}, \mathscr{W}, k)$, on définit $Bool(P)$ comme étant le CN $(\mathscr{X}, \mathscr{C})$ où $c_S \in \mathscr{C}$ si et seulement si $\exists w_S \in \mathscr{W}$ tel que $S \neq \emptyset$ et $\forall t \in l(S)(c_S(t) \Leftrightarrow w_S(t) = 0)$.*

En d'autres termes, $Bool(P)$ est un CN classique dont les tuples autorisés sont les tuples qui ont un coût 0 dans P. Les solutions de $Bool(P)$ (s'il y en a) ont un coût égal à $w_\emptyset$ dans P.

Définition 54 (VAC)**.** *Un WCN P satisfait la cohérence d'arc virtuelle VAC si et seulement si $AC(Bool(P)) \not\equiv \bot$.*

De cette définition nous retenons que si un WCN P ne satisfait pas la propriété VAC alors la fermeture arc-cohérente de $Bool(P)$ est vide. Cette dernière information va permettre d'augmenter la borne inférieure $w_\emptyset$. Pour comprendre la façon d'établir la cohérence d'arc virtuelle sur un WCN. Nous reprenons l'exemple du WCN P de la figure 2.15 donné par les auteurs dans [Cooper et al., 2008].

On peut vérifier que ce réseau satisfait la propriété EDAC. Notons que $Bool(P)$ est représenté par la figure 2.15(a) où un coût égal à 1 représente une combinaison interdite. Comme le montre la figure 2.15, l'établissement de la propriété VAC nécessite trois phases.

La première phase consiste à calculer AC(Bool(P)). Comme nous l'observons sur la figure 2.15(b) , les valeurs (y, a) et (z, b) n'ont pas de supports simple (en terme CSP) sur les contraintes w_{xy} et w_{xz} respectivement. Donc, les deux valeurs peuvent être supprimées des domaines de leurs variables (elles sont marquées avec un coût 1). Pour chaque suppression nous enregistrons la source qui a provoqué cette dernière avec une flèche en pointillée. La valeur (z, a) peut être aussi supprimée du fait de la suppression de la valeur (y, a). À la fin, nous obtenons une variable z avec un domaine vide. Nous concluons donc que $Bool(P)$ est incohérent et puisque les coûts du WCN sont des entiers naturels alors nous pouvons augmenter $w_\emptyset$ au moins d'une unité de coût avec un `UnaryProject` depuis la variable z.

La deuxième phase consiste à retracer les étapes de la première phase en commençant par la variable du domaine vide (ici z). Nous supposons qu'un coût α va pouvoir être transféré avec l'EPT `UnaryProject` depuis la variable z vers la borne inférieure $w_\emptyset$. Pour cela, il faut d'abord réussir à transférer le coût α sur chaque valeur de la variable z. Pour la valeur (z, b), nous pouvons obtenir le coût α directement depuis la variable x, en revanche, pour la valeur (z, a), ce coût α est obtenu depuis la variable y, mais cela n'est possible que si nous déplaçons le coût α depuis la variable x vers la valeur (y, a). Nous remarquons que nous avons deux

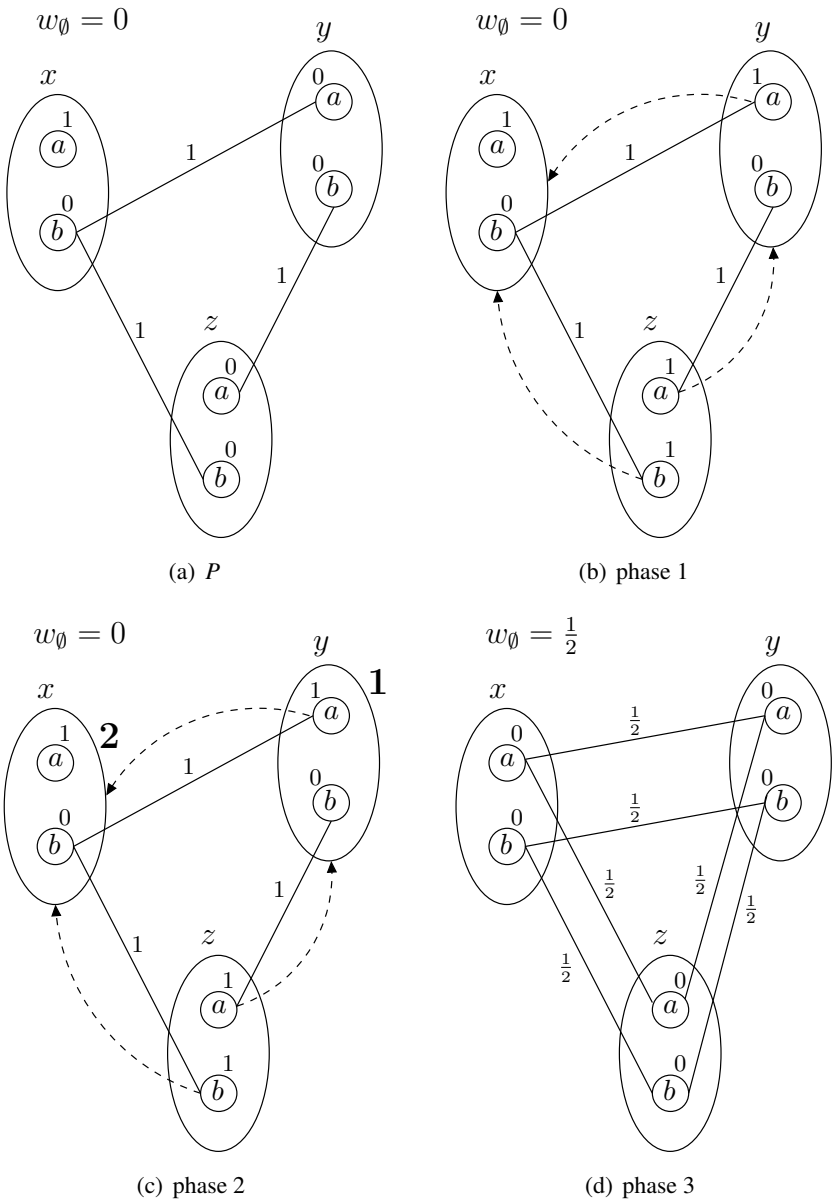


FIGURE 2.15 – Établissement de la cohérence d'arc virtuelle VAC

demandes de coût α depuis x vers les deux valeurs (z,b), (y,a) et une demande de coût α depuis y vers la valeur (z,a). Notons que ces demandes de coût sont représentées sur la figure 2.15(c) sur chacune des variables x et y avec un chiffre en gras. Étant donné que d'une part la quantité maximum de coût qu'on peut transférer depuis la variable x est de 1. Et d'autre part, nous avons deux demandes de transfert de coût α depuis la variable x. Nous concluons que la quantité maximum de coût que nous pouvons affecter à α est égale à $\frac{1}{2}$.

Dans la troisième phase, nous appliquons directement les transformations que nous avons identifiées dans la deuxième phase en utilisant les deux fonctions élémentaires `Project`,

`UnaryProject` et `Extend`. Ce qui permet d'obtenir le WCSP de la phase 3 de la figure qui est équivalent à P avec une borne inférieure améliorée.

Dans [Cooper et al., 2010], les auteurs ont montré à l'aide d'un exemple que l'algorithme proposé dans [Cooper et al., 2008] peut boucler indéfiniment. En effet, il peut exister des séquences infinies d'étapes faisant croître chacune d'une manière infinitésimale $w_\emptyset$. Pour résoudre ce problème un nouveau algorithme VAC_e est proposé avec une complexité temporelle en $O(ed^2k/\varepsilon)$ où ε représente le coût rationnel minimum non nul et une complexité spatiale en $O(ed)$.

2.3.10 La cohérence d'arc souple optimale (OSAC)

Les niveaux de cohérence présentés dans ce chapitre (AC, DAC, ...) peuvent être vus comme des heuristiques efficaces cherchant à s'approcher d'une fermeture arc-cohérente optimale (maximisant $w_\emptyset$). La cohérence d'arc souple optimale [Cooper et al., 2007] (en anglais *OSAC* pour Optimal Soft Arc Consistency) consiste à définir un ensemble d'EPT déplaçant des coûts rationnels entre les différentes portées d'un réseau WCN et maximisant la valeur de la fonction de coût d'arité nulle $w_\emptyset$. L'algorithme OSAC se base sur la résolution d'un problème de programmation linéaire (PL) sur $\mathbb{Q}$. Pour pouvoir générer ce PL, les auteurs ont relaxé la définition du problème WCSP pour permettre l'utilisation des coûts rationnels. La nouvelle structure proposé dans [Cooper et al., 2007] est la structure $S_{\mathbb{Q}}(k) = ([0,k], \oplus, <)$ où $[0,k]$ est l'intervalle des coûts rationnels (éléments de $\mathbb{Q}$) entre 0 et k. Avant de donner le premier théorème proposé dans [Cooper et al., 2007], nous introduisons la notion d'une transformation arc-cohérente souple (en anglais SAC pour Soft Arc Consistency Transformation).

Définition 55 (SAC)**.** *Soit P un WCN, une transformation arc-cohérente souple SAC sur P est un ensemble d'opérations de cohérence d'arc souple* `Project` *et* `UnaryProject`*, dont l'application simultanée transforme P en un autre WCN P' valide (avec des coûts rationnels positifs ou nuls) et équivalent.*

Définition 56 (OSAC)**.** *Un WCN P est dit OSAC-cohérent si et seulement s'il n'existe pas de transformation SAC sur P qui peut améliorer (augmenter) la borne inférieure $w_\emptyset$.*

Théorème 1. *Si la structure de valuation $S_{\mathbb{Q}}(+\infty)$ est utilisée, alors il est possible de trouver en temps polynomial une transformation SAC de P qui maximise la borne inférieure $w_\emptyset$ à condition que l'arité des contraintes de P soit bornée.*

Notons que si $k \neq +\infty$ l'augmentation de la borne inférieure $w_\emptyset$ après l'établissement de OSAC peut provoquer la perte de la propriété NC pour certaines valeurs, d'où la nécessité de supprimer ces valeurs. Après la suppression de ces valeurs, rétablir OSAC une nouvelle fois peut augmenter à nouveau $w_\emptyset$. Le nombre de valeurs à supprimer ne peut pas dépasser nd fois, donc le temps reste polynomial. En revanche, la preuve de l'optimalité de la fermeture finale reste une question ouverte. L'ensemble de transformation SAC qui fournit la borne inférieure maximale $w_\emptyset$ est défini par le système suivant :

$$\begin{cases} \max : \sum_{x \in \mathscr{X}} u_x \\ \forall x \in \mathscr{X}, \forall a \in dom(x), w_x(a) - u_x + \sum_{w_S \in \mathscr{W}, x \in S} P^S_{x,a} \geq 0 \\ \forall w_S \in \mathscr{W}, |S| > 1, \forall t \in l(S), w_S(t) - \sum_{x \in S} P^S_{x,t[x]} \geq 0 \end{cases}$$

$p^S_{x,a}$ représente le coût factorisé par l'EPT `Project` depuis les tuples de la contrainte w_S vers la valeur (x, a) et u_x représente le coût factorisé par `UnaryProject` depuis la contrainte unaire w_x vers la borne inférieure $w_\emptyset$.

La première inéquation concerne les coûts des valeurs de variables. Pour chaque valeur $a \in dom(x)$, le coût initial de cette valeur moins les transferts de coûts vers la contrainte d'arité nulle depuis la variable qui contient cette valeur u_x plus les transferts depuis les contraintes w_S d'arité supérieure ou égale à deux telles que $x \in S$ doit être au final positif ou nul. La seconde inéquation concerne les contraintes d'arité supérieure ou égale à deux. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les valeurs de ce tuple doit être au final positif ou nul. Ce problème de programmation linéaire est défini sur $\mathbb{Q}$ peut être résolu en temps polynomial [Karmarkar, 1984]. Pour expliquer le système

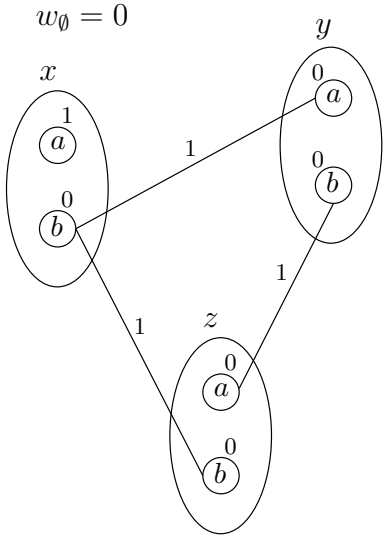


FIGURE 2.16 – Un exemple d'un WCN qui ne vérifie pas la propriété OSAC

d'équation OSAC considérons en premier l'exemple simple du WCN P de la figure 2.16. Le système d'équation de ce WCN se présente comme suit :

$\max : u_x + u_y + u_z$

$$\left\{\begin{array}{l|l|l} 1 - U_x + P^{xy}_{(x,a)} + P^{xz}_{(x,a)} \geq 0; & 0 - P^{xy}_{(x,a)} - P^{xy}_{(y,a)} \geq 0; & 0 - P^{xz}_{(x,b)} - P^{xz}_{(z,a)} \geq 0; \\ 0 - U_x + P^{xy}_{(x,b)} + P^{xz}_{(x,b)} \geq 0; & 0 - P^{xy}_{(x,a)} - P^{xy}_{(y,b)} \geq 0; & 1 - P^{xz}_{(x,b)} - P^{xz}_{(z,b)} \geq 0; \\ 0 - U_y + P^{xy}_{(y,a)} + P^{yz}_{(y,a)} \geq 0; & 1 - P^{xy}_{(x,b)} - P^{xy}_{(y,a)} \geq 0; & 0 - P^{yz}_{(y,a)} - P^{yz}_{(z,a)} \geq 0; \\ 0 - U_y + P^{xy}_{(y,b)} + P^{yz}_{(y,b)} \geq 0; & 0 - P^{xy}_{(x,b)} - P^{xy}_{(y,b)} \geq 0; & 0 - P^{yz}_{(y,a)} - P^{yz}_{(z,b)} \geq 0; \\ 0 - U_z + P^{xz}_{(z,a)} + P^{yz}_{(z,a)} \geq 0; & 0 - P^{xz}_{(x,a)} - P^{xz}_{(z,a)} \geq 0; & 1 - P^{yz}_{(y,b)} - P^{yz}_{(z,a)} \geq 0; \\ 0 - U_z + P^{xz}_{(z,b)} + P^{yz}_{(z,b)} \geq 0; & 0 - P^{xz}_{(x,a)} - P^{xz}_{(z,b)} \geq 0; & 0 - P^{yz}_{(y,b)} - P^{yz}_{(z,b)} \geq 0; \end{array}\right.$$

Pour bien comprendre le fonctionnement d'OSAC, nous reprenons l'exemple du WCN dans [Cooper et al., 2007]. Ce WCN est représenté par la figure 2.17(a). Sur cette figure, on peut vérifier que le WCN P est VAC-cohérent. En revanche, il n'est pas OSAC-cohérent car l'application de la transformation SAC suivante permet d'augmenter $w_\emptyset$ d'une unité de coût :

1. `Project`$(w_{yz}, y, a, -1)$: cette opération est équivalente à `Extend`$(w_{yz}, y, a, 1)$. Son but est de créer un coût égal à 1 pour chaque $t \in l(zy)$ avec $t[y] = a$.
2. `Project`$(w_{yz}, z, a, 1)$, `Project`$(w_{yz}, z, b, 1)$: ces opérations déplacent un coût égal à 1 depuis les tuples de la contrainte w_{yz} vers les valeurs a et b de la variable z.
3. `Project`$(w_{xz}, z, b, -1)$, `Project`$(w_{zw}, z, a, -1)$: ces opérations sont équivalentes à `Extend`$(w_{xz}, z, b, 1)$, `Extend`$(w_{zw}, z, a, 1)$ leur but est de projeter un coût égal à 1 vers les tuple des contraintes w_{xz} et w_{zw}.
4. `Project`$(w_{zw}, w, b, 1)$: cette opération déplace un coût égal à 1 depuis les tuples de la contrainte w_{zw} vers la valeur b de w.
5. `Project`$(w_{xz}, z, a, 1)$, `Project`$(w_{xz}, z, c, 1)$: ces opérations déplacent un coût égal à 1 depuis la contrainte w_{xz} vers les valeurs a et c de la variable x.
6. `Project`$(w_{xy}, x, c, -1)$, `Project`$(w_{xz}, y, a, 1)$: ces opérations déplacent un coût égal à 1 vers la valeur a de y.
7. `Project`$(w_{xw}, x, a, -1)$, `Project`$(w_{xw}, w, a, 1)$: ces opérations déplacent un coût égal à 1 vers la valeur a de w.
8. `UnaryProject`$(w_w, 1)$: cette opération augmente $w_\emptyset$ d'une unité de coût.

Les opérations décrites ci-dessus constituent la solution du PL donné par OSAC. Le WCN OSAC-cohérent que nous obtenons après ces opérations est le WCN P' présenté par la figure 2.17(b).

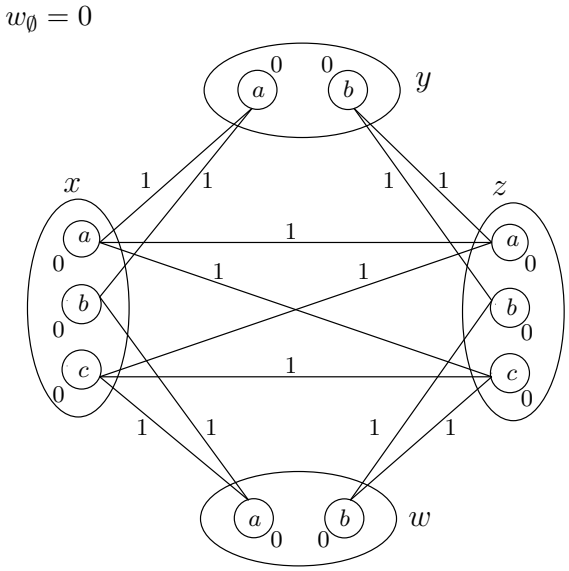
Parmi les cohérences citées précédemment, OSAC est celle fournissant la meilleure borne inférieure $w_\emptyset$. Néanmoins, elle nécessite des temps de calculs importants qui interdisent une utilisation systématique au cours de la recherche. Toutefois nous proposons au chapitre 4 une nouvelle cohérence plus efficace qu'OSAC en terme de calcul de borne inférieure.

Le tableau 2.4 résume la complexité spatiale et temporelle des différentes cohérences que nous avons présentées dans ce chapitre.

2.3.11 Les relations entre les différentes cohérences

Nous avons défini précédemment une relation d'implication entre les différentes cohérences. Il existe une deuxième relation qui consiste à comparer les cohérences selon la borne inférieure obtenue après l'établissement de ces dernières. Cette relation est dite $w_\emptyset$-supériorité, sa définition est donnée par :

Définition 57. *Soient deux cohérences ϕ et ψ. On dit que ϕ est $w_\emptyset$-*supérieur *à ψ si et seulement s'il n'existe pas un WCN P tel que $w_\emptyset(\psi \circ \phi(P)) \geq w_\emptyset(\phi(P))$, où $w_\emptyset(\theta(P))$ représente la borne inférieure maximale que l'on peut obtenir en établissant la propriété θ sur le WCN P.*



(a) P

(b) P'

FIGURE 2.17 – L'établissement de la propriété OSAC sur un WCN VAC-cohérent

Autrement dit, ϕ est $w_\emptyset$-supérieur à ψ si et seulement si appliquer ψ après ϕ ne permet jamais d'augmenter la borne inférieure $w_\emptyset$. Comme les cohérences ne sont pas forcément confluentes, la valeur de $w_\emptyset$ obtenue dépend de l'ordre dans lequel on effectue les EPTs et donc cette valeur n'est pas unique. La définition se fonde donc sur le plus grand $w_\emptyset$ que l'on peut obtenir pour s'affranchir de ce problème de confluence.

Si ϕ n'est pas $w_\emptyset$-supérieur à ψ et vice versa alors les deux propriétés sont dites incomparables noté ϕ est $w_\emptyset$-*incomparable* à ψ. Dans ce qui suit, nous établissons le lien entre les

algorithme	complexité en temps	complexité en espace
NC* [Larrosa, 2002]	$O(nd)$	$O(nd)$
AC* [Larrosa, 2002]	$O(n^2d^2 + ed^3)$	$O(ed)$
DAC* [Larrosa and Schiex, 2003]	$O(ed^2)$	$O(ed)$
FDAC* [Larrosa and Schiex, 2003]	$O(end^3)$	$O(ed)$
EDAC* [de Givry et al., 2005]	$O(ed^2 \max(nd, k))$	$O(ed)$
VAC_ε [Cooper et al., 2008]	$O(ed^2k/\varepsilon)$	$O(ed)$
OSAC [Cooper et al., 2008]	$poly(ed + n)$	$poly(ed^2 + nd)$

TABLE 2.4 – La complexité dans le pire cas des algorithmes de cohérence d'arc dans le cadre WCSP

différentes cohérences souples que nous avons présentées dans ce mémoire en se basant sur les deux relations d'implication et de $w_\emptyset$-supériorité définies précédemment. Pour les cohérences qui exigent un ordre total sur les variables, nous choisissons arbitrairement un ordre que nous fixons dès le départ de la comparaison. Notons qu'il existe un premier lien entre la relation d'implication et la relation de $w_\emptyset$-supériorité qui est donné par la proposition suivante :

Proposition 1. *Soient deux cohérences ϕ et ψ. Si ϕ implique ψ alors ϕ est $w_\emptyset$-supérieur à ψ.*

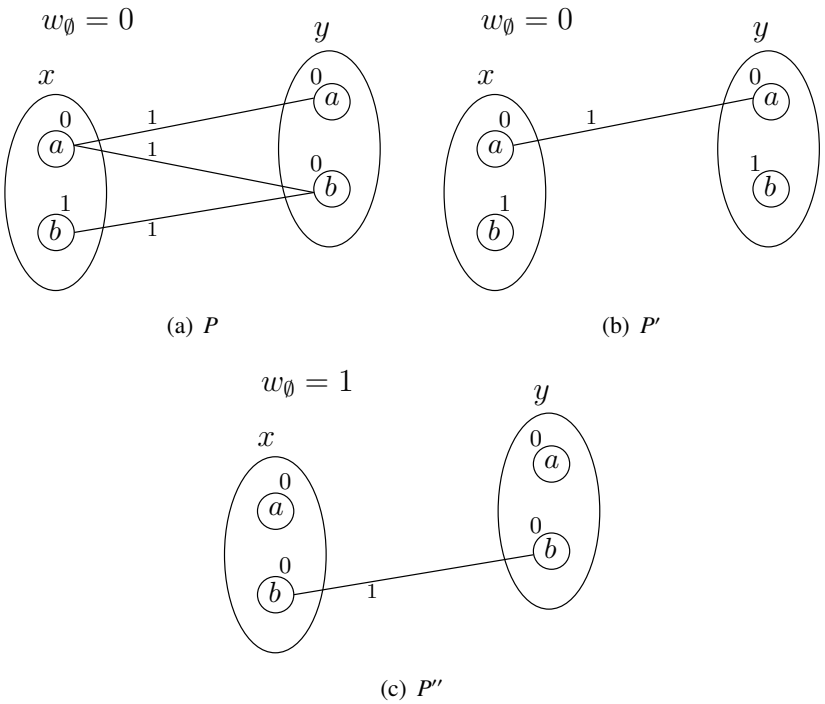
Preuve. Si une cohérence ϕ implique ψ alors tout WCN P ϕ-cohérent est aussi ψ-cohérent. Aussi $\psi(P) = P$ et $w_\emptyset(\psi(P)) = w_\emptyset(P)$. On fait l'hypothèse raisonnable qu'une cohérence ϕ appliquée à un réseau ϕ-cohérent ne modifie pas celui-ci. On en conclut que ϕ est $w_\emptyset$-supérieur à ψ. □

Cette proposition nous sera utile par la suite pour établir le lien entre certaines propriétés de cohérence.

2.3.11.1 AC* vs DAC*

Dans cette section, nous étudions la relation entre les deux propriétés AC* et DAC* présentées précédemment. Tout d'abord, nous comparons la borne inférieure $w_\emptyset$ lorsqu'on établit les deux propriétés, et nous montrons que AC* est $w_\emptyset$-incomparable à DAC*. Ensuite, nous étudions la relation d'implication entre ces deux propriétés.

Notons que l'établissement des propriétés AC* et DAC* n'est pas confluent, ce qui signifie entre autre ici qu'on peut obtenir plusieurs bornes inférieures $w_\emptyset$ selon l'ordre des opérations de transferts de coûts. Par exemple, considérons le WCN P donné par la figure 2.18. Sur cette figure, il est clair que P n'est pas AC*. Pour établir AC* nous avons le choix entre deux modes opératoires. Le premier mode opératoire consiste à commencer par établir des supports simples pour les valeurs de la variable y, et ensuite pour les valeurs de la variable x. Après l'établissement de la propriété AC*, nous obtenons le WCN P' avec une borne inférieure $w_\emptyset = 0$. Le second mode opératoire consiste à utiliser l'ordre inverse : nous établissons des supports pour les valeurs de la variable x et ensuite pour les valeurs de la variables y. Après l'établissement de la propriété AC*, nous obtenons cette fois le WCN P'' avec une borne



(a) P (b) P'

(c) P''

FIGURE 2.18 – L'établissement de la propriété AC* n'est pas confluent

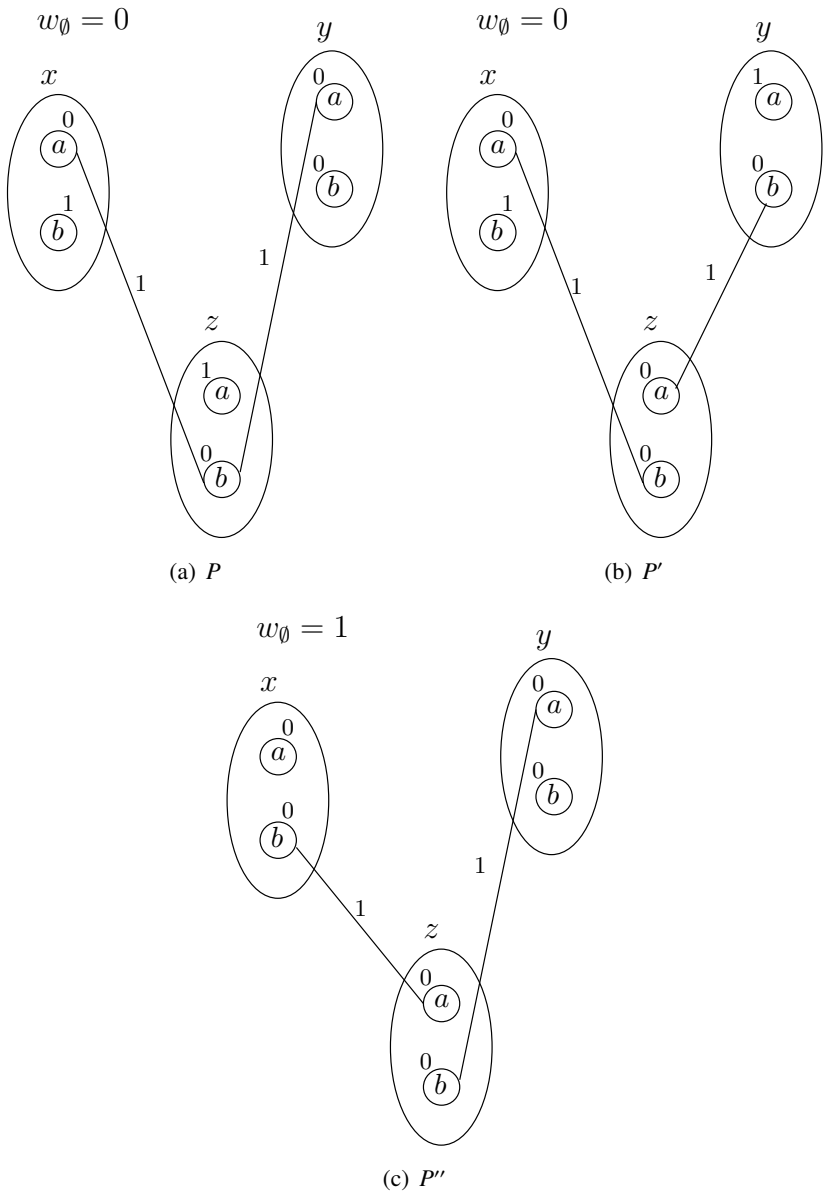
inférieure $w_\emptyset = 1$ qui est strictement supérieure à ce que nous avons obtenu auparavant. Cet exemple prouve que l'établissement de la propriété AC* n'est pas confluent.

De même, l'établissement de la propriété DAC* n'est pas confluent. En effet, considérons l'exemple du WCN P de la figure 2.19. Il est clair que P n'est pas DAC* selon l'ordre $x < y < z$. Si nous commençons d'abord par établir les supports complets pour les valeurs de la variable y sur la contrainte w_{yz}, nous obtenons le WCN P' qui est DAC*-cohérent et équivalent à P avec une borne inférieure $w_\emptyset = 0$. En revanche, si nous établissons d'abord les supports complets pour les valeurs de la variable x sur la contrainte w_{xz}, nous obtenons cette fois le WCN P'' qui est DAC*-cohérent et équivalent avec une borne inférieure $w_\emptyset = 1$. On en conclut donc que l'établissement de la propriété DAC* n'est pas confluent.

Dans la sous-section 2.3.6, nous avons indiqué avec un exemple que d'une part DAC* n'implique pas AC* et d'autre part AC* n'implique pas DAC*. Nous reprenons le même exemple pour montrer le lien de $w_\emptyset$-incomparabilité qui existe ente ces deux propriétés.

Nous remarquons que d'une part, l'exemple du WCN P de la figure 2.20 avec l'ordre $z < x < y$ est bien DAC* mais pas AC*. L'établissement de la propriété AC* sur P le transforme en un autre WCN P' équivalent et AC* avec une nouvelle borne plus grande que la première (voir la sous-section 2.3.6 pour les détails de transformation de P en P'). Cette première constatation permet de conclure que la propriété DAC* n'est pas $w_\emptyset$-supérieure à AC*. D'une autre part, le WCN P' obtenu n'est plus DAC*. L'établissement de DAC* le transforme en P'' équivalent et augmente la borne inférieure d'une unité de coût ce qui permet de conclure que la propriété AC* n'est pas $w_\emptyset$-supérieurte à DAC*.

Pour résumer, l'exemple précédent montre que les deux propriétés AC* et DAC* sont $w_\emptyset$-



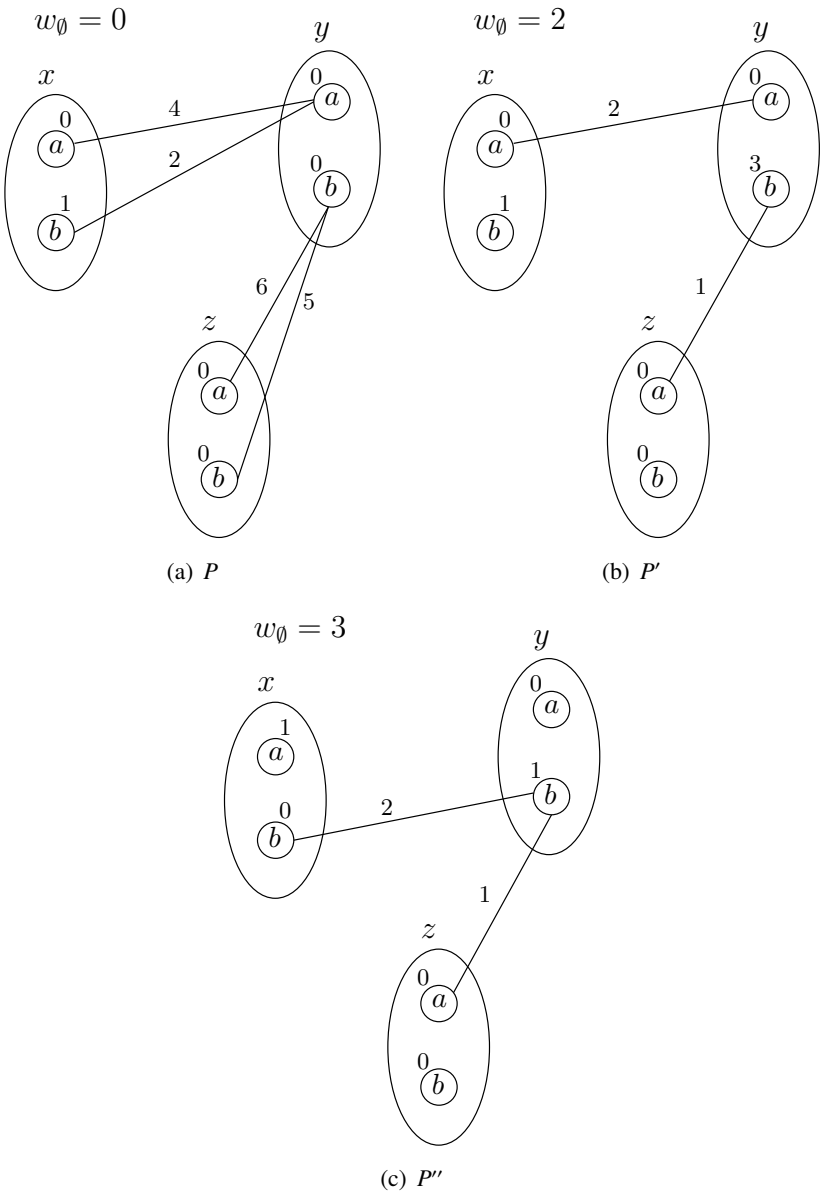
(a) P (b) P' (c) P''

FIGURE 2.19 – L'établissement de la propriété DAC* n'est pas confluent

incomparables et aucune des deux propriétés n'implique l'autre.

2.3.11.2 FDAC* vs AC* et DAC*

Établir la propriété FDAC* sur un WCN revient à établir simultanément les deux propriétés AC* et DAC*. Donc, tout WCN FDAC*-cohérent est à la fois AC*-cohérent et DAC*-cohérent. On en conclut que FDAC* implique AC* et DAC*. Selon la proposition 1, la propriété FDAC* est $w_\emptyset$-supérieure aux propriétés AC* et DAC*.



(a) P (b) P'

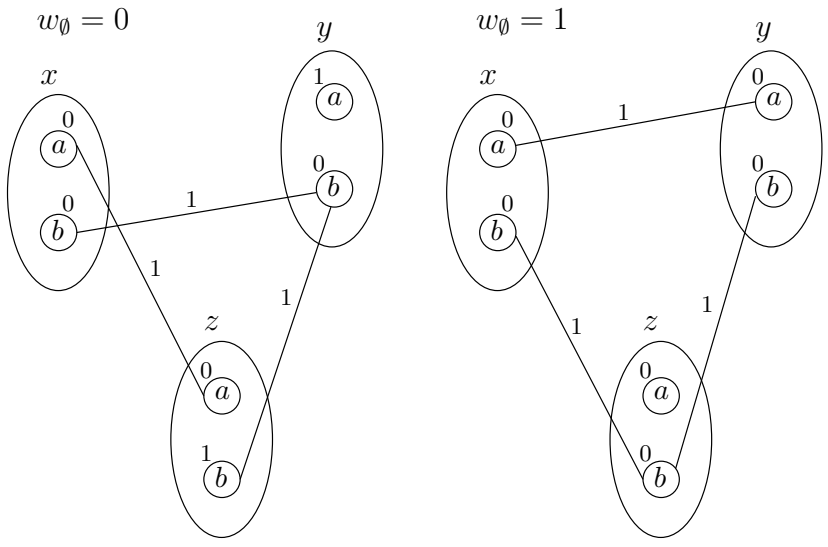
(c) P''

FIGURE 2.20 – Comparaison entre AC* et DAC*

2.3.11.3 FDAC* vs EAC

Dans la sous-section 2.3.7, nous avons indiqué qu'aucune des deux propriétés FDAC* et EAC n'implique l'autre. Pour montrer le lien de $w_\emptyset$-supériorité qui existe ente ces deux propriétés, nous reprenons l'exemple du WCN P avec l'ordre $x > z > y$ de la sous-section 2.3.7.

Sur la figure 2.21, il est clair que P est FDAC*-cohérent. Cependant, il n'est pas EAC-cohérent et l'établissement de la propriété EAC sur P augmente la borne inférieure d'une unité de coût et le transforme en un autre WCN P' équivalent. Cet exemple indique clairement que FDAC* n'est pas $w_\emptyset$-supérieur à EAC. De même, comme le montre le WCN P de la figure 2.22



(a) P (b) P'

FIGURE 2.21 – FDAC* n'est pas $w_\emptyset$-supérieur à EAC

avec l'ordre $w < x < y < z$, EAC n'est pas $w_\emptyset$-supérieur à FDAC*.

En effet, le WCN P donné par la figure 2.22 est clairement EAC-cohérent. En revanche, il n'est pas FDAC*-cohérent. Après l'établissement de la propriété FDAC*, nous obtenons le WCN P' qui est à la fois FDAC* et EAC avec une nouvelle borne inférieure $w_\emptyset = 1$ plus grande que celle de P. En résumé, les deux exemples ci-dessus montrent que les deux propriétés FDAC* et EAC sont $w_\emptyset$-incomparables.

2.3.11.4 EDAC* vs FDAC* et EAC

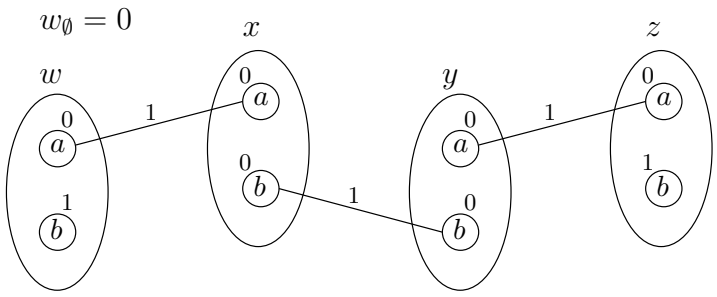
Établir la propriété EDAC* sur un WCN revient à établir simultanément les deux propriétés FDAC* et EAC. Donc, tout WCN EDAC*-cohérent est à la fois FDAC*-cohérent et EAC-cohérent. On en conclut donc que EDAC* implique FDAC* et EAC. Selon la proposition 1, la propriété EDAC* est $w_\emptyset$-supérieur aux propriétés FDAC* et EAC.

2.3.11.5 EDAC* vs VAC

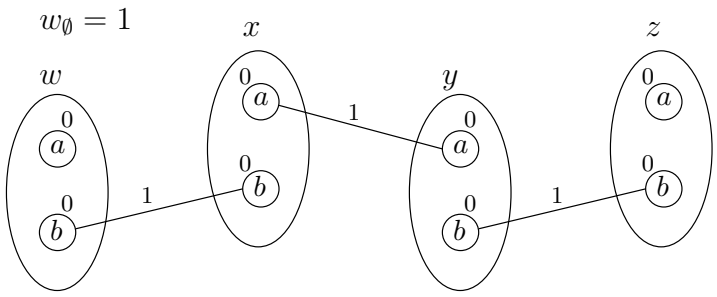
Dans [Cooper et al., 2010], les auteurs ont montré que la propriété VAC est plus forte que EDAC* en terme de borne inférieure. En revanche, sur l'exemple de la figure 2.23, on peut vérifier que ce problème est VAC-cohérent mais pas AC*-cohérent. Puisque AC* est la plus faible cohérence présentée jusqu'à maintenant (excepté NC*), la propriété VAC n'implique aucune des propriétés déjà vues dans ce chapitre.

2.3.11.6 OSAC vs VAC

Dans [Cooper et al., 2010], les auteurs ont montré qu'OSAC est $w_\emptyset$-supérieure à VAC et que pour tout WCN P si P est OSAC-cohérent alors il est VAC-cohérent, ce qui permet de



(a) P



(b) P'

FIGURE 2.22 – FDAC* vs EAC

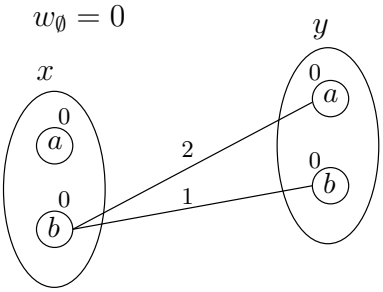


FIGURE 2.23 – VAC vs AC*

conclure que la propriété OSAC implique la propriété VAC.

Dans ce qui suit, nous résumons les deux relations d'implication et de $w_\emptyset$-supériorité sous forme d'un schéma donné par la figure 2.24.

Sur la figure 2.24, une relation d'implication entre deux cohérences ϕ et ψ telles que ϕ implique ψ est représentée par une flèche partant de ψ et pointant sur ϕ. La relation de $w_\emptyset$-supériorité est représentée par une hiérarchie entre les propriétés de cohérences, autrement dit, les propriétés de cohérences qui se trouvent dans les hauts niveaux sont $w_\emptyset$-supérieurs à celles qui se trouvent dans les bas niveaux. Enfin, deux propriétés ϕ et ψ qui sont $w_\emptyset$-incomparables sont reliées avec une ligne en pointillé.

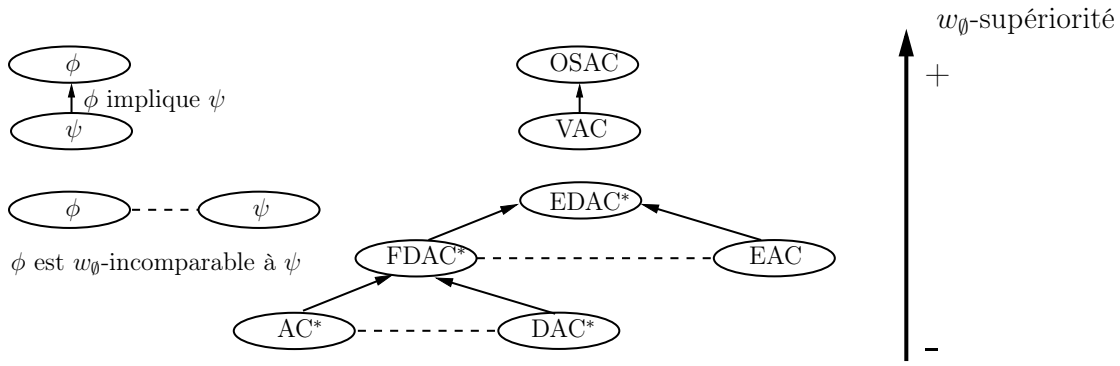


FIGURE 2.24 – Les relations entre les différents types de cohérence

Deuxième partie

Contributions

Chapitre 3

Substituabilité au voisinage pour le cadre WCSP

Sommaire

Dans ce chapitre, nous présentons la notion de substituabilité pour le cadre WCSP. Nous nous concentrons sur la substituabilité de voisinage souple (en anglais *SNS* pour Soft Neighbourhood Substitutability). Nous introduisons une propriété basée sur l'utilisation de paires de surcoût, appelée *pcost*, qui nous permet d'identifier efficacement des valeurs qui sont *souples-substituables*. Nous montrons que sous certaines hypothèses, *pcost* est équivalent à SNS. Cependant, dans le cas général, lorsque k qui est la valeur qui représente le niveau des coûts interdits n'est pas $+\infty$, *pcost* est plus faible que SNS qui est par ailleurs démontré coNP-difficile. Nous étudions également les liaisons entre SNS et certaines propriétés connues de cohérence d'arc souple comme la cohérence d'arc existentielle directionnelle (EDAC) et la cohérence d'arc virtuelle (VAC). Nous démontrons que SNS préserve la propriété VAC, mais pas nécessairement la propriété EDAC. Enfin, nous développons un algorithme pour *pcost* qui bénéficie d'une complexité en temps raisonnable, et nous montrons expérimentalement qu'il peut être associé à EDAC avec succès au cours de la recherche.

3.1 Définitions et notations

Dans cette section, nous introduisons quelques définitions et notations que nous utiliserons dans ce chapitre. À partir de maintenant, nous considérons un WCN $P = (\mathscr{X}, \mathscr{W}, k)$ donné.

Définition 58. *Pour toute instanciation I, nous désignons par $I_{x=a}$ l'instanciation $I[vars(I) \setminus x] \cup \{(x,a)\}$, qui est obtenue à partir de l'instanciation I soit par la substitution de la valeur affectée à x dans I par a, soit par l'extension de I avec (x,a).*

Par exemple, étant données deux instanciations $I = \{(x, b), (y, b), (z, b)\}$ et $I' = \{(y, b), (z, b)\}$, $I_{x=a} = I'_{x=a} = \{(x, a), (y, b), (z, b)\}$.

Définition 59 (Voisinage d'une variable)**.** *Pour toute variable x, l'ensemble des contraintes voisines de x est désigné par* $\Gamma(x) = \{w_S \in \mathscr{W} \mid x \in S\}$.

Quand $\Gamma(x)$ ne contient pas deux contraintes partageant au moins deux variables, on dit que $\Gamma(x)$ est *séparable*. Pour illustrer le concept, considérons l'exemple d'un WCN avec deux contraintes w_{txy} et w_{xyz}.

Le voisinage des variables est défini comme suit :

- $\Gamma(t) = \{w_{txy}\}$
- $\Gamma(z) = \{w_{xyz}\}$
- $\Gamma(x) = \Gamma(y) = \{w_{txy}, w_{xyz}\}$

Il est clair que $\Gamma(t)$ et $\Gamma(z)$ sont bien séparables. En revanche, $\Gamma(x)$ et $\Gamma(y)$ ne sont pas séparables. En effet, $\Gamma(x)$ et $\Gamma(y)$ contiennent deux contraintes qui partagent plus d'une variable (plus exactement les deux variables x et y). Dans un voisinage séparable, on peut construire une instanciation qui donne le coût minimal à chaque contrainte de ce voisinage. Ce n'est pas possible en général dans un voisinage non séparable.

Définition 60 (Coût sur un voisinage)**.** *Pour toute instanciation I et toute variable x, le coût de I sur le voisinage de x est désigné par* $cost_{\Gamma(x)} = \bigoplus_{w_S \in \Gamma(x)} w_S(I)$.

En d'autres termes, le coût de I sur un voisinage $\Gamma(x)$ est calculé en considérant uniquement les contraintes w_S qui appartiennent à $\Gamma(x)$.

3.2 La substituabilité dans le cadre CSP

Dans cette section, on considère un CN $P = (\mathscr{X}, \mathscr{C})$. L'interchangeabilité est une propriété générale des réseaux de contraintes introduite par Freuder [Freuder, 1991].

Définition 61 (Interchangeabilité complète)**.** *Deux valeurs a et b pour une variable x sont* interchangeables *si :*

- *pour toute solution I dans laquelle x = a,* $I_{x=b}$ *est également une solution.*
- *pour toute solution I dans laquelle x = b,* $I_{x=a}$ *est également une solution.*

En d'autres termes, la seule différence entre l'ensemble de solutions impliquant la valeur (x, a) et l'ensemble de solutions impliquant la valeur (x, b) est uniquement la valeur affectée à x. Dans ce cas, on peut remplacer un ensemble de valeurs entièrement interchangeables avec un seul représentant de l'ensemble sans perdre l'équisatisfiabilité. Par exemple, considérons l'exemple du CN P de la figure 3.1. Clairement, les deux valeurs (x, a) et (x, b) sont interchangeables. En effet, l'ensemble des solutions impliquant la valeur (x, a) noté $S_{x=a}$ est $\{S_1, S_2\}$ tel que $S_1 = \{(x, a), (y, b), (z, a)\}$ et $S_2 = \{(x, a), (y, b), (z, b)\}$. Par ailleurs, l'ensemble des solutions impliquant la valeur (x, b) noté $S_{x=b}$ est $\{S_3, S_4\}$ tel que $S_3 = \{(x, b), (y, b), (z, a)\}$ et $S_4 = \{(x, b), (y, b), (z, b)\}$. Nous remarquons immédiatement que l'ensemble $S_{x=a}$ est identique à l'ensemble $S_{x=b}$ si nous remplaçons (x, a) par (x, b) ce qui permet de conclure que (x, a) est interchangeable à (x, b). L'interchangeabilité (complète) a été relaxée sous plusieurs

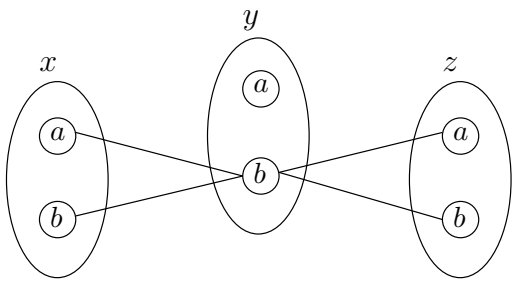


FIGURE 3.1 – (x, a) et (x, b) sont interchangeables

formes telles que l'interchangeabilité de voisinage, la k-interchangeabilité, la substituabilité et la substituabilité de voisinage.

Définition 62 (Interchangeabilité au voisinage)**.** *Deux valeurs a et b pour une variable x sont* interchangeables au voisinage *si pour toute contrainte c_S telle que $x \in S$:*

- *pour tout tuple $t \in l(S)$ tel que $t[x] = a$ satisfaisant la contrainte c_S, $t_{x=b}$ satisfait également la contrainte c_S.*
- *pour tout tuple $t \in l(S)$ tel que $t[x] = b$ satisfaisant la contrainte c_S, $t_{x=a}$ satisfait également la contrainte c_S.*

Autrement dit, deux valeurs (x, a) et (x, b) sont interchangeables au voisinage si pour toute contrainte c_S impliquant la variable x et pour tout tuple t impliquant la valeur (x, a) on a $c_S(t) \Leftrightarrow c_S(t_{x=b})$. L'interchangeabilité de voisinage est une forme limitée de l'interchangeabilité complète où seules les contraintes impliquant une variable donnée sont considérées.

Définition 63 (k-interchangeabilité)**.** *On dit que deux valeurs a et b pour une variable x sont k-interchangeables si et seulement si a et b sont interchangeables dans tout sous-problème de P contenant la variable x et $k - 1$ autres variables.*

Notons que pour le cadre des réseaux binaires, la 2-interchangeabilité est équivalente à l'interchangeabilité au voisinage et la n-interchangeabilité est équivalente à l'interchangeabilité complète, et pour $i < j$ la i-interchangeabilité est une condition nécessaire mais pas suffisante pour la j-interchangeabilité [Freuder, 1991]. Pour illustrer ce concept, considérons l'exemple du CN de la figure 3.2. La valeur (w, a) n'est pas 2-interchangeable avec (w, b) car dans le sous-problème P' de la figure 3.2 obtenu en ne conservant que w et la variable x, les deux valeurs (w, a) et (w, b) ne sont pas interchangeables (idem avec y et z). En revanche, ces deux valeurs sont 3-interchangeables car elles sont interchangeables dans le sous problème P'' de la figure 3.2 obtenu en conservant w et les variables x et y (idem avec d'autres paires de variables). Par conséquent, la valeur (w, a) est 4-interchangeable avec la valeur (w, b).

Définition 64 (Substituabilité complète)**.** *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$. (x, a) est* substituable *à (x, b) dans P si et seulement si pour toute solution I dans laquelle $x = b$, $I_{x=a}$ est également une solution.*

Dans ce cas, la valeur (x, a) peut remplacer la valeur (x, b) et on n'a pas besoin de chercher de solution avec $x = b$ si on ne s'intéresse qu'a prouver la satisfiabilité. Donc, le domaine de

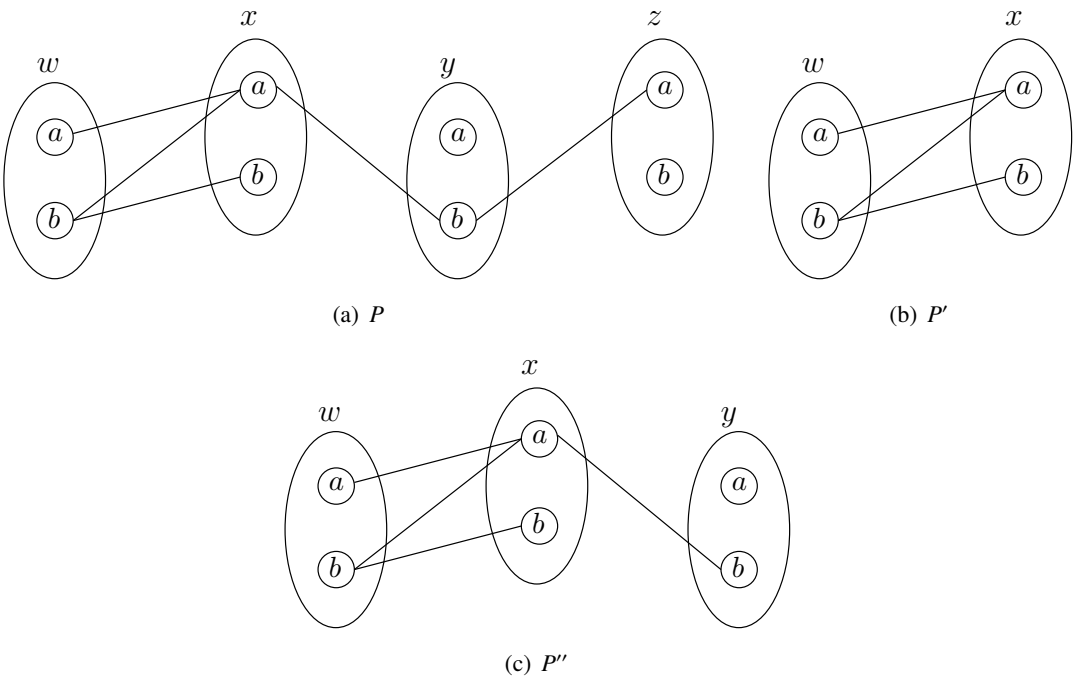


FIGURE 3.2 – (w, a) et (w, b) sont 3−interchangeables et 4−interchangeables mais pas 2−interchangeables.

la variable x peut être réduit en supprimant la valeur (x, b). Notons que l'interchangeabilité complète implique la substituabilité complète mais l'inverse est faux. En effet, si deux valeurs (x, a) et (x, b) sont interchangeables alors (x, a) est substituable à (x, b) et (x, b) est substituable à (x, a).

Définition 65 (Substituabilité au voisinage)**.** *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$. (x, a) est* substituable au voisinage *à (x, b) dans P si et seulement si pour toute contrainte c_S telle que $x \in S$ et pour tout tuple $t \in l(S)$ tel que $t[x] = b$, on a $c_S(t) \Rightarrow c_S(t_{x=a})$.*

La substituabilité au voisinage est une forme limitée de la substituabilité complète. Par exemple, considérons le CN de la figure 3.3. Nous remarquons que pour tout tuple t tel que $t[x] = b$ satisfaisant les deux contraintes c_{xy} et c_{xz}, le tuple $t_{x=a}$ satisfait également les deux contraintes c_{xy} et c_{xz}. Donc la valeur (x, a) est substituable au voisinage à (x, b). Notons que la valeur (x, a) n'est pas interchangeable avec (x, b) car pour le tuple $t = \{(x, a), (y, a)\}$, $c_{xy}(t) \nLeftrightarrow c_{xy}(t_{x=b})$.

L'interchangeabilité et la substituabilité ont été utilisées dans de nombreux contextes ; voir par exemple [Benson and Freuder, 1992; Haselbock, 1993; Choueiry et al., 1995; Freuder and Sabin, 1997; Bellicha et al., 1994; Cooper, 1997; Petcu and Faltings, 2003; Boussemart et al., 2004b; Lal et al., 2005]. Une taxonomie partielle de ces deux propriétés peut être trouvée dans [Karakashian et al., 2010].

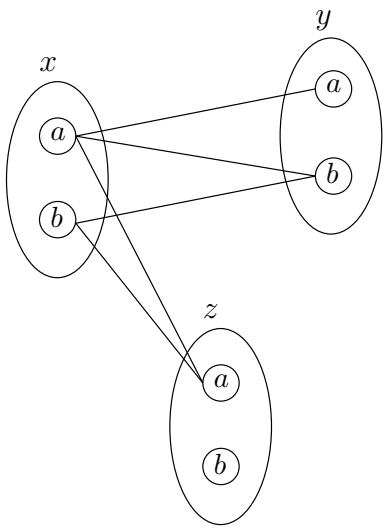


FIGURE 3.3 – (x, a) est substituable au voisinage à (x, b)

3.3 La substituabilité souple (cadre WCSP)

Dans cette section, nous introduisons la substituabilité souple (au voisinage) [Bistarelli et al., 2002] pour le cadre WCSP. Notons qu'une notion similaire à la substituabilité connue sous l'appellation d'*élimination par impasse* (en anglais, *DEE* pour Dead-End Elimination) a été introduite dans le cadre de la biologie computationnelle comme une technique de prétraitement pour réduire l'espace de recherche [Desmet et al., 1992; Goldstein, 1994; Dahiyat and Mayo, 1996; Pierce et al., 2000; Looger and Hellinga, 2001; Georgiev et al., 2006a]

Définition 66. *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$. (x, a) est* souple-substituable *à (x, b) dans P si et seulement si pour toute instanciation complète I de P, $cost(I_{x=a}) \leq cost(I_{x=b})$.*

Lorsque (x, a) est souple-substituable à (x, b), b peut être supprimé de $dom(x)$ sans changer le coût optimum de P. En effet, si elles existent, les solutions optimales possibles de P avec (x, b) sont perdues, mais il est garanti qu'il reste au moins une solution optimale avec (x, a).

Une forme réduite de la substituabilité souple pour le cadre WCSP appelé *substituabilité au voisinage* (en anglais *SNS* pour Soft Neighbourhood Substitutability) est introduite dans [Bistarelli et al., 2002]. Elle est définie comme suit :

Définition 67. *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$, (x, a) est souple-substituable à (x, b)* au voisinage *dans P si et seulement si pour chaque instanciation complète I de P, $cost_{\Gamma(x)}(I_{x=a}) \leq cost_{\Gamma(x)}(I_{x=b})$.*

Notons que dans la définition 67, le calcul de coût se fait uniquement dans le voisinage $\Gamma(x)$. On dira que (x, b) est SNS-éliminable (dans P) quand il existe une valeur (x, a) souple-substituable à (x, b) au voisinage. La substituabilité souple au voisinage implique la substituabilité souple (complète) (mais l'inverse n'est pas vrai). Il est particulièrement intéressant de noter que la substituabilité souple au voisinage autorise une compensation de coûts entre contraintes souples. Une telle compensation est rendue possible par la présence

de tous les coûts intermédiaires dans la structure d'évaluation, c'est-à-dire, les coûts différents de 0 et de k. Par exemple, considérons un WCN composé de trois variables x, y et z avec $dom(x) = dom(y) = dom(z) = \{a, b\}$ et deux contraintes w_{xy} et w_{xz} telles que $w_{xy}(a, a) = w_{xz}(b, a) = w_{xz}(b, b) = 1$; tous les autres coûts étant à 0. Une illustration de ce WCN est donnée par la figure 3.4. Comme d'habitude les coûts nuls ne sont pas représentés. Notons que (x, a) est souple-substituable à (x, b) au voisinage grâce à la compensation des coûts. En effet, si nous considérons uniquement la contrainte w_{xy}, on ne peut pas conclure que (x, a) est souple-substituable à (x, b) car il existe une instanciation $I = \{(x, a), (y, a)\}$ tel que $cost_{\Gamma(x)}(I_{x=a}) > cost_{\Gamma(x)}(I_{x=b})$. En revanche, la deuxième contrainte w_{xz} du WCN va permettre de compenser le sur-coût de la valeur (x, a) par rapport à la valeur (x, b) sur la contrainte w_{xy}. Enfin, pour le réseau global nous remarquons que toute instanciation complète I, $cost_{\Gamma(x)}(I_{x=a}) = cost_{\Gamma(x)}(I_{x=b})$.

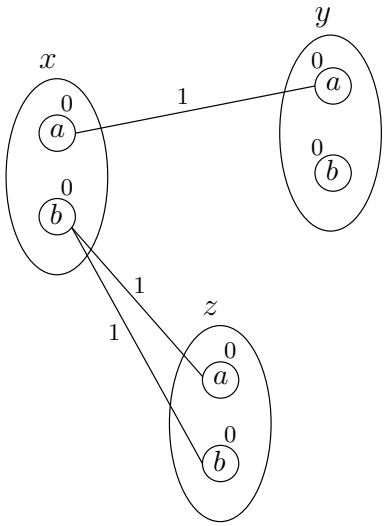


FIGURE 3.4 – (x, a) est souple-substituable à (x, b) au voisinage

La définition 67 nécessite de considérer chaque instanciation de $vars(\Gamma(x))$, ce qui reste très coûteux. La considération de chaque contrainte prise individuellement permettrait de réduire ce coût, mais dans ce cas, il est nécessaire de pouvoir identifier les compensations de coûts entre les différentes contraintes. Une façon simple de le faire est de calculer une somme de surcoûts minimaux, c'est à dire, une somme de différences de coût minimales sur toutes les contraintes de voisinage : $ocost(x : a \rightarrow b) = \sum_{w_S \in \Gamma(x)} min_{I \in l(S)}\{w_S(I_{x=b}) - w_S(I_{x=a})\}$; (en anglais *ocost* pour overcost), tel que mentionné dans [Koster, 1999; de Givry, 2004]. $ocost(x : a \rightarrow b)$ représente le surcoût de la valeur (x, b) par rapport à la valeur (x, a) sur l'ensemble des contraintes voisines à x. Si la valeur de $ocost(x : a \rightarrow b)$ est supérieure ou égale à 0 alors on peut déduire que (x, a) est souple-substituable à (x, b) au voisinage. Notons que *ocost* utilise le $-$ usuel pour calculer le sur-coût d'une valeur par rapport à une autre. Il est impossible d'utiliser directement l'opérateur $\ominus$ de la structure $\mathscr{S}(k)$. Rappelons que l'opération $u \ominus v$ est définie si et seulement si $u \geq v$. Or dans *ocst*, $w_S(I_{x=b})$ n'est pas toujours supérieur ou égal à $w_S(I_{x=a})$ ce qui ne permet pas de définir *ocost* avec l'opérateur $\ominus$. Mal-

heureusement, cette soustraction de coûts pose des problèmes subtils quand $k \neq +\infty$. Ceci est illustré ci-dessous.

Exemple 6. *Considérons les deux familles de contraintes* $W_i = \{w_i \mid i \in 1..n\}$ *et* $W'_i = \{w'_i \mid i \in 1..n'\}$ *définies par :*

x	y_i	w_i
a	c	0
b	c	1

x	z_i	w'_i
a	d	1
b	d	0

Quand $n = k$ *et* $n' = k + 1$, (x, a) *et* (x, b) *sont interchangeables parce que les deux valeurs sont interdites. Cependant,* $\forall w_i \in W_i, w_i(I_{x=b}) - w_i(I_{x=a}) = 1$ *et* $\forall w'_i \in W'_i, w'_i(I_{x=b}) - w'_i(I_{x=a}) = -1$.

En considérant la somme des différences sur les contraintes de $\Gamma(x) = W_i \cup W'_i$*, la valeur résultante est* $k - k - 1 = -1$ *ce qui indiquerait que b a globalement un coût inférieur à celui de a, ce qui est faux puisque les deux valeurs a et b ont un coût de k. Pour identifier correctement ce cas de substituabilité souple lorsque* $k \neq +\infty$*, il est nécessaire d'utiliser un opérateur non commutatif, ce qui nous empêche d'utiliser l'opérateur* $-$ *usuel.*

3.4 Calcul de la substituabilité souple

Nous allons maintenant nous concentrer sur la substituabilité souple au voisinage, et plus précisément sur la complexité de l'identification des valeurs SNS-éliminables. Nous commençons par discuter de certains travaux connexes à notre démarche. Pour le cadre général VCSP, des algorithmes efficaces pour la substituabilité souple au voisinage existent [Bistarelli et al., 2002] lorsque l'opérateur d'agrégation de la structure d'évaluation VCSP est idempotent. Pour le cadre FCSP (Fuzzy CSP), la notion de substituabilité floue au voisinage est proposée dans [Cooper, 2003] : il est montré que les valeurs floues-substituables au voisinage peuvent être efficacement identifiées lorsque l'opérateur d'agrégation de la structure d'évaluation FCSP est strictement monotone ou lorsque il s'agit de l'opérateur max. Plus récemment, la possibilité de calculer des formes de dominance plus faibles que la substituabilité souple au voisinage a été proposé dans [Koster, 1999]. Cependant, aucune étude précise qualitative n'a été menée pour le cadre WCSP. C'est ce que nous proposons de faire maintenant.

Tout d'abord, nous introduisons les paires de surcoût sur lesquelles notre mécanisme de calcul se base (ceci peut aussi être relié à ce qui a été proposé dans [Cooper, 2003] pour le cadre FCSP). En effet, une manière de contourner les problèmes de soustraction mentionnés ci-dessus est d'utiliser seulement l'addition sur des paires de surcoût et les comparer uniquement à la fin. Cette méthode est analogue à la construction d'entiers comme classes d'équivalence de paires ordonnées de nombres naturels où une paire (β, α) représente l'entier $\beta - \alpha$. Nous définissons + (addition) sur les paires de surcoût par $(\beta, \alpha) + (\beta', \alpha') = (\beta + \beta', \alpha + \alpha')$. Ceci est le + usuel et non $\oplus$, nous verrons un peu plus loin dans cette section (voir exemple 8) que l'utilisation de $\oplus$ dans les paires de coûts pour le calcul des valeurs substituables peut conduire à l'identification erroné de certaines de ces valeurs. Nous définissons aussi la comparaison des paires de surcoût avec 0 par $(\beta, \alpha) \geq 0 \Leftrightarrow \beta \geq \alpha$. Les paires sont ordonnées par la relation $\leq$ définie par $(\beta, \alpha) \leq (\beta', \alpha') \Leftrightarrow (\beta - \alpha < \beta' - \alpha') \vee (\beta - \alpha = \beta' - \alpha' \wedge \alpha < \alpha')$. En un sens,

la paire (β, α) porte deux informations : la différence $\beta - \alpha$ mais aussi $min(\beta, \alpha)$ qui est perdu lorsqu'on utilise une simple soustraction.

Définition 68. *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$,*

- *la* paire de surcoût *de (x, b) vis à vis de (x, a) dans $w_S \in \Gamma(x)$ est définie par :*
$$pcost(w_S, x : a \to b) = min_{I \in l(S)}\{(w_S(I_{x=b}), w_S(I_{x=a}))\};$$
- *la* paire de surcoût *de (x, b) vis à vis de (x, a) dans P est définie par :*
$$pcost(x : a \to b) = \sum_{w_S \in \Gamma(x)} pcost(w_S, x : a \to b).$$

Proposition 2. *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$. Si $pcost(x : a \to b) \geq 0$ alors (x, a) est souple-substituable à (x, b) au voisinage dans P.*

Preuve. Pour une contrainte w_S, soit I^{w_s} l'instanciation de $S - \{x\}$ qui donne la paire de surcoût minimale dans $min_{I \in l(S)}\{(w_S(I_{x=b}), w_S(I_{x=a}))\}$. Par définition, $pcost(w_S, x : a \to b) = (w_S(I^{w_S}_{x=b}), w_S(I^{w_S}_{x=a}))$. Par définition de min sur les paires de surcoût, nous avons $\forall I, \forall w_S \in \Gamma(x)$, $(w_S(I_{x=b}), w_S(I_{x=a})) \geq (w_S(I^{w_S}_{x=b}), w_S(I^{w_S}_{x=a}))$. Par sommation, nous obtenons $\forall I, \sum_{w_S \in \Gamma(x)}$ $(w_S(I_{x=b}), w_S(I_{x=a})) \geq \sum_{w_S \in \Gamma(x)}(w_S(I^{w_S}_{x=b}), w_S(I^{w_S}_{x=a}))$. Par hypothèse, $pcost(x : a \to b) \geq 0$, donc nous avons $\sum_{w_S \in \Gamma(x)}(w_S(I^{w_S}_{x=b}), w_S(I^{w_S}_{x=a})) \geq 0$, et donc $\forall I, \sum_{w_S \in \Gamma(x)}(w_S(I_{x=b}), w_S(I_{x=a})) \geq 0$. Par définitions de + et $\leq$ sur les paires de surcoût, nous pouvons tirer $\forall I, \sum_{w_S \in \Gamma(x)} w_S(I_{x=b}) \geq \sum_{w_S \in \Gamma(x)} w_S(I_{x=a})$ ce qui implique $\forall I, min(k, \sum_{w_S \in \Gamma(x)} w_S(I_{x=b})) \geq min(k, \sum_{w_S \in \Gamma(x)} w_S(I_{x=a}))$. Puisque $\forall a_i \in \{0, \ldots, k\}$, $a_1 \oplus \ldots \oplus a_n = min(k, a_1 + \ldots a_n\}$, nous pouvons conclure que $\forall I, \bigoplus_{w_S \in \Gamma(x)} w_S(I_{x=b})) \geq \bigoplus_{w_S \in \Gamma(x)} w_S(I_{x=a}))$. Ainsi, (x, a) est souple-substituable à (x, b) au voisinage dans P. □

La réciproque de la proposition 2 n'est pas vraie dans le cas général. Un premier cas où elle est fausse apparaît lorsque les portées de deux contraintes non binaires ont une intersection de plus d'une variable (voisinage non séparable). Dans cette situation, les contraintes ne peuvent pas être considérées individuellement.

Exemple 7. *Soient quatre variables x, y, z et t telles que $dom(x) = \{a, b\}$, $dom(y) = \{c, d\}$, $dom(z) = dom(t) = \{e\}$, et deux contraintes ternaires w_{xyz}, w_{xyt} définies par la table de coûts suivante :*

x	y	z	t	w_{xyz}	w_{xyt}
a	c	e	e	1	0
b	c	e	e	0	1
a	d	e	e	0	1
b	d	e	e	1	0

Il est clair que (x, a) est souple-substituable à (x, b) au voisinage. En effet, il est assez facile de constater que pour toute instanciation complète I tel que $I[x] = a$, $cost(I_{x=b}) = cost(I)$. En revanche, $pcost(w_{xyz}, x, a \to b) = pcost(w_{xyt}, x, a \to b) = (0, 1)$ et, par conséquent $pcost(x, a \to b) = (0, 2) \not\geq 0$.

Ainsi, une première condition pour que la réciproque de la proposition 2 tienne est que $\Gamma(x)$ soit séparable (ce qui est le cas des réseaux binaires normalisés). Un autre cas de figure où la réciproque de la proposition 2 est fausse est quand $k \neq +\infty$. Considérant le WCN de

l'exemple 6 avec $n = k$ et $n' = k + 1$, nous pouvons observer que $pcost(x : a \to b) = (n, n') = (k, k + 1) \not\geq 0$. Cependant, (x, a) et (x, b) sont tous deux interdits et par conséquent (x, a) est souple-substituable à (x, b) (et inversement). Dans cet exemple, il pourrait sembler une bonne idée d'utiliser $\oplus$ au lieu de $+$ pour l'addition de paires. Cependant, l'exemple 8 montre que cela conduirait à l'identification erronée de valeurs substituables.

Exemple 8. *Considérons la contrainte unaire w_x et la famille de contraintes binaires $W_i = \{w_i \mid i \in 1..n\}$ définie par :*

x	y_i	w_i
a	a	2
a	b	0
b	a	1
b	b	0

x	w_x
a	1
b	0

Clairement, (x, a) est non substituable à (x, b). Posons $n = k$. Avec la somme des paires définie par $+$, $pcost(x, a \to b) = (n, 2n + 1) \not\geq 0$. Si la somme de paires est définie par $\oplus$, nous obtiendrions $(k, k) \geq 0$.

Heureusement, il y a des situations où, même lorsque $k \neq +\infty$, l'utilisation des paires de surcoût nous permet d'identifier précisément l'ensemble des valeurs souples-substituables au voisinage.

Proposition 3. *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$ tel que $\Gamma(x)$ est séparable et $pcost(x, a \to b) = (\beta, \alpha)$ avec $\alpha < k$. Si (x, a) est souple-substituable à (x, b) au voisinage dans P alors $pcost(x : a \to b) \geq 0$.*

Preuve. Puisque par hypothèse $\Gamma(x)$ est séparable, nous pouvons définir l'instanciation I^{min} dans $vars(\Gamma(x)) \setminus \{x\}$ comme l'union pour chaque contrainte $w_s \in \Gamma(x)$ de l'instanciation I^{w_S} définie dans la preuve de la proposition 2. I^{min} est telle que $pcost(w_S, x : a \to b) = (w_S(I^{min}_{x=b}), w_S(I^{min}_{x=a}))$.

Par hypothèse, $\forall I, cost_{\Gamma(x)}(I_{x=b}) \geq cost_{\Gamma(x)}(I_{x=a})$, ce qui peut être réécrit sous la forme $\forall I, \bigoplus_{w_S \in \Gamma(x)} w_S(I_{x=b}) \geq \bigoplus_{w_S \in \Gamma(x)} w_S(I_{x=a})$. En particulier, cela est vrai pour $I = I^{min}$. Par conséquent, $\bigoplus_{w_S \in \Gamma(x)} w_S(I^{min}_{x=b}) \geq \bigoplus_{w_S \in \Gamma(x)} w_S(I^{min}_{x=a})$ qui peut être réécrit sous la forme $min(k, \beta) \geq min(k, \alpha)$ où $\beta = \sum_{w_S \in \Gamma(x)} w_S(I^{min}_{x=b})$ et $\alpha = \sum_{w_S \in \Gamma(x)} w_S(I^{min}_{x=a})$. Par définition, $pcost(x : a \to b) = (\beta, \alpha)$ et donc $pcost(x : a \to b) \geq 0$ si et seulement si $\beta \geq \alpha$. Maintenant, si $\alpha < k$, $min(k, \alpha) = \alpha$ et $min(k, \beta) \geq min(k, \alpha) \Rightarrow \beta \geq \alpha \Rightarrow pcost(x : a \to b) \geq 0$ (ceci est vrai pour $\beta < k$ et $\beta \geq k$). Notons que lorsque $\alpha \geq k$, $min(k, \beta) \geq min(k, \alpha) \not\Rightarrow \beta \geq \alpha$; un contre-exemple étant $\beta = k$ et $\alpha = k + 1$. □

Corollaire 1. *Soient $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$ tel que $\Gamma(x)$ est séparable et $pcost(x : a \to b) = (\beta, \alpha)$ avec $\alpha < k$. Si $(\beta, \alpha) < 0$ alors (x, a) n'est pas souple-substituable à (x, b) au voisinage dans P.*

Quand $pcost(x : a \to b) = (\beta, \alpha)$ avec $\alpha \geq k$, décider si (x, a) est souple-substituable à (x, b) au voisinage est beaucoup plus difficile. En effet, ce problème est coNP difficile. Pour prouver cela, nous introduisons le problème de double coût à choix multiple.

Le problème de choix multiple à double coût (MCDCP) Étant donnés m ensembles E_1, $E_2, \ldots, E_m$ d'objets tels que chaque objet $o_j \in E_i$ a une valeur de coût principale $r_{ij} \in \mathbb{Z}^+$ ainsi qu'une valeur de coût secondaire $s_{ij} \in \mathbb{Z}^+$. Étant donné un coût maximal $C \in \mathbb{Z}^+$, le problème MCDCP consiste à décider si il est possible de choisir un seul objet dans chaque ensemble de telle sorte que la somme des coûts principaux de ces objets sélectionnés ne dépasse pas C et ne dépasse pas également la somme des coûts secondaires. Ce problème peut être formulé comme suit :

$$\sum_{i=1}^{m} \sum_{j \in E_i} r_{ij} x_{ij} \leq C,$$
$$\sum_{i=1}^{m} \sum_{j \in E_i} r_{ij} x_{ij} \leq \sum_{i=1}^{m} \sum_{j \in E_i} s_{ij} x_{ij},$$
$$\sum_{j \in E_i} x_{ij} = 1, i = 1, \ldots, m,$$
$$x_{ij} \in \{0, 1\}, i = 1, \ldots, m, j \in E_i.$$

Proposition 4. *Le problème de choix multiple à double coût est NP-complet.*

Preuve. L'appartenance à NP est immédiat car il est possible de vérifier une solution efficacement en temps polynomial. Pour la NP-difficulté, nous réduisons le problème de sac à dos à choix multiple (MCKP pour Multiple-Choice Knapsack Problem) au problème de choix multiple à double coût. MCKP est connu pour être NP-difficile (par exemple, voir [Martello and Toth, 1990]). Pour MCKP, nous avons aussi m ensembles, et un profit $p_{ij} \in \mathbb{Z}^+$ est associé à chaque objet ainsi qu'un poids $w_{ij} \in \mathbb{Z}^+$. Étant donné un bénéfice minimal $P \in \mathbb{Z}^+$ et un poids maximal $W \in \mathbb{Z}^+$, le problème de décision MCKP est formulé comme suit :

$$\sum_{i=1}^{m} \sum_{j \in E_i} w_{ij} x_{ij} \leq W,$$
$$\sum_{i=1}^{m} \sum_{j \in E_i} p_{ij} x_{ij} \geq P,$$
$$\sum_{j \in E_i} x_{ij} = 1, i = 1, \ldots, m,$$
$$x_{ij} \in \{0, 1\}, i = 1, \ldots, m, j \in E_i.$$

Pour coder une instance MCKP en une instance MCDCP, nous conservons la même structure (ensembles) et définissons :

$$C = qW - mP,$$
$$r_{ij} = qw_{ij} - P, i = 1, \ldots, m, j \in E_i,$$
$$s_{ij} = mp_{ij} + r_{ij} - P, i = 1, \ldots, m, j \in E_i,$$

avec $q = 2mP$. Avec cette valeur choisie pour q, on peut montrer que toutes les valeurs C, r_{ij} et s_{ij} appartiennent à $\mathbb{Z}^+$. La première équation MCDCP peut être transformée comme suit :
$\sum_{i=1}^{m} \sum_{j \in E_i} r_{ij} x_{ij} \leq C$
$\Rightarrow \sum_{i=1}^{m} \sum_{j \in E_i} (qw_{ij} - P) x_{ij} \leq qW - mP$
$\Rightarrow \sum_{i=1}^{m} \sum_{j \in E_i} qw_{ij} x_{ij} - mP \leq qW - mP$ car exactement m variables x_{ij} sont assignées à 1,
$\Rightarrow \sum_{i=1}^{m} \sum_{j \in E_i} w_{ij} x_{ij} \leq W.$
La seconde équation MCDCP peut être transformée comme suit :
$\sum_{i=1}^{m} \sum_{j \in E_i} r_{ij} x_{ij} \leq \sum_{i=1}^{m} \sum_{j \in E_i} s_{ij} x_{ij},$
$\Rightarrow 0 \leq \sum_{i=1}^{m} \sum_{j \in E_i} (mp_{ij} - P) x_{ij}$ en simplifiant r_{ij} des deux côtés,
$\Rightarrow 0 \leq \sum_{i=1}^{m} \sum_{j \in E_i} mp_{ij} x_{ij} - mP$ car exactement m variables x_{ij} sont assignées à 1,
$\Rightarrow P \leq \sum_{i=1}^{m} \sum_{j \in E_i} p_{ij} x_{ij}.$ □

Proposition 5. *Décider si une valeur est souple-substituable au voisinage à une autre dans un WCN $(\mathscr{X}, \mathscr{C}, k)$ où $k \neq +\infty$ est coNP-difficile.*

Preuve. Toute instance MCDCP peut être réduite polynomialement au problème de décider si une valeur (x, a) n'est pas souple-substituable à une autre valeur (x, b) au voisinage dans un WCN P. À partir de l'instance MCDCP, nous construisons le WCN P comme suit :

- $\mathscr{X}$ contient une variable x tel que $dom(x) = \{a, b\}$ et une variable y_i pour chaque ensemble E_i ; le domaine de y_i contient les objects $o_{i1}, o_{i2}, \ldots$ de E_i.
- $\mathscr{W}$ contient exactement m contraintes binaires souple w_{xy_i} : nous avons $w_{xy_i}(\{(x, a), (y_i, o_{ij})\}) = s_{ij}$ et $w_{xy_i}(\{(x, b), (y_i, o_{ij})\}) = r_{ij}$.
- k est fixé à C.

Déterminer si (x, a) n'est pas souple-substituable à (x, b) au voisinage dans P est équivalent à trouver une instanciation I de $vars(\Gamma(x))$ telle que $cost_{\Gamma(x)}(I_{x=a}) > cost_{\Gamma(x)}(I_{x=b})$ qui est équivalent à $\sum_{w_S \in \Gamma(x)} w_S(I_{x=b}) < k \wedge \sum_{w_S \in \Gamma(x)} w_S(I_{x=a}) > \sum_{w_S \in \Gamma(x)} w_S(I_{x=b})$. La première (resp. seconde) condition encode la première (resp. seconde) inégalité de l'instance MCDCP. Comme les variables x_{ij} de l'instance MCDCP correspondent à l'affectation des variables y_i dans le WCN ($x_{ij} = 1 \Leftrightarrow y_i = o_{ij}$), la troisième équation de l'instance MCDCP est directement prise en compte dans le WCN. □

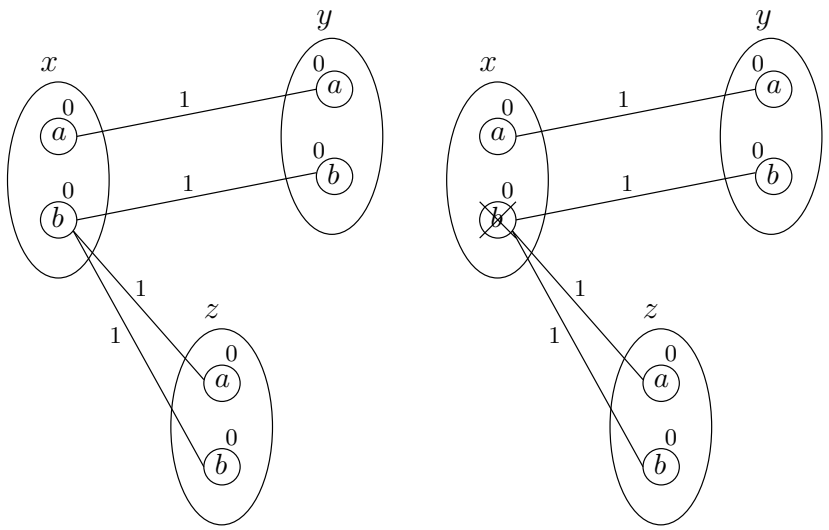
En pratique, il est plus facile de manipuler des différences de coûts en utilisant $ocost(x : a \to b) = \sum_{w_S \in \Gamma(x)} min_{I \in l(S)}\{w_S(I_{x=b}) - w_S(I_{x=a})\}$. En un sens, *pcost* est plus précis que *ocost* : quand $pcost(x : a \to b) = (\beta, \alpha)$ avec $\alpha < k$, (x, a) souple-substituable à (x, b) au voisinage est équivalent à $\beta \geq \alpha$, et quand $\alpha \geq k$, aucune conclusion ne peut être donnée. Avec *ocost*, l'information α est perdue. Par conséquent, lorsque $ocost(x : a \to b) < 0$, nous ne pouvons pas conclure avec certitude que (x, a) n'est pas souple-substituable à (x, b) au voisinage. Dans la pratique, cela ne fait aucune différence (tout du moins, si on considère nos développements algorithmiques actuels), ce qui est la raison pour laquelle les expériences dans le présent document ont été réalisées avec *ocost*.

3.5 Liaisons avec la cohérence d'arc souple

Après l'introduction de la fermeture par substituabilité souple au voisinage, cette section présente quelques résultats connectant la substituabilité souple au voisinage avec diverses formes de cohérence d'arc souple.

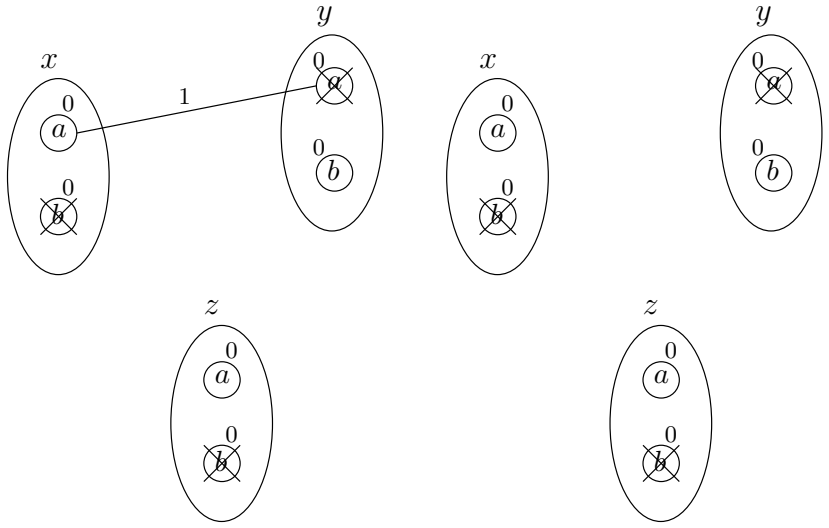
Définition 69. *La fermeture par substituabilité souple au voisinage (ou SNS-closure) d'un WCN P, noté $\texttt{SNS}(P)$, est tout WCN obtenu après l'élimination itérative des valeurs SNS-éliminables jusqu'à ce qu'un point fixe soit atteint.*

Par exemple, considérons le WCN P de la figure 3.5(a). Nous remarquons immédiatement que la valeur (x, a) est souple-substituable au voisinage à la valeur (x, b). Après l'élimination de la valeur (x, b), nous obtenons le WCN de la figure 3.5(b). Sur ce WCN nous remarquons aussi que la valeur (z, a) (resp. (y, b)) est souple-substituable au voisinage à la valeur (z, b) (resp. (y, a)). Après l'élimination de ces valeurs dites SNS-éliminables, nous obtenons le WCN de la figure 3.5(d) qui ne contient aucune valeur substituable souple au voisinage et qui représente la fermeture par substituabilité souple au voisinage du WCN original P. Puisque cette opération n'est pas confluente, $\texttt{SNS}(P)$ n'est pas unique. En effet, sur le WCN de la figure 3.5(b) les valeurs (z, a) et (z, b) sont interchangeable. Si nous éliminons la valeur (z, a) au



(a) P

(b) (x, a) est souple-substituable à (x, b) au voisinage



(c) (z, a) (resp. (y, b)) est souple-substituable au voisinage à (z, b) (resp. (y, a))

(d) P'

FIGURE 3.5 – La fermeture par substituabilité souple au voisinage du WCN P

lieu de la valeur (z, b) nous obtenons une autre fermeture par substituabilité souple au voisinage du WCN P. Une illustration de cette nouvelle fermeture est donnée par la figure 3.6

Quand nous utilisons l'approche *pcost* pour identifier des valeurs SNS-éliminables, on notera $\texttt{PSNS}(P)$.

Proposition 6. *Soit P un WCN EDAC-cohérent. $\texttt{SNS}(P)$ n'est pas nécessairement EDAC-cohérent.*

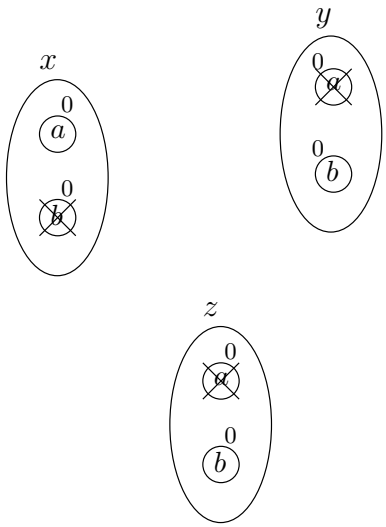


FIGURE 3.6 – Une autre fermeture par substituabilité souple au voisinage du WCN P de la figure 3.5(a)

Preuve. Considérons le WCN P représenté par la figure 3.7(a). Notons que P est EDAC-cohérent par rapport à l'ordre $w < x < z < y$, et que (w, a) et (x, a) sont respectivement souples-substituables à (w, b) et (z, b) au voisinage, puisque $pcost(w, a \rightarrow b) = (1, 1)$ et $pcost(x, a \rightarrow b) = (1, 1)$. Il existe une SNS-closure unique de P, $P' = \texttt{SNS}(P)$, qui est représentée par la figure 3.7(b). Il est clair que P', n'est pas EDAC-cohérent puisque (z, b) et (y, b) n'ont pas de support sur w_{yz}. □

À partir de maintenant, on peut définir une nouvelle propriété que nous appelons SNS+EDAC* qui consiste à éliminer en premier lieu toutes les valeurs SNS-éliminables et ensuite établir la propriété EDAC*. Dans ce qui suit, nous situons SNS+EDAC* par rapport aux autres propriétés de la figure 2.24.

Proposition 7. *SNS+EDAC* implique EDAC* et est également $w_\emptyset$-supérieur à EDAC*.*

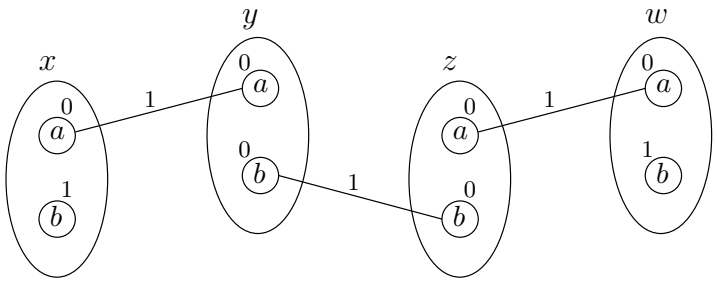
Preuve. Il est clair que tout WCN qui est SNS+EDAC* est forcément EDAC*. Mais l'inverse n'est pas toujours vrai (voir l'exemple de la figure 3.7). Pour tout WCN P EDAC*-cohérent, $\texttt{SNS}(P)$ n'est pas nécessairement EDAC*-cohérent (voir la proposition 6). D'où, établir à nouveau la propriété EDAC* sur $\texttt{SNS}(P)$ peut augmenter la borne inférieure $w_\emptyset$. En revanche, établir EDAC* sur tout WCN P qui est SNS+EDAC* ne permet jamais d'augmenter la borne inférieure $w_\emptyset$. □

Proposition 8. *Soit P un réseau qui est EDAC*. $\texttt{SNS}(P)$ n'est pas nécessairement AC*.*

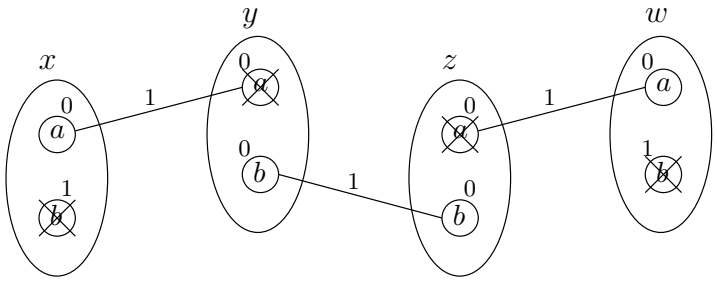
Preuve. Le WCN P représenté par la figure 3.7(a) est EDAC*-cohérent. La SNS-closure de P, $P' = \texttt{SNS}(P)$ (voir la figure 3.7(b)), ne vérifie pas AC*. □

Proposition 9. *Soit ϕ l'une des cohérences parmi AC*, DAC*, FDAC* et EAC. SNS+ϕ est $w_\emptyset$-supérieur à ϕ.*

Preuve. D'après la proposition 8, pour un WCN P ϕ-cohérent, $\texttt{SNS}(P)$ n'est pas nécessairement ϕ-cohérent. D'où, établir à nouveau la propriété ϕ sur $\texttt{SNS}(P)$ peut augmenter la borne inférieure $w_\emptyset$. En revanche, établir ϕ sur tout WCN P qui est SNS+ϕ ne permet jamais d'augmenter la borne inférieure $w_\emptyset$. □



(a) P



(b) P'

FIGURE 3.7 – EDAC versus SNS

Lemme 1. *Soient P un WCN VAC-cohérent, $x \in S$ et $\{a, b\} \subseteq dom(x)$ tel que (x, b) est AC-cohérent dans $Bool(P)$. Si $pcost(x : a \to b) \geq 0$ dans P alors (x, a) est substituable (au sens CSP [Freuder, 1991]) à (x, b) au voisinage dans $Bool(P)$.*

Preuve. Nous supposons que P est VAC-cohérent et (x, b) est AC-cohérent dans $Bool(P)$. Puisque (x, b) est AC-cohérent dans $Bool(P)$, nous savons que pour chaque contrainte $w_S \in \Gamma(x)$, il existe une instanciation I de S telle que $I[x] = b$ et $w_S(I) = 0$ (par construction de $Bool(P)$). Cela veut dire que pour chaque contrainte $w_S \in \Gamma(x)$, $pcost(w_S, x : a \to b) \leq 0$. Comme par hypothèse $pcost(x : a \to b) \geq 0$, pour chaque $w_S \in \Gamma(x)$, nous avons nécessairement $pcost(w_S, x : a \to b) = 0$. Nous pouvons déduire que pour chaque contrainte $w_S \in \Gamma(x)$, pour chaque instanciation I de S telle que $I[x] = b$ et $w_S(I) = 0$, l'instanciation $I' = I_{x=a}$ est telle que $w_S(I') = 0$. Pour finir, nous pouvons déduire que (x, a) est substituable à (x, b) au voisinage dans $Bool(P)$. □

Proposition 10. *Soient P un WCN VAC-cohérent, $x \in \mathscr{X}$ et $\{a, b\} \subseteq dom(x)$. Si $pcost(x : a \to b) \geq 0$ dans P alors $P \setminus \{(x, b)\}$ est VAC-cohérent.*

Preuve. Supposons (premier cas) pour commencer que (x, b) est AC-cohérent dans $Bool(P)$. D'après le lemme précédent, nous savons que (x, a) est substituable à (x, b) au voisinage dans

$Bool(P)$, et donc nous avons $AC(Bool(P)) \neq \bot \Leftrightarrow AC(Bool(P \setminus \{(x,b)\})) \neq \bot$. Puisque P est VAC-cohérent, nécessairement $P \setminus \{(x,b)\}$ est VAC-cohérent. Maintenant (deuxième cas), supposons que (x,b) n'est pas AC-cohérent dans $Bool(P)$. Il est clair que nous avons alors $AC(Bool(P)) = AC(Bool(P \setminus \{(x,b)\}))$, et donc $P \setminus \{(x,b)\}$ est VAC-cohérent (puisque P est VAC-cohérent). □

Corollaire 2. *Soit P un WCN. Si P est VAC-cohérent alors* SNS(P) *est VAC-cohérent.*

Il est à noter que les deux propriétés VAC et SNS+VAC se situent dans le même niveau hiérarchique de la $w_\emptyset$-supériorité car les deux propriétés fournissent la même borne inférieure $w_\emptyset$. Le corollaire et la note précédente sont également valables pour OSAC car OSAC implique VAC. Dans ce qui suit, nous rajoutons pour chaque propriété ϕ apparaissant sur la figure 2.24 la propriété SNS+ϕ en montrant le lien d'implication et de $w_\emptyset$-supériorité existant entre elles. Ces liens sont donnés par la figure 3.8.

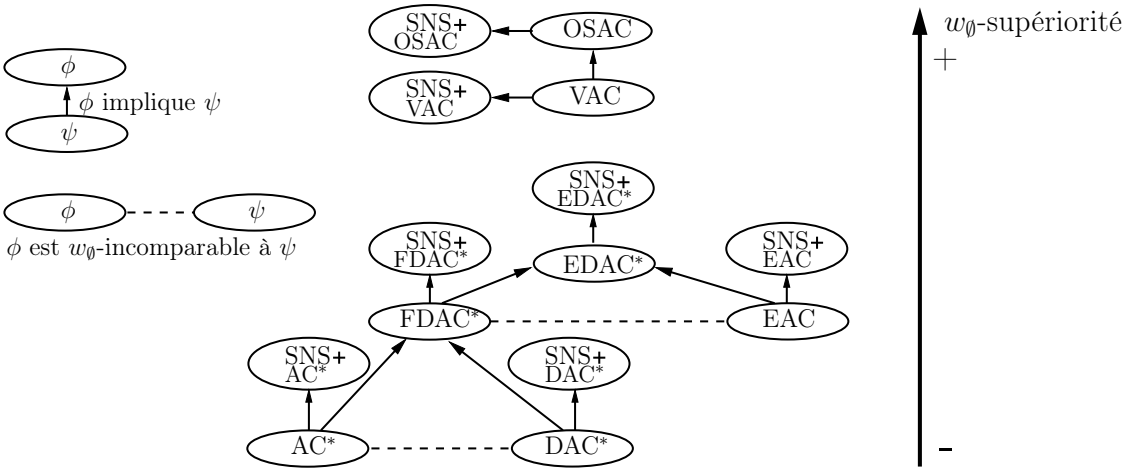


FIGURE 3.8 – Les relations entre les différents types de cohérence

3.6 Algorithme

Dans cette section, nous présentons un algorithme pour établir AC*+PSNS (qui peut être facilement adapté à EDAC*+PSNS, par exemple). L'idée directrice est de toujours commencer à identifier les valeurs SNS-éliminables à partir d'un WCN qui est AC*-cohérent. Cela nous permet de réduire l'effort de calcul en sortant de boucles avant leur fin et en utilisant des résidus.

La procédure principale est l'algorithme 19. Comme souvent, nous utilisons un ensemble, noté Q, pour stocker les variables dont le domaine a été récemment réduit. Au début, Q contient toutes les variables (ligne 4). Puis, à la ligne 6, un algorithme classique AC*, désigné ici par W-AC*, est exécuté (par exemple, cela peut être W-AC*2001 [Larrosa and Schiex, 2004]), avant de solliciter une fonction appelée PSNS^r. Les appels de W-AC* et de PSNS^r sont entrelacés jusqu'à ce qu'un point fixe soit atteint (i.e., $Q = \emptyset$).

Algorithme 16 : `pcost`

Entrées : w_S : Contrainte, x : Variable, a, b : Valeur
Sorties : Paire de surcoût

```
1  pcst ← (0, 1) ;
2  pour chaque I ∈ l(S \ {x}) faire
3  |  si (w_S(I_{x=b}), w_S(I_{x=a})) < pcst alors
4  |  |  pcst ← (w_S(I_{x=b}), w_S(I_{x=a})) ;
5  |  fin
6  fin
7  retourner pcst
```

Algorithme 17 : `pcost`

Entrées : x : Variable, a, b : Valeur
Sorties : Paire de surcoût

```
1  pcst ← (w_x(b), w_x(a)) ;
2  si pcst < 0 alors retourner pcst;
3  pcst ← pcst + pcost(residues[x, a, b], x, a → b) ;
4  si pcst < 0 alors retourner pcst;
5  pour chaque w_S ∈ Γ(x) | w_S ≠ residues[x, a, b] faire
6  |  d ← pcst(w_S, x, a → b) ;
7  |  si d < (w_x(a), w_x(b)) alors residues[x, a, b] ← w_S ;
8  |  pcst ← pcst + d ;
9  |  si pcst < 0 alors retourner pcst;
10 fin
11 retourner pcst;
```

La fonction PSNS^r, algorithme 18, itère sur toutes les variables afin de recueillir les valeurs SNS-éliminable dans un ensemble appelé Δ. Cet ensemble est initialisé à la ligne 1 et mis à jour aux lignes 6 et 8. Imaginons que toutes les valeurs SNS-éliminables (qui peuvent être identifiées avec le calcul de surcoûts) pour une variable x ont été supprimées, et que le domaine de toutes les variables dans le voisinage de x reste identique. Clairement, il n'est alors pas nécessaire de considérer à nouveau x pour rechercher des valeurs SNS-éliminables. C'est l'objet de la ligne 3. Ici, un mécanisme d'horodatage est utilisé. En introduisant un compteur global `time` et en associant un tampon temporel `stamp`$[x]$ à chaque variable x ainsi qu'un tampon `substamp` à la fonction PSNS^r, il est possible de déterminer quelles variables doivent être considérées. La valeur de `stamp`$[x]$ indique à quel moment une valeur a été récemment éliminée de $dom(x)$ tandis que la valeur de `substamp` indique à quel moment PSNS^r a été récemment appelé. Les variables `time`, `stamp`$[x]$ pour chaque variable x et `substamp` sont initialisées aux lignes 1 à 3 de l'algorithme 19. La valeur de `time` est incrémentée chaque fois qu'une variable est ajoutée à Q (ligne 17 de l'algorithme 18, et cela *doit* aussi être effectué au sein de W-AC*) et chaque fois que PSNS^r est appelé (ligne 13). Toutes les valeurs SNS-

Algorithme 18 : PSNS^r

Entrées : P : WCN AC*-consistent

```
1  Δ ← ∅ ;
2  pour chaque x ∈ 𝒳 faire
3      si ∃y ∈ vars(Γ(x)) | stamp[y] > substamp alors
4          pour chaque (a, b) ∈ dom(x)² | b > a faire
5              si pcost(x, a → b) ≥ 0 alors
6                  Δ ← Δ ∪ {(x, b)} ;
7              sinon
8                  si pcost(x, b → a) ≥ 0 alors Δ ← Δ ∪ {(x, a)} ;
9              fin
10         fin
11     fin
12 fin
13 substamp ← time++ ;
14 pour chaque (x, a) ∈ Δ faire
15     supprimer (x, a) de dom(x) ;
16     Q ← Q ∪ {x} ;
17     stamp[x] ← time++ ;
18 fin
```

Algorithme 19 : AC*-PSNS

Entrées : P : WCN

Sorties : P : WCN rendu AC*-cohérent et PSNS-clos

```
1  time ← 0 ;
2  substamp ← −1 ;
3  stamp[x] ← 0, ∀x ∈ 𝒳 ;
4  Q ← 𝒳 ;
5  répéter
6      PSNS^r(W-AC*(P, Q)) ;
7  jusqu'à Q ≠ ∅ ;
```

éliminables collectées dans Δ sont supprimées tandis que Q est mis à jour pour le prochain appel à W-AC* (lignes 15 et 16).

L'algorithme 17 nous permet de calculer la paire de surcoût *pcost* de (x, b) par rapport à (x, a). Puisque nous savons que le WCN est AC*-cohérent, nous avons la garantie que la paire de surcoût de (x, b) par rapport à (x, a) dans toute contrainte non-unaire w_S où $x \in S$ est inférieur ou égal à $(0, 0)$. Cela signifie que nous ne pourrons jamais compenser une paire de surcoût négative avec une une paire de surcoût positive (une fois que la paire de surcoût unaire a été pris en compte). Un grand avantage de cette observation est la possibilité d'utiliser les arrêts précoces de boucle au cours de tels calculs. Cette opération est effectuée aux lignes 2, 4

et 9. Les résidus sont un autre mécanisme introduit pour augmenter la performance de l'algorithme. Pour chaque variable x, et pour tout couple (a, b) de valeurs de $dom(x)$, nous stockons dans `residues`$[x, a, b]$ la contrainte w_S qui garantit que (x, b) n'est pas SNS-éliminable par (x, a), si elle existe. La contrainte résiduelle est prioritaire (lignes 3-4) ; de cette façon, si elle permet de compenser le surcoût initial unaire, elle nous évite tout travail supplémentaire. Elle est mise à jour aux lignes 7. Notons que nous pouvons initialiser le tableau `residues` avec n'importe quelle contrainte arbitraire et que l'algorithme 16 retourne nécessairement une paire de surcoût inférieure ou égale à $(0, 1)$ (ce qui explique l'initialisation de `pcst` à $(0, 1)$ à la ligne 1).

Nous discutons maintenant de la complexité de PSNS^r tout en faisant l'hypothèse (pour simplifier) que le WCN est binaire. La complexité en espace est $O(nd^2)$ en raison de l'utilisation de la structure `residues`. La complexité en temps de l'algorithme 16 est $O(d)$, et la complexité en temps de l'algorithme 17 est $O(1)$ dans le meilleur des cas (si il est arrêté à la ligne 3) et $O(qd)$ dans le pire des cas, où $q = |\Gamma(x)|$. Sans la ligne 3, la complexité en temps de l'algorithme 18 est $O(nd^2)$ dans le meilleur des cas et $O(d^3e)$ dans le pire des cas (à noter que $\sum_{x \in \mathcal{X}} |\Gamma(x)|$ est $O(e)$). Bien sûr, l'algorithme 18 peut être appelé à plusieurs reprises à la ligne 6 de l'algorithme 19, ainsi obtient-on une complexité dans le pire des cas en $O(nd^4e)$. Cependant, nous avons observé dans nos expérimentations que le nombre d'appels successifs à PSNS^r était très limité en pratique (comme on l'imaginait). En outre, imaginons maintenant que nous appelions AC*-PSNS après l'assignation d'une valeur à une variable (i.e. au cours de la recherche) x. Dans le meilleur des cas (du point de vue de la complexité temporelle), aucune suppression n'est effectué par *W-AC*$*$, et on considère donc uniquement le voisinage de x (à la ligne 3 de l'algorithme 18), ce qui donne une complexité en temps en $O(qd^2)$. Ce dernier résultat nous permet d'envisager relativement sereinement l'expérimentation du maintien de AC*-PSNS pendant la recherche.

3.7 Résultats expérimentaux

Pour démontrer l'intérêt pratique de la suppression des valeurs SNS-éliminables, nous avons mené une expérimentation en utilisant les séries d'instances WCSP disponibles à l'adresse `http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/BenchmarkS` et un cluster de Xeon 3.0GHz avec 2 Go de RAM sous Linux. Notre but est d'observer l'efficacité relative de la résolution d'instances WCSP lorsqu'on *maintient* AC*, AC*+PSNS, FDAC, FDAC+PSNS, EDAC, et EDAC+PSNS. Pour l'ordre de sélection des variables au cours de la recherche, nous utilisons l'heuristique statique simple *max degree* qui est indépendante de l'efficacité des algorithmes de filtrage utilisés, comme dans [de Givry, 2004] où certaines expériences ont été réalisées avec une forme partielle de SNS appliquée lors d'une étape de pré-traitement.

Le tableau 3.2 montre les moyennes des résultats obtenus sur les différentes séries. Pour chaque série, le nombre d'instances considérées (#inst) est donné au-dessous du nom de la série. Nous avons écarté les instances qui n'ont pas été résolues par au moins un des algorithmes, en moins de 1,200 secondes. Ici, une instance résolue signifie qu'une solution optimale a été trouvée et prouvée être optimale. Le nombre moyen (avg-sub) de valeurs SNS-éliminables supprimées lors de la recherche (à chaque étape) est également indiqué (avec des valeurs ar-

Series		AC*	AC* +PSNS	FDAC	FDAC +PSNS	EDAC	EDAC +PSNS
celar	#solved	5	4	6	6	6	**7**
#inst=7	cpu	337	231	316	341	344	**461**
	avg-sub	0	3	0	6	0	4
driver	#solved	18	18	19	**19**	19	19
#inst=19	cpu	103	52	39.3	**19.7**	68.5	56.8
	avg-sub	0	1	0	1	0	1
geom	#solved	5	5	5	**5**	5	5
#inst=5	cpu	37.4	12.4	18.8	**11.0**	21.0	12.0
	avg-sub	0	0	0	1	0	0
mprime	#solved	4	8	4	**8**	5	8
#inst=8	cpu	9.82	17.4	11.6	**12.0**	206	21.0
	avg-sub	0	1	0	1	0	1
myciel	#solved	3	3	3	**3**	3	3
#inst=3	cpu	122	77.0	72.9	**34.3**	80.4	37.8
	avg-sub	0	2	0	3	0	3
scens+graphs	#solved	1	3	**8**	7	6	5
#inst=9	cpu	20.4	113	**242**	186	266	19.8
	avg-sub	0	8	0	8	0	8
spot5	#solved	0	0	3	3	3	**3**
#inst=3	cpu			20.5	12.6	20.1	**11.4**
	avg-sub	0	0	0	0	0	0
warehouse	#solved	12	12	18	24	28	**29**
#inst=34	cpu	286	46.2	166	139	53.2	**79.8**
	avg-sub	0	4	0	7	0	27
	#solved	48	53	66	75	75	**79**

TABLE 3.1 – Résultats obtenus pour différentes séries (une échéance de 1,200 secondes par instance).

rondies à l'entier le plus proche). Sur les instances RLFAP (CELAR, scens, graphs), l'intérêt d'utiliser PSNS est plutôt chaotique, mais sur les instances (driver, mprime), coloring (myciel, geom), spot et warehouse, on peut constater qu'il y a un avantage clair à l'intégration de PSNS. Dans l'ensemble, le maintien de PSNS est rentable car il offre en général un avantage à la fois en termes d'instance résolues (voir dernière ligne du tableau) et en temps CPU. Le tableau 3.2 présente les résultats obtenus sur certaines instances représentatives. Il est intéressant de noter que sur les instances warehouse (ici, *cap101*, *cap111* et *capmo1*), l'application de PSNS n'entraîne pas une réduction de la taille de l'arbre de recherche (voir les valeurs de #nodes). Cependant, PSNS permet de réduire la taille des domaines, ce qui rend la propagation des contraintes souples plus rapide. Sur *celar7-sub1*, notons que (maintenir) EDAC+PSNS est seulement environ 50% plus rapide que (maintenir) EDAC alors que le nombre de nœuds a été

Instances		AC*	AC* +PSNS	FDAC	FDAC +PSNS	EDAC	EDAC +PSNS
cap101	cpu	232	42.1	1.6	1.62	1.48	**1.12**
	#nodes	242K	242K	835	835	75	75
	avg-sub	0	1	0	2	0	15
cap111	cpu	>1,200	>1,200	633	162	3.03	**2.74**
	#nodes	–	–	72,924	72,924	439	199
	avg-sub	0	2	0	3	0	12
capmo1	cpu	>1,200	>1,200	>1,200	>1,200	>1,200	**984**
	#nodes	–	–	–	–	–	–
	avg-sub	0	15	0	23	0	64
driverlog02ac	cpu	253	49.5	10.6	**7.99**	19.3	13.5
	#nodes	4,729K	701K	19,454	8,412	19,444	8,402
	avg-sub	0	0	0	0	0	0
driverlogs06	cpu	>1,200	>1,200	187	**61.0**	218	80.1
	#nodes	–	–	2,049K	609K	2,049K	609K
	avg-sub	0	0	0	0	0	0
mprime04ac	cpu	>1,200	30.9	>1,200	**15.7**	>1,200	43.7
	#nodes	–	189K	–	22,373	–	20,381
	avg-sub	0	0	0	1	0	1
myciel5g-3	cpu	38.1	19.6	**3.87**	4.18	4.36	4.57
	#nodes	518K	168K	10,046	9,922	6,159	6,128
	avg-sub	0	2	0	3	0	2
celar7-sub1	cpu	925	820	147	145	135	**86.4**
	#nodes	9,078K	1,443K	732K	91,552	796K	70,896
	avg-sub	0	6	0	9	0	6
graph07	cpu	>1,200	261	**3.11**	3.58	3.86	4.3
	#nodes	6,156K	145K	1,112	647	1,796	1,514
	avg-sub	0	23	0	9	0	4
scen06-24	cpu	>1,200	>1,200	**1,061**	>1,200	>1,200	>1,200
	#nodes	–	–	–	375K	–	–
	avg-sub	0	2	0	6	0	7
spot5-29	cpu	>1,200	>1,200	30.1	**18.0**	47.2	22.5
	#nodes	–	–	343K	174K	352K	185K
	avg-sub	0	0	0	0	0	0

TABLE 3.2 – Résultats illustratifs obtenus sur certaines instances.

divisé par 10. Cela signifie que sur de telles instances, PSNS est assez coûteux, ce qui laisse sans doute la place à des optimisations supplémentaires.

3.8 Conclusion

Dans ce chapitre, nous avons étudié la propriété de substituabilité souple au voisinage pour les réseaux de contraintes pondérées (WCNs) et analysé les conditions permettant l'identification de valeurs substituables par un algorithme de complexité raisonnable (i.e., polynomial). Nous avons prouvé que, même dans les cas simples, lorsque $k \neq +\infty$, le problème de décider si une valeur est souple-substituable à une autre au voisinage est coNP-difficile. Nous avons également étudié les relations entre substituabilité souple au voisinage et cohérence d'arc souple. Enfin, nous avons proposé un algorithme qui exploite des arrêts précoces de boucles, des résidus et un mécanisme d'horodatage, et montré expérimentalement qu'il peut être efficace au cours de la recherche.

Chapitre 4

Extension des cohérences WCSP aux tuples

Sommaire

Nous avons présenté dans l'état de l'art des algorithmes utilisés dans le cadre WCSP pour filtrer l'espace de recherche comme par exemple : FDAC, EDAC, VAC et OSAC (voir chapitre 2). Nous avons mentionné aussi que ces algorithmes utilisent des opérations de transfert de coûts, appelées transformations préservant l'équivalence (EPTs). Nous rappelons que ces EPTs dites classiques permettent la projection (ou extension) de coûts entre contraintes binaires et contraintes unaires ainsi que la projection de coûts entre contraintes unaires et la contrainte particulière, $w_\emptyset$, représentant un coût à additionner à toute instanciation complète. Nous proposons dans cet article d'étudier une nouvelle opération (EPT) pour le cadre WCSP, appelée `TupleProject` qui généralise les EPTs classiques. Nous l'utilisons pour définir de nouvelles propriétés basées sur le transfert de coûts entre contraintes, y compris lorsque ces contraintes sont d'arité supérieure ou égale à 2. Une nouvelle propriété, la cohérence de tuples (TC), est identifiée, ainsi qu'une version "optimale", OTC, qui peut être perçue comme une généralisation de OSAC (Optimal Soft Arc Consistency).

Pour illustrer l'intérêt du transfert de coût entre tuples de différentes contraintes (non unaires), considérons un WCN contenant 4 variables x, y, z et t avec pour chaque variable deux valeurs possibles a et b, et 3 contraintes w_{xy}, w_{xyz} et w_{xyt} définies dans la figure 4.1 (pour chaque tuple possible, le coût est donné dans la colonne de droite). Ce WCN est arc-cohérent (AC) et même arc-cohérent souple optimal (OSAC) car il n'existe pas de transformation (classique) permettant d'augmenter la borne $w_\emptyset$ qui fournit un minorant du coût du réseau. En revanche, nous remarquons pour la contrainte w_{xyz} que l'instanciation $\{(x,a),(y,b)\}$ a nécessairement un coût de 1. Par conséquent, ce coût peut être déplacé (sans perte d'équivalence) vers le tuple $\{(x,a),(y,b)\}$ de la contrainte w_{xy} à partir de la contrainte w_{xyz}. De même, pour

la contrainte w_{xyt}, nous remarquons que l'instanciation $\{(x,b),(y,a)\}$ a également nécessairement un coût de 1. Par conséquent, ce coût peut être déplacé vers le tuple $\{(x,b),(y,a)\}$ de la contrainte w_{xy} à partir de la contrainte w_{xyz}. On obtient alors le réseau de la figure 4.2. Nous sommes capables maintenant de transférer un coût de 1 sur les valeurs de x ou y depuis la contrainte w_{xy}, ensuite, de le transférer vers $w_\emptyset$. Le WCN obtenu (voir figure 4.3) est équivalent au problème d'origine et a maintenant un minorant $w_\emptyset = 1$ et toutes les contraintes sont uniformément nulles.

x	y	w_{xy}
a	a	1
a	b	0
b	a	0
b	b	1

$w_\emptyset$
0

x	y	z	w_{xyz}
a	a	a	0
a	a	b	0
a	b	a	1
a	b	b	1
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

x	y	t	w_{xyt}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	1
b	a	b	1
b	b	a	0
b	b	b	0

FIGURE 4.1 – Réseau initial

x	y	w_{xy}
a	a	1
a	b	1
b	a	1
b	b	1

$w_\emptyset$
0

x	y	z	w_{xyz}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

x	y	t	w_{xyt}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

FIGURE 4.2 – Réseau obtenu après les premiers transferts

4.1 Projection entre tuples

Nous proposons une nouvelle opération (EPT) qui généralise les EPTs classiques : la projection entre tuples notée `TupleProject`. Notons que dans [Cooper, 2005], l'auteur a proposé une opération similaire qui transfère les coûts entre des contraintes d'arité quelconques pour le cadre VCSP. Il est à noter également que dans [Favier et al., 2011], les auteurs utilisent des transferts de coût entre les contraintes n-aires et les contraintes binaires. L'idée est de pouvoir transférer un coût α entre des tuples de différentes contraintes et d'arité quelconque. La

x	y	w_{xy}
a	a	0
a	b	0
b	a	0
b	b	0

$w_\emptyset$
1

x	y	z	w_{xyz}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

x	y	t	w_{xyt}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

FIGURE 4.3 – Réseau final

définition de la projection entre tuples est la suivante :

Définition 70. *Soient w_S et $w_{S'}$ deux contraintes telles que $S \subset S'$.*
L'opération `TupleProject`*$(w_S, t, w_{S'}, \alpha)$ consiste à projeter un coût α sur un tuple t de w_S depuis les tuples t' de $w_{S'}$ tels que $t \subset t'$, avec $0 < \alpha \leq \min_{t' \in l(S'), t \subset t'} w_{S'}(t')$. Il s'agit :*

- *d'ajouter α à $w_S(t)$, à l'aide de l'opérateur $\oplus$;*
- *de soustraire α de $w_{S'}(t')$ à l'aide de l'opérateur $\ominus$, et ceci pour chaque $t' \in l(S')$ tel que $t \subset t'$.*

Dans l'exemple de la figure 4.1, la projection sur le tuple $t = \{(x, a), (y, b)\}$ d'un coût 1 peut se faire grâce à la fonction `TupleProject`$(w_{xy}, t, w_{xyz}, 1)$.

Clairement, `TupleProject` préserve l'équivalence et généralise `Project` et `UnaryProject`. En effet, l'opération `Project` correspond au cas $|S| = 1$ et $|S'| > 1$ tandis que l'opération `UnaryProject` correspond au cas $|S| = 0$ et $|S'| = 1$.

L'EPT `TupleProject` est également décrite par l'algorithme 20 qui factorise un coût α depuis les tuples de $w_{S'}$ vers le tuple t de w_S. De même, l'EPT classique `Extend` est généralisée comme indiqué dans l'algorithme 21.

Algorithme 20 : `TupleProject`

Entrées : w_S : Contrainte, t : Tuple, $w_{S'}$: Contrainte, α : Coût

1 **pré-condition :** $S \subset S' \wedge t \in l(S)$;
2 **pré-condition :** $0 < \alpha \leq \min_{t' \in l(S'), t \subset t'} w_{S'}(t')$;
3 $w_S(t) \leftarrow w_S(t) \oplus \alpha$;
4 **pour chaque** $t' \in l(S') \mid t \subset t'$ **faire**
5 $\quad w_{S'}(t') \leftarrow w_{S'}(t') \ominus \alpha$

Algorithme 21 : `TupleExtend`

Entrées : w_S : Contrainte, t : Tuple, $w_{S'}$: Contrainte, α : Coût

1 **pré-condition :** $S \subset S' \wedge t \in l(S)$;
2 **pré-condition :** $0 < \alpha \leq w_S(t)$;
3 $w_S(t) \leftarrow w_S(t) \ominus \alpha$;
4 **pour chaque** $t' \in l(S') \mid t \subset t'$ **faire**
5 $\quad w_{S'}(t') \leftarrow w_{S'}(t') \oplus \alpha$

4.2 La cohérence de tuples TC

Dans ce qui suit, nous introduisons un nouveau type de cohérence qui ressemble à la k-cohérence proposé dans [Cooper, 2005]. Notons que la nouveauté dans notre travail est que nous exploitons les intersections de contraintes afin d'améliorer la borne inférieure $w_\emptyset$. Avant de définir ce type de cohérence, nous introduisons la notion de *sur-contrainte* d'un ensemble de variables. Étant donné un réseau de contraintes pondérées (WCN) $P = (\mathscr{X}, \mathscr{W}, k)$:

Définition 71. *Soient un WCN et $S \subseteq \mathscr{X}$ un ensemble de variables. Une sur-contrainte de S est une contrainte $w_{S'}$ telle que $S \subset S'$. L'ensemble des sur-contraintes de S est noté $\Gamma(S) = \{w_{S'} \in \mathscr{W} \mid S \subset S'\}$.*

Définition 72. *Soient $S \subseteq \mathscr{X}$ un ensemble de variables, t un tuple de $l(S)$, et $w_{S'}$ une contrainte de $\Gamma(S)$. L'ensemble $\sigma_{w_{S'}}(t) = \{t' \in l(S') \mid t \subset t'\}$ contient tous les tuples définis sur S' étendant t.*

Définition 73. *Soient $S \subseteq \mathscr{X}$ un ensemble de variables, t un tuple de $l(S)$, et $w_{S'}$ une contrainte de $\Gamma(S)$. On dit que $t' \in \sigma_{w_{S'}}(t)$ est un support de t sur $w_{S'}$ ssi $w_{S'}(t') = 0$.*

Quand un tuple n'a pas de support, une factorisation est possible et permet d'en créer un. Soit une contrainte w_S de $\mathscr{W}$ et un tuple t de w_S, un support de t sur une contrainte $w_{S'} \in \Gamma(S)$ est créé en procédant comme suit :

- chercher le tuple $t'' = argmin_{t' \in \sigma_{w_{S'}}(t)} w_{S'}(t')$ qui va devenir le support de t sur $w_{S'}$;
- et exécuter `TupleProject`$(w_S, t, w_{S'}, w_{S'}(t''))$.

Pour illustrer ce dernier point, nous reprenons notre exemple introductif (voir figure 4.1). Sur ce WCN, il est clair que le tuple $t = \{(x, a), (y, b)\}$ n'a pas de support sur la contrainte w_{xyz}. Pour créer un support à ce tuple, il suffit d'appliquer l'EPT `TupleProject`$(w_{xy}, t, w_{xyz}, 1)$.

Définition 74. *Soit un ensemble de variables $S \subseteq \mathscr{X}$. Un tuple $t \in l(S)$ est tuple-cohérent (TC) ssi il possède un support sur chaque contrainte $w_{S'} \in \Gamma(S)$. On dit que S est TC ssi tous les tuples de $l(S)$ sont TC. On dit que P est TC ssi tout ensemble de variables $S \subseteq \mathscr{X}$ est TC.*

Chercher à rendre TC un WCN n'est pas toujours réaliste car, dans certains cas, cela peut nécessiter l'introduction d'une contrainte pour tout sous-ensemble possible de variables du problème, ce qui est exponentiel. C'est la raison pour laquelle nous proposons d'étudier une forme limitée de cette cohérence, notée TC_r, en ne cherchant des supports que pour les tuples d'arité inférieure ou égale à une arité limite r donnée. En pratique, r aura une petite valeur, ce qui nous permettra de n'introduire, si nécessaire, que des contraintes d'arité faible.

Définition 75. *Pour tout entier strictement positif r, P est TC_r si $\forall S \subseteq \mathcal{X}$ tel que $|S| \leq r$, S est TC.*

Avec la propriété TC_r, chaque fois qu'un tuple t défini sur un ensemble de variables S d'arité $i \leq r$ n'admet pas de support sur une contrainte, une factorisation doit être effectuée vers t. S'il n'existe pas de contrainte w_S dans le WCN P, une nouvelle contrainte doit être ajoutée à P, en initialisant les coûts de tous ses tuples à 0 (avant la projection vers t) : $\forall t \in l(S), w_S(t) = 0$. Il est à noter que pour $r = 1$, TC_r est équivalent à AC. En effet, dans ce cas, l'établissement de TC_r consiste à créer pour chaque valeur de chaque variable un support sur les contraintes qui portent sur cette variable. Si la structure de valuation $\mathcal{S}(1)$ est utilisée, dans ce cas, TC_r est équivalent à la *pairwise consistency* [Janssen et al., 1989].

Algorithme 22 : $TC1_r$

Entrées : P : WCN

```
1  pour i ← r à 0 faire
2      pour chaque S ⊆ 𝒳 tel que |S| = i faire
3          pour chaque t ∈ l(S) faire
4              pour chaque w_S' ∈ Γ(S) faire
5                  α ← min_{t'∈l(S'),t⊂t'} w_S(t');
6                  si α > 0 alors
7                      si w_S ∉ 𝒲 alors
8                          ajouter la contrainte w_S de coût uniformément nul à P ;
9                      fin
10                     TupleProject(w_S, t, w_S', α);
11                 fin
12             fin
13         fin
14     fin
15 fin
```

Établir TC_r peut être réalisé à l'aide d'un premier algorithme appelé $TC1_r$ (voir algorithme 22). Nous commençons d'abord par établir la propriété TC pour les tuples d'arité r, puis les tuples d'arité $r-1$, etc. De fait, nous établissons la propriété TC par ordre décroissant d'arité, car les transferts de coûts s'effectuent de contraintes d'arité j vers des contraints d'arité i avec $j > i$. L'intérêt est qu'un support établi pour un tuple t d'une contrainte d'arité i reste valide lorsque la recherche de supports s'effectue aux niveaux inférieurs à i.

Proposition 11. *Appliqué à tout WCN P, l'algorithme $TC1_r$ produit un WCN qui est TC_r et qui est équivalent à P.*

Preuve. L'algorithme n'applique que des EPTs, donc le réseau obtenu est équivalent au réseau initial. Les lignes 2 à 5 construisent un support (le tuple qui a le coût α dans $w_{S'}$) pour l'ensemble de variables S tel que $|S| = i$. On peut noter que seuls les coûts des tuples de $l(S)$ peuvent augmenter. Comme la boucle externe est décroissante, une fois un support établi, son

coût ne peut plus jamais augmenter et reste donc égal à 0. Autrement dit, une fois établi, un support n'est jamais remis en cause par l'algorithme. La boucle énumérant toutes les arités de r à 0, un support est établi pour tout ensemble S t.q. $|S| \leq r$. □

Algorithme 23 : TC2$_r$

Entrées : P : WCN

```
1  compléter P avec toutes les contraintes manquantes d'arité inférieure ou égale à r ;
2  pour chaque w_S ∈ 𝒲 tel que |S| = r faire
3      pour chaque t ∈ l(S) faire
4          pour chaque w_S' ∈ Γ(S) faire
5              α ← min_{t'∈l(S'),t⊂t'} w_S(t');
6              si α > 0 alors
7                  TupleProject(w_S, t, w_S', α);
8              fin
9          fin
10     fin
11 fin
12 pour i ← r − 1 à 0 faire
13     pour chaque w_S ∈ 𝒲 tel que |S| = i faire
14         pour chaque t ∈ l(S) faire
15             pour chaque w_S' ∈ Γ(S) t.q. |S'| = i + 1 faire
16                 α ← min_{t'∈l(S'),t⊂t'} w_S(t');
17                 si α > 0 alors
18                     TupleProject(w_S, t, w_S', α);
19                 fin
20             fin
21         fin
22     fin
23 fin
```

L'algorithme 22 nécessite de considérer pour tout ensemble de variables S toutes les contraintes de $\Gamma(S)$. Une autre stratégie pour établir TC$_r$ consiste à ajouter a priori toutes les contraintes d'arité inférieure ou égale à r qui n'existeraient pas déjà dans le réseau. Cet ajout systématique permet de simplifier les factorisations en exploitant des propriétés de transitivité et reste envisageable tant que r est faible. Nous obtenons un second algorithme $TC2_r$ (voir algorithme 23).

La figure 4.4 illustre le principe de ce second algorithme. Soit S_1, S_2 et S_3 trois ensembles de variables tels que $S_1 \subset S_2 \subset S_3$ et $r \geq |S_3| = |S_2| + 1 = |S_1| + 2$. Lorsqu'une factorisation est possible de S_3 vers S_1, elle peut s'effectuer en deux étapes : de S_3 vers S_2 puis de S_2 vers S_1.

Proposition 12. *Appliqué à tout WCN P, l'algorithme TC2$_r$ produit un WCN qui est TC$_r$ et*

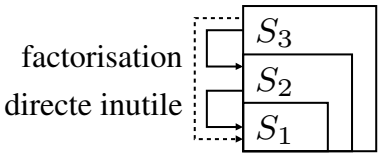


FIGURE 4.4 – Exploitation de la transitivité des factorisations

qui est équivalent à P.

Preuve. La première instruction de l'algorithme assure que toutes les contraintes d'arité inférieure ou égale à r sont présentes dans le réseau. Aussi, quand on peut appliquer `TupleProject` de $w_{S'}$ d'arité a' vers w_S d'arité a avec $a < a' \leq r$, on a alors la garantie qu'il existe une succession de contraintes w_{S_i} pour $i \in a..a'$ telles que $S_{a'} = S'$, $S_a = S$, $\forall i \in a..a' - 1, S_i \subset S_{i+1}$ et $|S_i| = i$. De ce fait, on peut remplacer une factorisation directe de $w_{S'}$ vers w_S par une succession de factorisations de $w_{S_{i+1}}$ vers w_{S_i}. □

Avec l'algorithme TC2$_r$, il faut donc d'abord ajouter les contraintes w_S qui portent sur les différents sous-ensembles de variables $S \subseteq vars(P)$ tel que $|S| \leq r$ et $w_S \notin \mathscr{W}$. Pour illustrer ce point, prenons un WCN P comportant seulement deux contraintes ternaires w_{xyz} et w_{wyz}. L'ensemble des contraintes du WCN P' équivalent à P sur lequel nous appliquerons la propriété TC$_r$ avec $r = 2$ est donné par la figure 4.5.

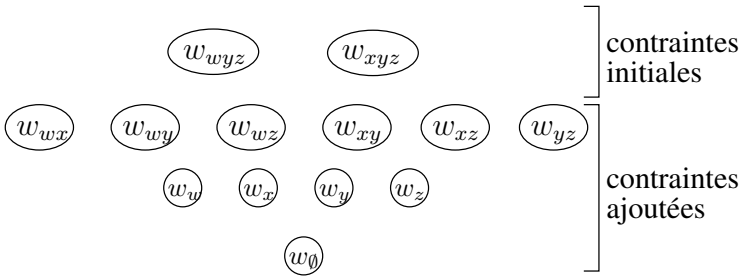


FIGURE 4.5 – Ajout des contraintes d'arité inférieure ou égale à $r = 2$

En fait, il est possible d'éviter l'introduction de certaines contraintes inutiles. Sur l'exemple précédent, il est inutile d'introduire w_{wx} car elle n'a pas de sur-contrainte. La définition 76 formalise cette remarque.

Définition 76. *Un ensemble de variables $S \subseteq \mathscr{X}$ est dit* isolé *ssi* $\Gamma(S) = \emptyset$.

Clairement, si un ensemble de variables S est isolé alors il est TC. Ainsi, lorsqu'un ensemble de variables S est isolé et que la contrainte w_S est manquante, il n'est pas nécessaire d'ajouter cette contrainte au réseau. Notre réseau de contraintes de la figure 4.5 peut être simplifié en supprimant la contrainte w_{wx}.

Une variante de TC$_r$ consiste à ne pas ajouter de contraintes au réseau, et à simplement créer des supports pour les tuples des contraintes existantes d'arité inférieure ou égale à r.

Comme il s'agit d'une version affaiblie de TC_r, nous appellerons cette variante TC_r^w (weak TC_r). Établir TC_r^w peut s'effectuer sur la base de l'algorithme $TC1_r$ (voir algorithme 22) en s'interdisant simplement l'ajout de contrainte et en ne parcourant que les ensembles S qui sont le scope de contraintes existantes.

4.3 La cohérence de tuples optimale (OTC)

Nous proposons maintenant une méthode cherchant à tirer le meilleur parti de l'opération `TupleProject`, tout comme OSAC exploite au mieux les EPTs classiques.

Il est facile de montrer que l'application itérée de `TupleProject` n'est pas confluente, ce qui signifie qu'on peut obtenir plusieurs résultats différents selon l'ordre dans lequel on effectue les opérations. D'une part, le WCN obtenu après factorisation n'est pas unique car les coûts peuvent être factorisés de différentes manières, mais également, la borne inférieure $w_\emptyset$ peut également être différente.

Dans cette section, nous développons la même stratégie que pour OSAC [Cooper et al., 2007], à savoir rechercher un ensemble de projections `TupleProject` qui peuvent être appliquées simultanément de manière à augmenter le plus possible le minorant $w_\emptyset$. Cette recherche est codée sous la forme d'un problème linéaire. De manière à permettre la résolution de ce problème en temps polynomial, nous autorisons la factorisation de coûts rationnels et pas seulement entier. Nous considérons donc dans ce qui suit que les coûts du WCN sont rationnels.

Comme pour OSAC, on ne cherche pas à déterminer dans quel ordre appliquer les opérations `TupleProject`, ni à vérifier si à chaque étape le WCN obtenu est valide (tous les coûts restent positifs). On considère que les opérations sont effectuées simultanément et notre seule exigence est que le WCN final soit valide et équivalent au réseau initial.

Définition 77. *Étant donné un WCN P, un ensemble d'opérations* `TupleProject` *est valide si et seulement si, appliquées simultanément, ces opérations transforment P en un WCN P' qui est valide (tous les coûts restent positifs) et équivalent (toute instanciation complète a le même coût dans P et P').*

La cohérence de tuples optimale (OTC) généralise OSAC car elle utilise l'opération `TupleProject` qui subsume les opérations *Project* et *UnaryProject* utilisées dans OSAC.

Définition 78. *Un WCN P est tuple-cohérent optimal (OTC) s'il n'existe pas d'ensemble d'opérations* `TupleProject` *valide qui, appliqué à P, permette d'augmenter le minorant $w_\emptyset$.*

La méthode OTC peut être déclinée de deux manières : soit en n'ajoutant pas de contrainte comme dans OSAC, soit en complétant initialement le réseau avec des contraintes supplémentaires. Dans les deux cas, il est nécessaire en pratique de limiter l'arité des tuples sur lesquels porte *TupleProject* afin de conserver une complexité raisonnable. Nous nommerons OTC_r^w (OTC_r weak) la version qui n'ajoute pas de contrainte, et OTC_r la version qui ajoute des contraintes. Dans les deux cas, r est l'arité maximale considérée par la méthode. Nous présentons ces deux méthodes dans les sections qui suivent.

$$\begin{cases} \max : \sum_{x \in vars(P)} \alpha^x_{w_\emptyset} \\ \forall w_S \in \mathscr{W}, \text{ t.q. } |S| > r, \forall t \in l(S) : w_S(t) - \sum_{\substack{S' \subset S \\ |S'| \leq r}} \alpha^S_{t[S']} \geq 0 \\ \forall w_S \in \mathscr{W}, \text{ t.q. } |S| \leq r, \forall t \in l(S) : w_S(t) - \sum_{S' \subset S} \alpha^S_{t[S']} + \sum_{w_{S'} \in \Gamma(S)} \alpha^{S'}_t \geq 0 \end{cases}$$

FIGURE 4.6 – Système linéaire généré par OTC^w_r

4.3.1 Cohérence de tuples optimale restreinte faible OTC^w_r

Dans cette version, on s'interdit de modifier l'ensemble de contraintes (nous supposons l'existence d'une contrainte unaire pour chaque variable ainsi que $w_\emptyset$). Par ailleurs, pour des raisons d'efficacité, on restreindra les opérations $TupleProject$ au cas où au moins l'une des contraintes est d'arité inférieure ou égale à r. Cette version est nommée OTC^w_r.

Pour trouver l'ensemble d'opérations `TupleProject` valide qui maximise $w_\emptyset$, on peut définir un problème de programmation linéaire (PL) comme suit. Nous notons par $\alpha^{S'}_t$ le coût transféré depuis une contrainte $w_{S'}$ vers le tuple t de la contrainte w_S avec $t \in l(S)$ par l'opération $\texttt{TupleProject}(w_S, t, w_{S'}, \alpha^{S'}_t)$. En utilisant la notation $t[S]$ pour représenter la projection du tuple t sur le scope S, le problème linéaire obtenu est donné en figure 4.6.

La première inéquation traite des contraintes d'arité strictement supérieure à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité inférieure ou égale à r doit être au final positif ou nul.

La seconde inéquation traite des contraintes d'arité inférieure ou égale à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité inférieure plus les transferts depuis les contraintes d'arité supérieure doit être au final positif ou nul.

La fonction objectif maximise le transfert de coûts vers $w_\emptyset$ qui est la somme des transferts $\alpha^x_{w_\emptyset}$ de la contrainte unaire portant sur x vers $w_\emptyset$.

On notera que dans le système linéaire, les variables α^S_t peuvent être soit positives, soit négatives. Une valeur positive correspond à une projection d'un coût depuis plusieurs tuples vers un seul, tandis qu'une valeur négative correspond à une extension, c'est à dire le transfert d'un coût depuis un tuple unique vers plusieurs tuples. De ce fait, OTC ne se contente pas d'établir des supports mais alterne les opérations de projection et d'extension de manière à obtenir la meilleure borne $w_\emptyset$.

Lorsque les coûts sont rationnels, ce système linéaire se résout en temps polynomial [Karmarkar, 1984].

4.3.2 Cohérence de tuples optimale restreinte OTC_r

Une seconde déclinaison de OTC consiste à ajouter au réseau de départ toutes les contraintes d'arité inférieure ou égale à r et ensuite effectuer les opérations $TupleProject$ en se restreignant au cas ou au moins l'une des contraintes est d'arité inférieure ou égale à r. Cette méthode est nommée OTC_r.

$$\begin{cases} \max : \sum\limits_{x \in vars(P)} \alpha^{x}_{w_\emptyset} \\ \forall w_S \in \mathscr{W} \text{ t.q. } |S| > r, \forall t \in l(S) : w_S(t) - \sum\limits_{\substack{S' \subset S, \\ |S'|=r}} \alpha^{S}_{t[S']} \geq 0 \\ \forall w_S \in \mathscr{W} \text{ t.q. } |S| = r, \forall t \in l(S) : w_S(t) - \sum\limits_{\substack{S' \subset S, \\ |S|=|S'|+1}} \alpha^{S}_{t[S']} + \sum\limits_{w_{S'} \in \Gamma(S)} \alpha^{S'}_{t} \geq 0 \\ \forall w_S \in \mathscr{W} \text{ t.q. } |S| < r, \forall t \in l(S) : w_S(t) - \sum\limits_{\substack{S' \subset S, \\ |S|=|S'|+1}} \alpha^{S}_{t[S']} + \sum\limits_{\substack{S \subset S', \\ |S'|=|S|+1}} \alpha^{S'}_{t} \geq 0 \end{cases}$$

FIGURE 4.7 – Système linéaire généré par OTC_r

Dans ce cas, en utilisant la transitivité des factorisations, le système linéaire peut être simplifié. Ce système est présenté en figure 4.7.

La première inéquation traite des contraintes d'arité strictement supérieure à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité égale à r doit être au final positif ou nul.

La deuxième inéquation traite des contraintes d'arité égale à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité immédiatement inférieure plus les transferts depuis les contraintes d'arité plus grande que r doit être au final positif ou nul.

La troisième inéquation traite des contraintes d'arité inférieure à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité immédiatement inférieure plus les transferts depuis les contraintes d'arité immédiatement supérieure doit être au final positif ou nul.

La fonction objectif maximise le transfert de coûts vers $w_\emptyset$ qui est la somme des transferts $\alpha^{x}_{w_\emptyset}$ de la contrainte unaire portant sur x vers $w_\emptyset$.

Il est clair que OTC_r (avec $r \geq 1$) est également une généralisation de OSAC, et donc OTC_r obtient un minorant $w_\emptyset$ au moins aussi grand que OSAC.

L'intérêt de OTC_r est illustré par l'exemple de la figure 4.8. Ce réseau est OTC_2 et donc OSAC. Par contre, il n'est pas OTC_3 car l'introduction de la contrainte ternaire w_{xyz} va permettre d'augmenter le minorant $w_\emptyset$. En effet, une fois la contrainte w_{xyz} introduite (avec des coûts nuls au départ), on peut transférer le coût de chaque tuple de chaque contrainte binaire vers les tuples correspondant de la contrainte ternaire. Par exemple, le coût 1 assigné au tuple $\{(x,a),(y,a)\}$ est étendu aux deux tuples de $\{(x,a),(y,a),(z,a)\}$ et $\{(x,a),(y,a),(z,b)\}$ de w_{xyz}. À ce moment, les contraintes binaires ont toutes des coûts nuls et les coûts de la contrainte ternaire sont donnés dans la figure 4.9.

À partir de ce moment, il est direct de factoriser un coût de 1 depuis w_{xyz} vers chacun des tuples de w_{xy} (par exemple), puis de factoriser ce même coût vers w_x (par exemple) pour enfin factoriser 1 vers $w_\emptyset$.

Cet exemple montre que dans certains cas, l'ajout de sur-contraintes dans un WCN permet d'augmenter le minorant $w_\emptyset$.

Il est à noter que le réseau obtenu par OTC_r n'est pas nécessairement TC_r, tout comme le réseau obtenu par OSAC n'est pas nécessairement AC.

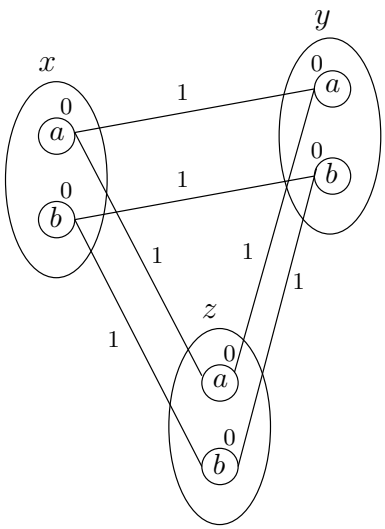


FIGURE 4.8 – Un WCN OTC_2 cohérent mais non OTC_3 cohérent

x	y	z	w_{xyz}
a	a	a	3
a	a	b	1
a	b	a	1
a	b	b	1
b	a	a	1
b	a	b	1
b	b	a	1
b	b	b	3

FIGURE 4.9 – La contrainte w_{xyz} après les opérations d'extension

4.4 OSAC vs TC et OTC

Dans cette section, nous revenons sur OSAC, TC et OTC, cherchant à établir des rapports entre ces cohérences.

Proposition 13. *OSAC n'est pas $w_\emptyset$-supérieur à TC.*

Preuve. L'exemple du WCN de la figure 4.1 est OSAC-cohérent. En revanche, il n'est pas TC-cohérent. L'établissement de la propriété TC sur ce WCN permet d'augmenter la borne inférieure $w_\emptyset$ d'une unité de coût (voir la figure 4.3). D'où, OSAC n'est pas $w_\emptyset$-supérieur à TC. □

Proposition 14. *TC n'est pas $w_\emptyset$-supérieur à OSAC.*

Preuve. Considérons l'exemple du WCN P de la figure 4.10. Il est assez facile de vérifier que P est TC-cohérent. En revanche, il n'est pas OSAC-cohérent car il existe une transformation

SAC qui peut augmenter la borne inférieure $w_\emptyset$ d'une unité de coût. □

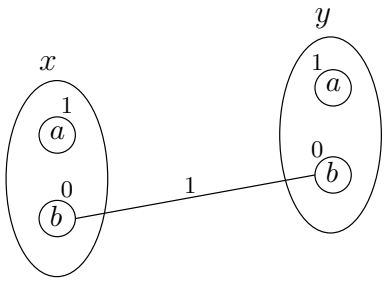


FIGURE 4.10 – Les relations entre les différents types de cohérence

De ces deux dernières propositions, on en conclut que OSAC est $w_\emptyset$-incomparable à TC.

Proposition 15. *OSAC est équivalent à OTC^w_1.*

Preuve. Le système linéaire généré par OSAC est identique au système linéaire généré par OTC^w_1. □

Proposition 16. *Sur l'ensemble des réseaux de contraintes pondérées binaires, OSAC est équivalent à OTC^w_r, $\forall r \geq 1$.*

Preuve. Dans le cas des réseaux binaires, que ce soit avec OSAC ou OTC^w_r, les seuls transferts possibles sont effectués depuis les contraintes binaires vers les contraintes unaires, puis, depuis les contraintes unaires vers $w_\emptyset$. Aussi, le système linéaire défini par OSAC est le même que celui défini par OTC^w_r, sous condition que $r \geq 1$. □

Appliquer OSAC (respectivement, OTC^w_r) sur un WCN P permet de calculer un minorant que nous noterons $w_\emptyset^{OSAC}(P)$ (respectivement, $w_\emptyset^{OTC^w_r}(P)$).

Proposition 17. *$\forall r \geq 2$, OTC^w_r est strictement plus fort que OSAC, signifiant que :*
- *pour tout WCN P, $w_\emptyset^{OTC^w_r}(P) \geq w_\emptyset^{OSAC}(P)$*
- *il existe des réseaux de contrainte pondérées P t.q. $w_\emptyset^{OTC^w_r}(P) > w_\emptyset^{OSAC}(P)$*

Preuve. Pour OTC^w_r, le système généré contient les inéquations correspondant aux transferts générés par OSAC plus des inéquations pour des transferts potentiels entres tuples. Cela garantit que la valeur du minorant obtenu après l'application de OTC^w_r est supérieure ou égale à celle obtenue après l'application de OSAC. L'exemple introductif à ce papier démontre le caractère strict de cette relation entre OTC^w_r et OSAC. □

De ces résultats nous pouvons conclure que tout réseau qui est OTC-cohérent est forcément OSAC-cohérent car le système linéaire généré par OTC contient le système linéaire généré par OSAC. D'où, la propriété OTC implique la propriété OSAC. Selon la proposition 1, la propriété OTC est $w_\emptyset$-supérieure à la propriété OSAC. Dans la figure 4.11, nous mettons à jour les deux relations d'implication et de $w_\emptyset$-supériorité que nous avons présentés précédemment, en ajoutant les deux propriétés TC et OTC.

Pour finir, nous nous intéressons au cas des WCNs binaires acycliques (i.e., des WCNs dont le graphe de contraintes est acyclique).

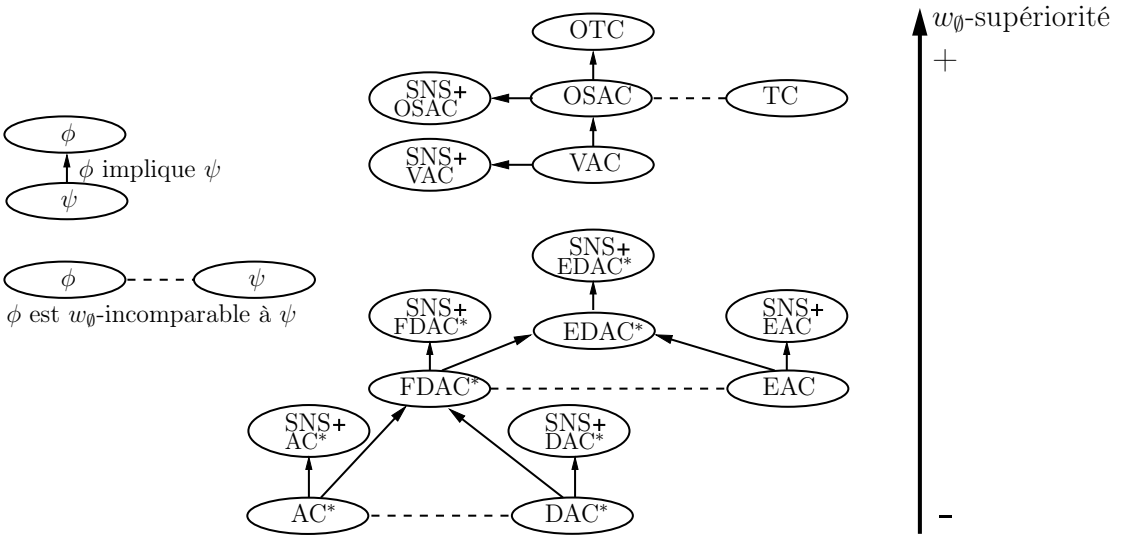


FIGURE 4.11 – Les relations entre les différents types de cohérence

Proposition 18. *Soit P un WCN acyclique. Si P est AC***−cohérent alors $w_\emptyset$ ne représente pas forcément le coût de la solution optimale.*

Preuve. Considérons le WCN acyclique P défini comme suit :
$\mathscr{X} = \{x, y\}$, $dom(x) = dom(y) = \{a, b\}$,k=$\infty$ et $\mathscr{W} = \{w_{xy}, w_x, w_y, w_\emptyset\}$.
Les contraintes sont définies par les tables suivantes :

x	y	w_{xy}
a	a	0
a	b	0
b	a	0
b	b	1

x	w_x
a	1
b	0

y	w_y
a	1
b	0

$w_\emptyset$
0

Il est clair que, P AC^*−cohérent. En revanche, il est facile de voir que le coût de la solution optimale est de 1. D'où, la valeur de $w_\emptyset$ ne représente pas le coût de la solution optimale dans cet exemple. □

Définition 79. *Soient P un WCN acyclique, G le graphe de contraintes associé à P et > un ordre sur l'ensemble de variable $\mathscr{X}$. On dit que G est un* ***arbre enraciné*** *ou* ***une arborescence*** *(en anglais rooted tree) si pour chaque $x \in \mathscr{X}$, il existe au plus une contrainte w_{xy} avec $y > x$.*

Proposition 19. *Soient P un WCN acyclique et > l'ordre sur l'ensemble de variable $\mathscr{X}$ qui permet d'obtenir le graphe enraciné G associé à P . Si P est DAC***-cohérent alors la valeur de $w_\emptyset$ représente le coût de la solution optimale.*

Preuve. Si le graphe G associé à P est un arbre enraciné selon l'ordre > et que P est DAC*-cohérent selon l'ordre > alors on peut construire une instanciation complète I de P qui a un

coût égal à 0 plus la borne inférieure $w_\emptyset$. En effet, il faut commencer par la variable r qui se trouve à la racine en lui affectant la valeur qui a un coût 0 (cela est possible car le WCN P est NC*-cohérent). Ensuite pour chaque variable x fille de la variable racine r, nous affectons à x la valeur qui représente le support complet de la variable r sur la contrainte w_{rx} (cela est possible car le WCN P est DAC-cohérent). Nous utilisons le même principe de manière récursive pour chaque variable y fille de chaque variable x. Ainsi, on peut construire l'instanciation complète I de coût $w_\emptyset$. □

La condition que G soit un arbre enraciné est une condition nécessaire. Dans le cas où G ne vérifie pas cette condition, la proposition 19 n'est pas vérifiée. Pour bien comprendre ce point, soit l'exemple du WCN de la figure 4.12. Ce WCN est DAC*-cohérent selon l'ordre $x > z > y$. En revanche, le graphe G n'est pas un arbre enraciné, ce qui justifie que le coût de la solution optimale (1 sur cet exemple) est plus grand que la valeur de $w_\emptyset$.

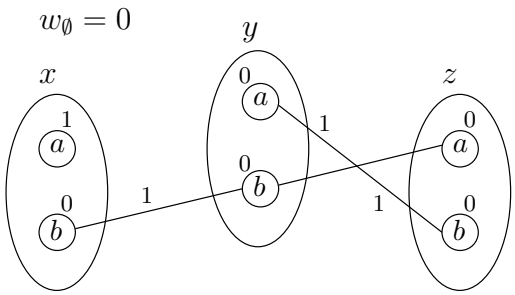


FIGURE 4.12 – Un WCN DAC*−cohérent avec l'ordre $x > z > y$ avec la valeur de $w_\emptyset$ différente du coût de la solution optimale

Corollaire 3. *Soient P un WCN acyclique, $>$ un ordre sur l'ensemble de variable $\mathscr{X}$ et G le graphe associé a P . Si P est FDAC*-cohérent ou P est EDAC*-cohérent selon l'ordre $>$ et G est un arbre enraciné alors la valeur de $w_\emptyset$ représente le coût de la solution optimale.*

Preuve. On sait que le graphe G représentant le WCN P est un arbre enraciné. Si nous assignons la racine avec la valeur qui constitue son support NC^* et ensuite chaque sommet avec la valeur qui représente le support complet de son parent, nous obtenons une instanciation complète I avec un coût 0. D'où, le coût de la solution optimale est le coût de I + la fonction d'arité nulle $w_\emptyset$. □

Proposition 20. *Soit P un WCN binaire acyclique. Si P est VAC (virtual arc-consistent) alors la valeur de $w_\emptyset$ représente le coût de la solution optimale.*

Preuve. Pour établir VAC, un réseau de contraintes classique (CN) $Bool(P)$ est construit à partir du WCN P en transformant chaque contrainte souple en une contrainte dure, où seuls les tuples de coût 0 sont autorisés. Comme par hypothèse, P est VAC, cela signifie que P n'est pas détecté incohérent par un algorithme établissant la cohérence d'arc (AC). D'autre part, le graphe de contraintes de P est le même que celui de $Bool(P)$, et celui-ci est acyclique. Tout CN acyclique qui est AC est connu pour admettre au moins une solution S. On en déduit que P admet S comme solution de coût 0, auquel il faut ajouter la valeur de $w_\emptyset$. □

Corollaire 4. *Soit P un WCN binaire acyclique. Si P est OSAC alors la valeur de $w_\emptyset$ représente le coût de la solution optimale (puisque un réseau OSAC est nécessairement VAC [Cooper et al., 2008]).*

De ce dernier corollaire, nous pouvons retenir que si le graphe de contraintes est acyclique alors OSAC permet de factoriser le coût optimal vers $w_\emptyset$. Par ailleurs, si on veut établir OTC sur un WCN binaire acyclique, il n'est pas nécessaire d'ajouter de sur-contraintes dans le but d'essayer d'obtenir un minorant plus grand que celui obtenu par OSAC. En revanche, si le WCN n'est pas acylique, l'ajout de sur-contraintes peut éventuellement produire un minorant plus important.

4.5 Résultats expérimentaux

Les expérimentations ont été menées en utilisant les séries d'instances WCSP disponibles aux adresses `http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/BenchmarkS` et `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`, et un cluster de Xeon 3.0GHz avec 2 Go de RAM sous Linux. Nous avons comparé notre type de cohérence OTC_r^w avec OSAC en prétraitement. Nous avons fixé le paramètre r de OTC à 2 pour limiter le nombre d'équation généré par ce type de cohérence. Rappelons que si le réseau de contraintes est binaire OTC_2^w est équivalent à OSAC. Afin de néanmoins améliorer $w_\emptyset$ sur les réseaux binaires, nous modifions le réseau de départ en ajoutant des contraintes ternaires de coût uniformément nul sur chaque triplet de variables reliées par une clique de contraintes binaires, afin de permettre des transferts de coût entre les tuples de contraintes binaires et les tuples de contraintes ternaires comme dans le cas de la série depot par exemple. Le temps limite alloué pour trouver une borne à une instance donnée est de 1200 secondes. Nous utilisons CPLEX version 12.4 pour résoudre les problèmes linéaires générés. Dans le tableau 4.1, nous résumons la comparaison entre OSAC et OTC. Nous donnons pour chaque série :

- le nombre d'instances où la borne inférieure donnée par OSAC est plus petite que celle donnée par OTC (OSAC< OTC).
- le nombre d'instances où la borne inférieure donnée par OTC est plus petite que celle donnée par OSAC, cela ne peut se produire que si le temps de résolution du PL généré par OTC dépasse le temps limite (OSAC> OTC).
- le nombre d'instances où la borne inférieure donnée par OSAC est égale à celle donnée par OTC (OSAC= OTC).
- le nombre d'instances qui dépasse le temps limite d'exécution pour la méthode OSAC (TO OSAC).
- le nombre d'instances qui dépasse le temps limite d'exécution pour la méthode OTC (TO OTC).

Comme le montre le tableau 4.1, les résultats de la comparaison entre OSAC et OTC sont décevants. Mis à part la série depot, l'établissement au prétraitement de la propriété OSAC renvoie une borne inférieure plus grande que celle fournit par la propriété OTC. Nous constatons une égalité entre OSAC et OTC sur les séries d'instances dimacs, jnh et pedigree. Nous remarquons aussi que le temps CPU de la résolution du PL généré par OTC dépasse les 1200 secondes fixées au départ sur plusieurs séries d'instances comme celar et graphs. Cela est jus-

tifié par la taille du PL générer par cette méthode. Nous rappelons que chaque ligne du PL représente un transfert de coût possible entre les arités de contraintes. D'où la nécessité de trouver un moyen pour réduire le nombre de transferts de coûts et par conséquent la taille du problème pour améliorer le temps CPU. On en conclut donc que l'établissement de la propriété OTC telle quelle en pratique n'est pas rentable.

Dans le tableau 4.2, nous détaillons pour chaque instance le temps de génération du problème linéaire (TG), le temps de résolution du problème (TR) et que la borne inférieure (LB) pour les deux méthodes OSAC et OTC.

Série	**OSAC<OTC**	**OSAC>OTC**	**OSAC=OTC**	**TO OSAC**	**TO OTC**
academics #inst=15	0	4	8	3	7
bwt #inst=10	0	7	2	1	8
celar #inst=15	0	12	0	3	15
coloring #inst=22	1	6	15	0	6
depot #inst=4	4	0	0	0	0
dimacs #inst=25	0	0	25	0	0
driver #inst=22	3	18	0	1	19
graphs #inst=10	0	4	0	6	10
jnh #inst=44	0	0	44	0	0
logistics #inst=4	0	4	0	0	4
mprime #inst=12	0	8	4	0	8
pedigree #inst=19	0	0	19	0	0
rover #inst=4	0	4	0	0	4
satellite #inst=7	0	7	0	0	7
scens #inst=21	0	15	0	6	21
spot5 #inst=21	5	15	1	0	15
warehouse #inst=55	0	1	37	18	18
zenotravel #inst=8	0	8	0	0	8

TABLE 4.1 – Comparaison entre OSAC et OTC_2^w sur les différente séries

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
Série academics						
16wqueens-wcsp	1.1	10.34	0.0	118.81	TO	-
4queens-wcsp	0.13	0.01	0.0	0.21	0.27	0.0
4wqueens-wcsp	0.05	0.01	0.0	0.17	0.17	0.0
8queens-wcsp	0.24	0.1	0.0	2.18	TO	-
8wqueens-wcsp	0.22	0.17	1.33	2.03	440.23	1.99
donald-wcsp	0.7	0.73	0.0	6.48	TO	-
langford-2-4-wcsp	0.22	0.1	0.0	2.09	387.72	0.0
langford-3-11-wcsp	6.85	TO	-	TO	TO	-
langford-3-9-wcsp	3.79	TO	-	TO	TO	-
send-wcsp	0.83	0.47	0.0	3.27	TO	-
slangford-3-11-wcsp	7.57	TO	-	TO	TO	-
warehouse0-wcsp	0.12	0.03	328.0	0.12	0.03	328.0
warehouse1-wcsp	0.79	2.37	730567.0	0.82	2.29	730567.0
zebra-wcsp	0.28	0.01	0.0	0.29	0.01	0.0
zebre-ext-wcsp	0.21	0.11	0.0	0.97	9.03	0.0
Série bwt						
bwt3cc-wcsp	0.34	0.3	177.0	2.27	105.76	177.02
bwt3c-wcsp	0.34	0.59	400.0	2.34	83.99	400.0
bwt4bc-wcsp	1.59	16.3	292.5	76.5	TO	-
bwt4cc-wcsp	1.63	15.95	348.40	71.51	TO	-
bwt4c-wcsp	1.61	13.58	491.66	81.48	TO	-
bwt5ac-wcsp	8.94	153.8	632.14	TO	TO	-
bwt5bc-wcsp	9.33	169.21	315.65	TO	TO	-
bwt5cc-wcsp	9.55	230.62	514.46	TO	TO	-
bwt5c-wcsp	9.17	150.85	632.14	TO	TO	-
bwt6-wcsp	36.73	TO	-	TO	TO	-
Série celar						
celar6-sub0-wcsp	2.04	15.24	0.0	TO	TO	-
celar6-sub1-24-wcsp	1.37	10.52	0.0	TO	TO	-
celar6-sub1-wcsp	2.67	42.62	0.0	TO	TO	-
celar6-sub2-wcsp	2.69	70.94	0.0	TO	TO	-
celar6-sub3-wcsp	3.22	110.29	0.0	TO	TO	-
celar6-sub4-20-wcsp	1.55	24.12	0.0	TO	TO	-
celar6-sub4reduc-wcsp	3.2	175.22	0.0	TO	TO	-
celar6-sub4-wcsp	3.77	169.35	0.0	TO	TO	-
celar7-sub0-wcsp	2.31	221.38	10006.49	TO	TO	-
celar7-sub1-20-wcsp	1.13	3.55	0.0	TO	TO	-
celar7-sub1-wcsp	2.79	20.64	0.0	TO	TO	-
celar7-sub2-wcsp	2.99	TO	-	TO	TO	-
celar7-sub3-wcsp	3.22	TO	-	TO	TO	-

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
celar7-sub4-22-wcsp	1.63	94.23	0.0	TO	TO	-
celar7-sub4-wcsp	3.6	TO	-	TO	TO	-
Série coloring						
dsjc125-1-4-wcsp	0.79	1.69	0.0	2.26	116.89	0.0
dsjc125-1-5-wcsp	0.94	2.91	0.0	3.28	450.93	0.0
geom30a-3-wcsp	0.14	0.04	0.0	1	2.44	0.0
geom30a-4-wcsp	0.21	0.08	0.0	1.36	13.08	0.0
geom30a-5-wcsp	0.24	0.14	0.0	1.51	47.45	0.0
geom30a-6-wcsp	0.33	0.2	0.0	2.12	169.42	0.0
geom40-2-wcsp	0.12	0.01	0.0	0.29	0.14	21.0
geom40-3-wcsp	0.14	0.03	0.0	0.58	1.02	0.0
geom40-4-wcsp	0.19	0.05	0.0	0.97	3.31	0.0
geom40-5-wcsp	0.26	0.08	0.0	1.23	13.9	0.0
geom40-6-wcsp	0.31	0.13	0.0	1.92	50.16	0.0
le450-5a-2-wcsp	1.88	2.95	0.0	16.99	TO	-
le450-5a-3-wcsp	2.42	16.59	0.0	37.75	TO	-
le450-5a-4-wcsp	2.86	34.48	0.0	60.58	TO	-
le450-5a-5-wcsp	3.67	119.36	0.0	TO	TO	-
myciel5g-3-wcsp	0.28	0.33	0.0	0.29	0.33	0.0
myciel5g-4-wcsp	0.39	0.26	0.0	0.42	0.61	0.0
myciel5g-5-wcsp	0.51	0.72	0.0	0.5	0.74	0.0
myciel5g-6-wcsp	0.61	1.11	0.0	0.67	1.33	0.0
queen5-5-3-wcsp	0.24	0.16	0.0	1.37	31.69	0.0
queen5-5-4-wcsp	0.31	0.34	0.0	2.09	TO	-
queen5-5-5-wcsp	0.37	0.52	0.0	3.5	TO	-
Série depot						
depot01ac-wcsp	0.3	0.23	906.66	1.43	14.61	1720.0
depot01bc-wcsp	0.31	0.2	1056.66	1.45	16.34	1965.0
depot01cc-wcsp	0.29	0.22	1374.0	1.64	13.05	2142.0
depot01c-wcsp	0.31	0.24	906.66	1.5	14.16	1720.0
Série dimacs						
bf0432-007-ext	1.32	0.45	0.0	1.51	0.44	0.0
bf2670-001-ext	1.72	0.45	0.0	1.35	0.38	0.0
dubois100-ext	0.16	0.0	0.0	0.16	0.0	0.0
dubois20-ext	0.07	0.01	0.0	0.07	0.01	0.0
dubois21-ext	0.06	0.01	0.0	0.06	0.0	0.0
dubois22-ext	0.07	0.01	0.0	0.07	0.01	0.0
dubois23-ext	0.07	0.01	0.0	0.06	0.01	0.0
dubois24-ext	0.07	0.01	0.0	0.07	0.0	0.0
dubois25-ext	0.07	0.01	0.0	0.08	0.0	0.0
dubois26-ext	0.08	0.0	0.0	0.07	0.0	0.0

	OSAC			**OTC$_2^w$**		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
dubois27-ext	0.07	0.0	0.0	0.07	0.01	0.0
dubois28-ext	0.07	0.0	0.0	0.07	0.01	0.0
dubois29-ext	0.09	0.0	0.0	0.09	0.01	0.0
dubois30-ext	0.08	0.0	0.0	0.07	0.01	0.0
dubois50-ext	0.11	0.0	0.0	0.1	0.01	0.0
pret150-25-ext	0.11	0.01	0.0	0.1	0.0	0.0
pret150-40-ext	0.09	0.01	0.0	0.1	0.01	0.0
pret150-60-ext	0.09	0.0	0.0	0.1	0.01	0.0
pret150-75-ext	0.1	0.01	0.0	0.13	0.0	0.0
pret60-25-ext	0.08	0.0	0.0	0.07	0.01	0.0
pret60-40-ext	0.06	0.01	0.0	0.07	0.01	0.0
pret60-60-ext	0.07	0.0	0.0	0.06	0.01	0.0
pret60-75-ext	0.07	0.01	0.0	0.07	0.0	0.0
ssa0432-003-ext	0.68	0.08	0.0	0.72	0.07	0.0
ssa2670-141-ext	1.02	0.18	0.0	1.23	0.21	0.0
Série driver						
driverlog01bc-wcsp	0.29	0.14	2143.75	1.19	8.59	2445.0
driverlog01cc-wcsp	0.28	0.15	1045.8	1.23	8.97	1117.0
driverlog01c-wcsp	0.29	0.12	995.0	1.23	8.8	1144.99
driverlog02ac-wcsp	2.78	35.1	430.12	TO	TO	-
driverlog02bc-wcsp	2.83	49.31	521.81	TO	TO	-
driverlog02cc-wcsp	2.99	47.65	617.55	TO	TO	-
driverlog02c-wcsp	2.72	36.17	430.12	TO	TO	-
driverlog04ac-wcsp	2.59	45.97	405.13	TO	TO	-
driverlog04bc-wcsp	2.67	38.74	449.05	TO	TO	-
driverlog04cc-wcsp	2.79	37.32	1263.22	TO	TO	-
driverlog04c-wcsp	2.52	48.14	405.13	TO	TO	-
driverlog05ac-wcsp	5.03	77.35	464.58	TO	TO	-
driverlog05bc-wcsp	4.65	84.68	450.55	TO	TO	-
driverlog05cc-wcsp	4.69	78.91	929.52	TO	TO	-
driverlog05c-wcsp	5.22	76.01	464.58	TO	TO	-
driverlog08ac-wcsp	14.92	158.33	512.66	TO	TO	-
driverlog08bc-wcsp	14.87	178.09	506.61	TO	TO	-
driverlog08cc-wcsp	14.82	173.92	1169.63	TO	TO	-
driverlog08c-wcsp	15.7	160.09	512.66	TO	TO	-
driverlog09-wcsp	31.51	TO	-	TO	TO	-
driverlogs03-wcsp	2.05	22.26	150.08	TO	TO	-
driverlogs06-wcsp	2.19	26.99	679.65	TO	TO	-
Série graphs						
graph05-wcsp	7.75	416.17	221.0	TO	TO	-
graph06-wcsp	16.18	TO	-	TO	TO	-

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
graph07-wcsp	7.82	207.08	4324.0	TO	TO	-
graph11reducmore-wcsp	7.82	257.4	247.0	TO	TO	-
graph11reduc-wcsp	8.2	272.95	247.0	TO	TO	-
graph11-wcsp	22.87	TO	-	TO	TO	-
graph12-wcsp	12.93	TO	-	TO	TO	-
graph13reducmore-wcsp	21.06	TO	-	TO	TO	-
graph13reduc-wcsp	21.51	TO	-	TO	TO	-
graph13-wcsp	42.3	TO	-	TO	TO	-
Série jnh						
jnh01-wcsp	1.36	0.07	0.0	1.35	0.15	0.0
jnh04-wcsp	0.87	0.05	0.0	0.9	0.04	0.0
jnh05-wcsp	1.29	0.09	0.0	1.25	0.07	0.0
jnh06-wcsp	1.18	0.07	0.0	1.1	0.08	0.0
jnh07-wcsp	0.76	0.03	0.0	0.79	0.04	0.0
jnh08-wcsp	1.45	0.09	0.0	1.53	0.1	0.0
jnh09-wcsp	0.87	0.05	0.0	0.86	0.04	0.0
jnh10-wcsp	1.11	0.06	0.0	1.26	0.08	0.0
jnh11-wcsp	1.11	0.05	0.0	1.02	0.11	0.0
jnh12-wcsp	0.96	0.04	0.0	0.94	0.05	0.0
jnh13-wcsp	1.23	0.08	0.0	1.25	0.08	0.0
jnh14-wcsp	1.06	0.04	0.0	1.14	0.08	0.0
jnh15-wcsp	1.31	0.08	0.0	1.3	0.08	0.0
jnh16-wcsp	1.16	0.07	0.0	1.16	0.06	0.0
jnh17-wcsp	1.01	0.05	0.0	1.05	0.04	0.0
jnh18-wcsp	1.4	0.08	0.0	1.41	0.09	0.0
jnh19-wcsp	0.96	0.04	0.0	1	0.08	0.0
jnh201-wcsp	1.12	0.06	0.0	0.99	0.06	0.0
jnh202-wcsp	0.9	0.04	0.0	0.93	0.04	0.0
jnh203-wcsp	1.21	0.08	0.0	1.35	0.13	0.0
jnh205-wcsp	1.1	0.05	0.0	1.11	0.05	0.0
jnh207-wcsp	0.84	0.03	0.0	0.81	0.04	0.0
jnh208-wcsp	1	0.05	0.0	1.13	0.07	0.0
jnh209-wcsp	1.11	0.04	0.0	1.05	0.06	0.0
jnh210-wcsp	0.78	0.04	0.0	0.82	0.03	0.0
jnh211-wcsp	1.18	0.08	0.0	1.16	0.08	0.0
jnh212-wcsp	0.91	0.04	0.0	0.85	0.04	0.0
jnh214-wcsp	1.19	0.07	0.0	1.05	0.07	0.0
jnh215-wcsp	0.89	0.04	0.0	0.92	0.06	0.0
jnh216-wcsp	1.14	0.09	0.0	1.16	0.07	0.0
jnh217-wcsp	1.04	0.04	0.0	1.12	0.06	0.0
jnh218-wcsp	0.87	0.04	0.0	0.86	0.04	0.0

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
jnh219-wcsp	1.3	0.08	0.0	1.34	0.12	0.0
jnh220-wcsp	0.99	0.04	0.0	0.98	0.04	0.0
jnh301-wcsp	1.09	0.08	0.0	1.2	0.08	0.0
jnh302-wcsp	0.97	0.04	0.0	1	0.04	0.0
jnh303-wcsp	1.33	0.11	0.0	1.43	0.1	0.0
jnh304-wcsp	0.94	0.04	0.0	0.84	0.05	0.0
jnh305-wcsp	1.27	0.11	0.0	1.29	0.11	0.0
jnh306-wcsp	0.9	0.04	0.0	0.94	0.06	0.0
jnh307-wcsp	1.2	0.07	0.0	1.2	0.07	0.0
jnh308-wcsp	1.05	0.06	0.0	1.06	0.05	0.0
jnh309-wcsp	1	0.05	0.0	0.97	0.05	0.0
jnh310-wcsp	1.3	0.08	0.0	1.4	0.1	0.0
Série logistics						
logistics01ac-wcsp	0.91	1.54	7100.0	6.53	TO	-
logistics01bc-wcsp	0.87	1.17	6444.83	6.85	TO	-
logistics01cc-wcsp	1.05	1.44	3171.91	6.84	TO	-
logistics01c-wcsp	0.86	1.56	7100.0	7.36	TO	-
Série mprime						
mprime01ac-wcsp	2.26	52.49	250.0	TO	TO	-
mprime01bc-wcsp	2.3	64.24	283.99	TO	TO	-
mprime01cc-wcsp	2.04	64.92	472.60	TO	TO	-
mprime01c-wcsp	2.11	53.24	250.0	TO	TO	-
mprime03ac-wcsp	0.27	0.15	320.0	1.23	35.89	320.0
mprime03bc-wcsp	0.28	0.15	380.0	1.4	36.21	379.99
mprime03cc-wcsp	0.27	0.14	344.0	1.52	33.84	344.0
mprime03c-wcsp	0.27	0.14	320.0	1.4	37.21	320.0
mprime04ac-wcsp	4.59	111.82	350.20	TO	TO	-
mprime04bc-wcsp	4.38	104.64	341.66	TO	TO	-
mprime04cc-wcsp	4.76	102.63	307.5625	TO	TO	-
mprime04c-wcsp	4.44	104.7	350.20	TO	TO	-
Série pedigree						
connell-wcsp	0.04	0.01	0.0	0.04	0.01	0.0
eye-wcsp	0.13	0.01	0.0	0.13	0.01	0.0
pedck1000-wcsp	0.67	0.02	0.0	0.95	0.01	0.0
pedck350l2-wcsp	0.33	0.0	0.0	0.33	0.01	0.0
pedck350l3-wcsp	0.32	0.01	0.0	0.33	0.01	0.0
pedck350-wcsp	0.32	0.01	0.0	0.31	0.01	0.0
pedck60-L12-wcsp	0.11	0.0	0.0	0.12	0.0	0.0
pedck60-L1-wcsp	0.12	0.0	0.0	0.11	0.0	0.0
pedck60-L2-wcsp	0.1	0.0	0.0	0.12	0.0	0.0
saudiarabia-wcsp	0.13	0.0	0.0	0.12	0.0	0.0

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
sheep4nr-wcsp	38.97	1.08	0.0	31.54	0.74	0.0
sheep4r-4-0-wcsp	3.17	0.35	0.0	3.55	0.34	0.0
sheep4r-4-1-wcsp	3.54	0.43	0.0	3.61	0.5	0.0
sheep4r-4-2-wcsp	3.5	0.31	0.0	3.13	0.29	0.0
sheep4r-4-3-wcsp	3.64	0.35	0.0	3.52	0.3	0.0
sheep4r-wcsp	34.54	1.06	0.0	32.63	1.07	0.0
simple-wcsp	0.04	0.0	0.0	0.06	0.01	0.0
sobel-wcsp	0.06	0.0	0.0	0.04	0.01	0.0
wijsmanguo-wcsp	0.29	0.0	0.0	0.29	0.01	0.0
Série rover						
rovers02ac-wcsp	1.34	17.56	648.75	37.88	TO	-
rovers02bc-wcsp	1.33	16.53	692.49	47.02	TO	-
rovers02cc-wcsp	1.35	18.67	531.21	TO	TO	-
rovers02c-wcsp	1.32	17.94	648.75	39.27	TO	-
Série satellite						
satellite01ac-wcsp	1.2	8.3	466.0	46.51	TO	-
satellite01bc-wcsp	1.17	7.25	613.89	48.95	TO	-
satellite01cc-wcsp	1.21	8.09	466.0	TO	TO	-
satellite01c-wcsp	1.2	7.04	466.0	44.49	TO	-
satellite02ac-wcsp	2.57	60.59	557.23	TO	TO	-
satellite02bc-wcsp	2.56	50.75	634.48	TO	TO	-
satellite02cc-wcsp	2.21	51.53	1194.39	TO	TO	-
Série scens						
scen06-16reduc-wcsp	3.33	118.64	3.49	TO	TO	-
scen06-16-wcsp	3.87	130.47	3.49	TO	TO	-
scen06-18reduc-wcsp	2.99	82.79	3.50	TO	TO	-
scen06-18-wcsp	3.3	110.26	3.49	TO	TO	-
scen06-20reduc-wcsp	2.71	76.14	3.49	TO	TO	-
scen06-20-wcsp	2.93	95.34	3.49	TO	TO	-
scen06-22reduc-wcsp	2.37	53.97	3.50	TO	TO	-
scen06-22-wcsp	2.69	48.87	3.49	TO	TO	-
scen06-24reduc-wcsp	2.03	38.64	3.0	TO	TO	-
scen06-24-wcsp	2.29	35.75	3.0	TO	TO	-
scen06-30reduc-wcsp	1.4	8.37	3.0	43.25	TO	-
scen06-30-wcsp	1.55	15.1	2.99	56.49	TO	-
scen06reduc-wcsp	7.25	362.77	3.50	TO	TO	-
scen06-wcsp	8.28	TO	-	TO	TO	-
scen07-10000-30r-wcsp	2.1	106.33	24999.99	67.81	TO	-
scen07reduc-wcsp	11.88	TO	-	TO	TO	-
scen07-wcsp	17.14	TO	-	TO	TO	-
scen08reduc-wcsp	30.29	TO	-	TO	TO	-

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
scen08-wcsp	31.28	TO	-	TO	TO	-
scen09-wcsp	8.74	121.87	10756.5	TO	TO	-
scen10-wcsp	8.12	TO	-	TO	TO	-
Série spot5						
spot5-1401-wcsp	6.1	20.95	165553.0	TO	TO	-
spot5-1403-wcsp	17.49	25.96	165606.5	TO	TO	-
spot5-1405-wcsp	39.51	44.41	165645.5	TO	TO	-
spot5-1407-wcsp	37.44	49.67	165675.5	TO	TO	-
spot5-1502-wcsp	0.33	0.04	26040.0	0.81	2	28041.33
spot5-1504-wcsp	2.23	3.65	94621.5	26.36	TO	-
spot5-1506-wcsp	25.28	27.44	138175.5	TO	TO	-
spot5-28-wcsp	2.22	5.6	105067.5	TO	TO	-
spot5-29-wcsp	0.37	0.29	7038.5	3.76	436.94	8051.83
spot5-404-wcsp	0.47	0.36	67.0	4.14	305.22	94.66
spot5-408-wcsp	0.98	1.36	4123.0	15.24	TO	-
spot5-412-wcsp	1.75	3.86	17161.5	104.13	TO	-
spot5-414-wcsp	4.76	6.74	20184.0	TO	TO	-
spot5-42-wcsp	0.84	1.19	72549.5	14.33	TO	-
spot5-503-wcsp	0.43	0.15	7573.0	1.92	23.3	11103.66
spot5-505-wcsp	1.09	1.16	12118.5	13.98	TO	-
spot5-507-wcsp	2.36	3.29	15167.0	58.03	TO	-
spot5-509-wcsp	4.18	5.63	19179.0	131.71	TO	-
spot5-54-wcsp	0.28	0.19	24.5	1.81	80.29	31.66
spot5-5-wcsp	2.86	16.08	63.0	TO	TO	-
spot5-8-wcsp	0.06	0.0	2.0	0.14	0.04	2.0
Série warehouse						
cap101-wcsp	1.75	18.42	7966472.0	2	18.01	7966472.0
cap102-wcsp	1.77	20.89	8547029.0	1.8	21.78	8547029.0
cap103-wcsp	1.79	21.6	8937809.0	1.97	22.32	8937809.0
cap104-wcsp	1.91	17.19	9289407.0	2.01	19.71	9289407.0
cap111-wcsp	4.43	314	7934385.0	4.68	245.7	7934385.0
cap112-wcsp	4.97	253.84	8514942.0	5.02	297.09	8514942.0
cap113-wcsp	4.3	261.06	8930757.0	4.63	241.93	8930757.0
cap114-wcsp	4.47	348.95	9289407.0	4.91	257.19	9289407.0
cap121-wcsp	4.73	287.76	7934385.0	4.81	247.22	7934385.0
cap122-wcsp	4.72	309.01	8514942.0	4.89	250.99	8514942.0
cap123-wcsp	4.37	304.05	8930757.0	4.61	243.05	8930757.0
cap124-wcsp	4.53	331.41	9289407.0	4.53	261.02	9289407.0
cap131-wcsp	4.45	275.18	7934385.0	5.04	257.34	7934385.0
cap132-wcsp	4.58	246.75	8514942.0	4.6	254.28	8514942.0
cap133-wcsp	4.69	241.45	8930757.0	4.88	254.6	8930757.0

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
cap134-wcsp	4.46	286.83	9289407.0	4.5	268.97	9289407.0
cap41-wcsp	1.18	4.34	9326144.0	1.09	4.53	9326144.0
cap42-wcsp	1.18	4.86	9777981.0	1.27	4.13	9777981.0
cap43-wcsp	1.23	4.93	10106402.0	1.37	4.61	10106402.0
cap44-wcsp	1.2	4.79	10349757.0	1.26	4.66	10349757.0
cap51-wcsp	1.19	4.54	10106402.0	1.2	4.62	10106402.0
cap61-wcsp	1.12	4.28	9326144.0	1.22	4.31	9326144.0
cap62-wcsp	1.14	4.83	9777981.0	1.33	4.66	9777981.0
cap63-wcsp	1.21	4.54	10106402.0	1.26	4.71	10106402.0
cap64-wcsp	1.14	5.22	10349757.0	1.17	4.79	10349757.0
cap71-wcsp	1.13	4.05	9326144.0	1.18	4.42	9326144.0
cap72-wcsp	1.12	5	9777981.0	1.15	4.1	9777981.0
cap73-wcsp	1.18	4.95	10106402.0	1.2	5.31	10106402.0
cap74-wcsp	1.13	4.89	10349757.0	1.18	4.64	10349757.0
cap81-wcsp	1.79	19.91	7966472.0	1.84	17.86	7966472.0
cap82-wcsp	1.92	21.62	8547029.0	1.99	19.86	8547029.0
cap83-wcsp	1.84	20.57	8937809.0	1.89	17.59	8937809.0
cap84-wcsp	1.79	18.68	9289407.0	2.03	19.07	9289407.0
cap91-wcsp	2.01	16.56	7966472.0	1.95	19.33	7966472.0
cap92-wcsp	1.81	21.16	8547029.0	1.83	19.13	8547029.0
cap93-wcsp	1.85	22.7	8937809.0	1.87	23.66	8937809.0
cap94-wcsp	1.83	18.95	9289407.0	1.92	18.16	9289407.0
capa-wcsp	TO	TO	-	TO	TO	-
capb-wcsp	TO	TO	-	TO	TO	-
capc-wcsp	TO	TO	-	TO	TO	-
capmo1-wcsp	30.93	TO	-	32.44	TO	-
capmo2-wcsp	34.71	TO	-	36.99	TO	-
capmo3-wcsp	34.37	TO	-	37.62	TO	-
capmo4-wcsp	33.96	TO	-	35.15	TO	-
capmo5-wcsp	34.78	TO	-	36.87	TO	-
capmp1-wcsp	TO	TO	-	TO	TO	-
capmp2-wcsp	TO	TO	-	TO	TO	-
capmp3-wcsp	TO	TO	-	TO	TO	-
capmp4-wcsp	TO	TO	-	TO	TO	-
capmp5-wcsp	TO	TO	-	TO	TO	-
capmq1-wcsp	TO	TO	-	TO	TO	-
capmq2-wcsp	TO	TO	-	TO	TO	-
capmq3-wcsp	TO	TO	-	TO	TO	-
capmq4-wcsp	TO	TO	-	TO	TO	-
capmq5-wcsp	TO	TO	-	TO	TO	-
Série zenotravel						

	OSAC			OTC_2^w		
	TG	**TR**	**LB**	**TG**	**TR**	**LB**
zenotravel02ac-wcsp	1.46	20.97	1935.0	62.32	TO	-
zenotravel02bc-wcsp	1.36	24.17	1760.0	59.45	TO	-
zenotravel02cc-wcsp	1.34	22.99	608.5	66.65	TO	-
zenotravel02c-wcsp	1.32	21.77	1935.0	63.45	TO	-
zenotravel04ac-wcsp	2.95	237.94	1907.5	TO	TO	-
zenotravel04bc-wcsp	2.94	246.46	2016.25	TO	TO	-
zenotravel04cc-wcsp	2.79	225.67	959.75	TO	TO	-
zenotravel04c-wcsp	3.09	232.16	1907.5	TO	TO	-

TABLE 4.2: Résultats obtenus pour différentes séries OSAC vs OTC_2^w (une échéance de 1,200 secondes par instance)

4.6 Conclusion

Dans ce chapitre, nous avons étudié quelques pistes concernant l'utilisation d'une opération de transferts de coûts généralisant les opérations classiques. Celle-ci permet le transfert de coûts entre deux contraintes toutes deux non unaires. Cela nous a permis de définir une nouvelle cohérence, TC, ainsi que sa version optimale, OTC. Nous avons proposé deux versions d'algorithmes qui permettent d'établir la propriété TC. Ensuite, nous nous nous somme intéressés à la version optimisée OTC. Nous avons écrit un nouveau problème linéaire qui permet d'établir OTC. Nous avons montré expérimentalement que la propriété OTC telle qu'elle est maintenant est difficilement utilisable en pratique. Un travail futur consiste à trouver une nouvelle écriture du problème linéaire où nous réduisons les transferts de coût afin d'améliorer le temps CPU de résolution du PL. Il nous reste également à étudier comment mettre en œuvre de manière efficace la cohérence TC au cours de la recherche en s'inspirant des algorithmes de maintien des cohérences classiques (AC, DAC, EDAC, ...).

Chapitre 5

Substituabilité de tuples au voisinage pour le cadre WCSP

Sommaire

La notion de substituabilité présentée en chapitre 3 peut être facilement étendue aux tuples. Des travaux ont déjà été effectués dans [de Givry et al., 2013; Georgiev et al., 2006b] qui utilisent la notion de substituabilité pour supprimer des tuples binaires. Dans ce qui suit, nous définissons la substituabilité de tuple au voisinage pour le cadre WCSP. Ensuite, nous étendons la propriété *pcost* (voir chapitre 3) aux tuples. Nous montrons que *pcost* permet d'identifier efficacement des tuples qui sont *souples-substituables*. Enfin, nous proposons un algorithme qui combine la cohérence de tuple TC et *pcost* pour éliminer efficacement les tuples souples-substituables.

5.1 Définitions et notations

Avant de définir la substituabilité souple pour les tuples dans le cadre WCSP, nous introduisons quelques définitions que nous utiliserons par la suite. Nous rappelons qu'un tuple est un ensemble de couples (variable,valeur).

Définition 80. *Pour toute instanciation I et pour tout tuple t, nous désignons par $I \triangleright t$ l'instanciation $I[vars(I) \setminus vars(t)] \cup t$ qui combine I et t en donnant la priorité à t.*

Autrement dit, l'instanciation $I \triangleright t$ est obtenue à partir de l'instanciation I soit par la substitution des valeurs affectées à $vars(t)$ dans I par t, soit par l'extension de I avec t. Par exemple, étant données deux instanciations $I = \{(x,a),(y,a),(z,a)\}$, $I' = \{(w,a),(z,a)\}$ et un tuple $t = \{(x,b),(y,b)\}$, la substitution des valeurs affectées à $vars(t)$ dans I (resp. I') par t est $I \triangleright t = \{(x,b),(y,b),(z,a)\}$ (resp. $I' \triangleright t = \{(w,a),(x,b),(y,b),(z,a)\}$).

Définition 81 (Voisinage d'un ensemble de variables)**.** *Pour tout ensemble de variables S, l'ensemble des contraintes voisines de S est désigné par $\Gamma(S) = \{w_{S'} \in \mathscr{W} | S \cap S' \neq \emptyset\}$; Autrement dit, $\Gamma(S) = \cup_{x \in S} \Gamma(x)$.*

Comme pour le voisinage d'une variable, le voisinage d'un ensemble de variables S est dit séparable si toutes ses contraintes ne partagent pas d'autres variables que l'ensemble S. Par exemple, considérons l'exemple d'un WCN avec les deux contraintes w_{txyz} et w_{xyz}. Il est clair que le voisinage de l'ensemble $S = \{x, y\}$ n'est pas séparable car les deux contraintes w_{txyz} et w_{xyz} partagent plus de variables que l'ensemble S (la variable z en plus). En revanche, le voisinage de l'ensemble $S = \{x, y, z\}$ est bien séparable car les deux contraintes partagent au plus l'ensemble S.

Définition 82. *Soient $S \in \mathscr{X}$ un ensemble de variables et $t, t' \in l(S)$ deux tuples distincts. Le tuple t est* souple-substitutable *au tuple t' dans P si et seulement si pour toute instanciation complète I de P, $cost(I \rhd t) \leq cost(I \rhd t')$.*

En d'autres termes, un tuple t est souple-substituable à un tuple t' si toute instanciation complète I extension de t' a un coût plus élevé que $I \rhd t$. Lorsqu'un tuple t est souple-substituable à t' dans P, t' peut être considéré comme un tuple interdit (c'est à dire son coût marqué à k) sans changer le coût optimum de P. En revanche, on peut perdre certaines solutions contenant le tuple t'.

Définition 83. *Soient $S \in \mathscr{X}$ un ensemble de variables et $t, t' \in l(S)$ deux tuples distincts de l'ensemble S. Le tuple t est* souple-substitutable *au tuple t' au voisinage dans P si et seulement si pour toute instanciation complète I de P, $cost_{\Gamma(S)}(I \rhd t) \leq cost_{\Gamma(S)}(I \rhd t')$.*

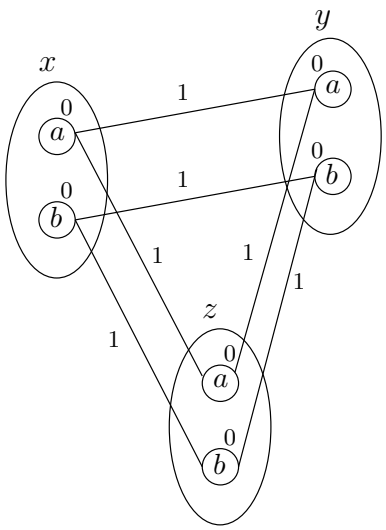
On dira que t' est *SNST-éliminable* (dans P) quand il existe un tuple t souple-substituable à t' au voisinage. Bien évidement, la substituabilité souple au voisinage implique la substituabilité souple (complète) mais l'inverse n'est pas vrai. Pour bien comprendre la substituabilité souple au voisinage pour les tuples, considérons un WCN composé de trois variables x, y et z avec $dom(x) = dom(y) = dom(z) = \{a, b\}$ et trois contraintes w_{xy}, w_{xz} et w_{yz}. Une illustration de ce WCN est donnée par la figure 5.1(a).

Notons que le WCN donné par la figure 5.1(a) est OSAC-cohérent et qu'il ne contient pas de valeurs souple-substituables au voisinage. Il est assez facile de constater à l'aide du tableau 5.1 que le tuple $t_2 = \{(x, a), (y, b)\}$ est souple-substituable aux tuples $t_1 = \{(x, a), (y, a)\}$, $t_3 = \{(x, b), (y, a)\}$ et $t_4 = \{(x, b), (y, b)\}$ au voisinage.

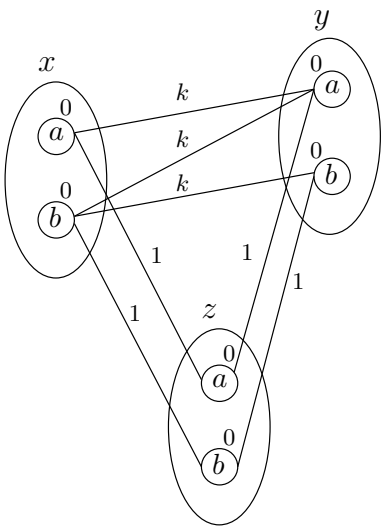
$I \setminus t \mid vars(t) = \{x, y\}$	$cost_{\Gamma(S)}(I \rhd t_1)$	$cost_{\Gamma(S)}(I \rhd t_2)$	$cost_{\Gamma(S)}(I \rhd t_3)$	$cost_{\Gamma(S)}(I \rhd t_4)$
(z, a)	3	**1**	1	1
(z, b)	1	**1**	1	3

TABLE 5.1 – Le tuple $t_2 = \{(x, a), (y, b)\}$ est souple-substituable aux tuples $t_1 = \{(x, a), (y, a)\}$, $t_3 = \{(x, b), (y, a)\}$ et $t_4 = \{(x, b), (y, b)\}$ au voisinage

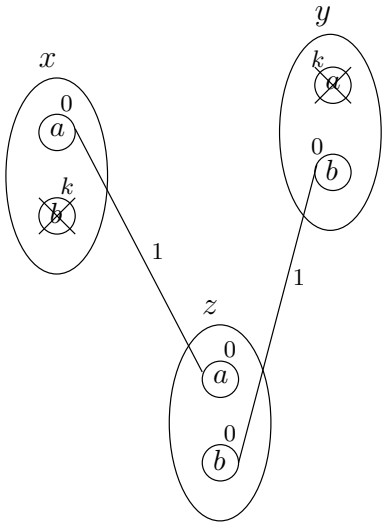
Par conséquent, les tuples t_1, t_3 et t_4 peuvent être interdits et leur coût est marqué à k comme le montre la figure 5.1(b). Nous remarquons aussi sur la contrainte w_{xy} que les valeurs (x, b) et (y, a) sont interdites car tous les tuples qui les contiennent ont un coût de k. Ces valeurs peuvent être supprimées sans changer le coût optimum du WCN. Le WCN obtenu après ces opérations est donné par la figure 5.1(c). Sur ce dernier, nous remarquons que les deux valeurs (z, a) et (z, b) sont interchangeables, par conséquent une des deux valeurs peut être supprimée



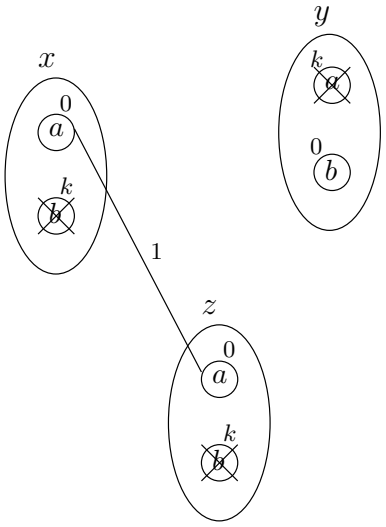
(a) Un WCN P OSAC-cohérent qui ne contient pas de valeurs souple-substituables au voisinage



(b) Le tuple $\{(x,a),(y,b)\}$ est souple-substituable aux tuples $\{(x,a),(y,a)\}$, $\{(x,b),(y,a)\}$ et $\{(x,b),(y,b)\}$ au voisinage



(c) Les valeurs (x,b) et (y,a) sont interdites sur la contrainte w_{xy}



(d) La valeur (z,a) est souple-substituable à la valeur (z,b) au voisinage

FIGURE 5.1 – La substituabiluté souple au voisinage

(par exemple (x,a) voir la figure 5.1(c)). Enfin, après toutes ces opérations, nous obtenons le WCN de la figure 5.1(d) qui représente une solution au WCN de départ.

Cet exemple permet de conclure immédiatement le lien qui existe entre les différentes cohérences que nous avons présentées dans l'état de l'art et la substituabilité souple au voisinage pour les tuples. Avant d'illustrer ce lien, nous commençons d'abord par introduire la notion de

la fermeture par substituabilité souple au voisinage pour les tuples.

Définition 84. *La fermeture par substituabilité souple pour les tuples au voisinage (ou SNST-closure) d'un WCN P, noté* $\texttt{SNST}(P)$*, est tout WCN obtenu après l'élimination itérative des tuples SNST-éliminables jusqu'à ce qu'un point fixe soit atteint.*

Proposition 21. *Soit P un WCN OSAC-cohérent.* $\texttt{SNST}(P)$ *n'est pas nécessairement OSAC-cohérent.*

Preuve. Le WCN P représenté par la figure 5.1(a) est OSAC-cohérent. En revanche, il existe une SNST-closure $P' = \texttt{SNST}(P)$ (voir la figure 5.1(d)) qui n'est pas OSAC-cohérent. En effet, Il existe une transformation SAC qui peut augmenter la borne inférieure $w_\emptyset(P')$ d'une unité de coût. □

5.2 Calcul de la substituabilité souple pour les tuples

La substituabilité souple pour les tuples est une généralisation de la substituabilité souple pour les valeurs. Nous utiliserons donc le même principe que pcost pour définir un mécanisme qui nous permettra de calculer les tuples souples substituables.

Définition 85. *Soient* $S \in \mathscr{X}$ *un ensemble de variables et* $t, t' \in l(S)$ *deux tuples distincts,*

- *la* paire de surcoût *de* t' *vis à vis de* t *dans* $w_{S'} \in \Gamma(S)$ *est définie par :*
$$pcost(w_{S'}, t \to t') = min_{I \in l(S')}\{(w_{S'}(I \rhd t'), w_{S'}(I \rhd t))\}\,;$$
- *la* paire de surcoût *de* t' *vis à vis de* t *dans* P *est définie par :*
$$pcost(t \to t') = \sum_{w_{S'} \in \Gamma(x)} pcost(w_{S'}, t \to t').$$

Proposition 22. *Soient* S *un ensemble de variables et* $t, t' \in l(S)$ *deux tuples distincts. Si* $pcost(S : t \to t') \geq 0$ *alors le tuple* t *est souple-substituable à* t' *au voisinage dans* P*.*

Preuve. Pour une contrainte $w_{S'} \in \Gamma(S)$, soit $I^{w_{S'}}$ l'instanciation de $S' - \{S \cap S'\}$ qui donne la paire de surcoût minimale dans $min_{I \in l(S')}\{(w_{S'}(I \rhd t'), w_{S'}(I \rhd t))\}$. Par définition, $pcost(w_{S'}, t \to t') = (w_{S'}(I^{w_{S'}} \rhd t'), w_{S'}(I^{w_{S'}} \rhd t))$. Par définition de min sur les paires de surcoût, nous avons $\forall I, \forall w_{S'} \in \Gamma(S)$, $(w_{S'}(I \rhd t'), w_{S'}(I \rhd t)) \geq (w_{S'}(I^{w_{S'}} \rhd t'), w_{S'}(I^{w_{S'}} \rhd t))$. Par sommation, nous obtenons $\forall I, \sum_{w_{S'} \in \Gamma(S)} (w_{S'}(I \rhd t'), w_{S'}(I \rhd t)) \geq \sum_{w_{S'} \in \Gamma(S)} (w_{S'}(I^{w_{S'}} \rhd t'), w_{S'}(I^{w_{S'}} \rhd t))$. Par hypothèse, $pcost(t \to t') \geq 0$, donc nous avons $\sum_{w_{S'} \in \Gamma(S)} (w_{S'}(I^{w_{S'}} \rhd t'), w_S(I^{w_{S'}} \rhd t)) \geq 0$, et donc $\forall I, \sum_{w_{S'} \in \Gamma(S)} (w_{S'}(I \rhd t'), w_{S'}(I \rhd t)) \geq 0$. Par définitions de $+$ et $\leq$ sur les paires de surcoût, nous pouvons tirer $\forall I, \sum_{w_{S'} \in \Gamma(S)} w_{S'}(I \rhd t') \geq \sum_{w_{S'} \in \Gamma(S)} w_{S'}(I \rhd t)$ ce qui implique $\forall I, min(k, \sum_{w_{S'} \in \Gamma(S)} w_{S'}(I \rhd t')) \geq min(k, \sum_{w_{S'} \in \Gamma(S)} w_{S'}(I_{t=}))$. Puisque $\forall a_i \in \{0, \ldots, k\}, a_1 \oplus \ldots \oplus a_n = min(k, a_1 + \ldots a_n\}$, nous pouvons conclure que $\forall I, \bigoplus_{w_{S'} \in \Gamma(S)} w_{S'}(I \rhd t'))$ $\geq \bigoplus_{w_{S'} \in \Gamma(S)} w_{S'}(I \rhd t))$. Ainsi, le tuple t est souple-substituable au tuple t' au voisinage dans P. □

Comme pour le cas de substituabilité souple des valeurs, la réciproque de la proposition 22 n'est pas vraie lorsque le voisinage de l'ensemble S est non séparable ou quand $k \neq +\infty$.

Exemple 9. *Soient cinq variables x, y, z, t et w telles que* $dom(x) = dom(y) = dom(z) = \{a, b\}$, $dom(t) = dom(w) = \{a\}$*, et deux contraintes ternaires* w_{xyz}, w_{xyt} *définies par la table de coûts suivante :*

x	y	z	t	w	w_{xyzw}	w_{xyzt}
a	a	a	a	a	1	0
a	a	b	a	a	0	1
a	b	a	a	a	0	0
a	b	b	a	a	0	0
b	a	a	a	a	0	1
b	a	b	a	a	1	0
b	b	a	a	a	0	0
b	b	b	a	a	0	0

Considérons l'ensemble de variables $S = \{x, y\}$ *et les deux tuples* $t = \{(x, a), (y, a)\}$, $t' = \{(x, b), (y, a)\}$ *Il est clair que t est souple-substituable à* t' *au voisinage. En effet, il est assez facile de constater que pour toute instanciation complète I,* $cost(I \triangleright t') \geq cost(I \triangleright t)$. *En revanche,* $pcost(w_{xyzt}, t \to t') = pcost(w_{xyzw}, t \to t') = (0, 1)$ *et, par conséquent* $pcost(t \to t') = (0, 2) \not\geq 0$.

L'exemple 9 montre que pour que la réciproque de la proposition 22 tienne soit vraie il faut que $\Gamma(x)$ soit séparable. Le deuxième cas de figure , où la réciproque de la proposition 22 est fausse quand $k \neq +\infty$. Rappelons que pour la substituabilité souple au voisinage des valeurs nous avons montré avec un exemple (voir l'exemple 6) que la réciproque de la proposition 2 est fausse est quand $k \neq +\infty$. Il est assez facile d'étendre l'exemple 6 pour la substituabilité souple au voisinage des tuples. Considérant le WCN de l'exemple 10

Exemple 10. *Considérons les deux familles de contraintes* $W_i = \{w_i \mid i \in 1..n\}$ *et* $W'_i = \{w'_i \mid i \in 1..n'\}$ *définies par :*

q	x	y_i	w_i
a	a	c	0
a	b	c	1

q	x	z_i	w'_i
a	a	d	1
a	b	d	0

Si nous nous intéressons aux deux tuples $t = \{(q, a), (x, a)\}$ et $t' = \{(q, a), (x, b)\}$ au lieu des deux valeurs (x, a) et (x, b) nous obtenons le même problème que dans l'exemple 6. En effet, avec $n = k$ et $n' = k + 1$, nous pouvons observer que $pcost(t \to t') = (n, n') = (k, k + 1) \not\geq 0$. Cependant, t et t' sont tous deux interdits et par conséquent t est souple-substituable à t' (et inversement).

Proposition 23. *Soient S un ensemble de variables et* $t, t' \in l(S)$ *tel que* $\Gamma(S)$ *est séparable et* $pcost(t \to t') = (\beta, \alpha)$ *avec* $\alpha < k$. *Si le tuple t est souple-substituable au tuple t au voisinage dans P alors* $pcost(t \to t') \geq 0$.

Preuve. Puisque par hypothèse $\Gamma(S)$ est séparable, nous pouvons définir l'instanciation I^{min} dans $vars(\Gamma(S)) \setminus \{S\}$ comme l'union pour chaque contrainte $w_{S'} \in \Gamma(S)$ de l'instanciation $I^{w_{S'}}$ définie dans la preuve de la proposition 22. I^{min} est telle que $pcost(w_{S'}, t \to t') = (w_{S'}(I^{min} \triangleright t'), w_{S'}(I^{min} \triangleright t))$.

Par hypothèse, $\forall I, cost_{\Gamma(S)}(I \triangleright t') \geq cost_{\Gamma(S)}(I \triangleright t)$, ce qui peut être réécrit sous la forme $\forall I, \bigoplus_{w_{S'} \in \Gamma(S)} w_{S'}(I \triangleright t') \geq \bigoplus_{w_{S'} \in \Gamma(S)} w_{S'}(I \triangleright t)$. En particulier, cela est vrai pour $I = I^{min}$.

Par conséquent, $\bigoplus_{w_{S'} \in \Gamma(S)} w_{S'}(I^{min} \rhd t') \geq \bigoplus_{w_{S'} \in \Gamma(S)} w_{S'}(I^{min} \rhd t)$ qui peut être réécrit sous la forme $min(k,\beta) \geq min(k,\alpha)$ où $\beta = \sum_{w_{S'} \in \Gamma(S)} w_{S'}(I^{min} \rhd t'))$ et $\alpha = \sum_{w_{S'} \in \Gamma(S)} w_{S'}(I^{min} \rhd t)$. Par définition, $pcost(t \rightarrow t') = (\beta, \alpha)$ et donc $pcost(t \rightarrow t') \geq 0$ si et seulement si $\beta \geq \alpha$. Maintenant, si $\alpha < k$, $min(k,\alpha) = \alpha$ et $min(k,\beta) \geq min(k,\alpha) \Rightarrow \beta \geq \alpha \Rightarrow pcost(t \rightarrow t') \geq 0$ (ceci est vrai pour $\beta < k$ et $\beta \geq k$). Notons que lorsque $\alpha \geq k$, $min(k,\beta) \geq min(k,\alpha) \not\Rightarrow \beta \geq \alpha$; un contre-exemple étant $\beta = k$ et $\alpha = k + 1$. □

5.3 Algorithme

Algorithme 24 : `pcost`

Entrées : w_S : Contrainte, t, t' : Valeur
Sorties : Paire de surcoût

```
1  pcst ← (1, 0) ;
2  pour chaque I ∈ l(S \ {vars(t)}) faire
3      si (w_S(I ▷ t'), w_S(I ▷ t)) < pcst alors
4          pcst ← (w_S(I ▷ t'), w_S(I ▷ t)) ;
5      fin
6  fin
7  retourner pcst
```

Algorithme 25 : `pcost`

Entrées : S : Scope, t, t' : Tuple
Sorties : Paire de surcoût

```
1  pcst ← (w_S(t'), w_S(t)) ;
2  si pcst < 0 alors retourner pcst;
3  pcst ← pcst + pcost(residues[S, t, t'], t → t') ;
4  si pcst < 0 alors retourner pcst;
5  pour chaque w_S' ∈ Γ(S) | w_S' ≠ residues[S, t, t'] faire
6      d ← pcst(w_S', t → t') ;
7      si d < (w_S(t), w_S(t')) alors residues[S, t, t'] ← w_S' ;
8      pcst ← pcst + d ;
9      si pcst < 0 alors retourner pcst;
10 fin
11 retourner pcst;
```

L'algorithme que nous présentons dans cette section représente une généralisation de l'algorithme de la section 3.6. Le principe est toujours de commencer à identifier les valeurs SNS-éliminables à partir d'un réseau qui est cette fois $TC1_r$-cohérent pour réduire l'effort de calcul. Rappelons que $TC1_r$ est une généralisation de l'algorithme AC. Donc pour $r = 1$ l'algorithme que nous proposons dans cette section est équivalent à celui de la section 3.6.

Algorithme 26 : PSNST

Entrées : P : WCN TC1$_r$-cohérent

```
1  Δ ← ∅ ;
2  pour chaque S ⊆ 𝒳 | |S| = r faire
3      si ∃S′ ⊂ vars(Γ(S)) | stamp[S′] > substamp alors
4          pour chaque (t, t′) ∈ l(S)² | t′ > t faire
5              si pcost(t → t′) ≥ 0 alors
6                  Δ ← Δ ∪ t′ ;
7              sinon
8                  si pcost(t′ → t) ≥ 0 alors Δ ← Δ ∪ t ;
9              fin
10         fin
11     fin
12 fin
13 substamp ← time++ ;
14 pour chaque t ∈ Δ faire
15     marqué le coût de t à k sur la contrainte w_(vars(t)) ;
16     Q ← Q ∪ vars(t) ;
17     stamp[vars(t)] ← time++ ;
18 fin
```

Algorithme 27 : TC1$_r$-PSNST

Entrées : P : WCN

Sorties : P : WCN rendu TC1$_r$-cohérent et PSNST-clos

```
1  time ← 0 ;
2  substamp ← −1 ;
3  stamp[S] ← 0, ∀S ⊆ 𝒳 | |S| = r ;
4  Q ← S ⊆ 𝒳 | |S| = r ;
5  répéter
6      PSNST(W-TC1_r(P, Q)) ;
7  jusqu'à Q ≠ ∅ ;
```

À partir de maintenant, nous considérons uniquement les sous-ensembles S de variables d'arité égal à r. La procédure principale est l'algorithme 27. Nous utilisons une queue notée Q pour stocker les sous-ensembles de variables S dont l'ensemble des tuples autorisé $l(S)$ a été récemment changé. La queue Q est initialisée à la ligne 4 avec les sous-ensembles S. À la ligne 6, l'algorithme classique TC1$_r$ est exécuté avant de solliciter la fonction PSNST. Les appels de TC1$_r$ et de PSNST sont entrelacés jusqu'à ce que la queue Q deviens vide.

La fonction PSNST, algorithme 26, itère sur tous les sous-ensembles S de $\mathscr{X}$ afin de recueillir les tuples SNST-éliminables dans un ensemble appelé Δ. Cet ensemble est initialisé à la ligne 1 et mis à jour aux lignes 6 et 10. Si tous les tuples SNS-éliminables pour un

sous-ensemble S ont été supprimés, et que les tuples des sous-ensembles de variables dans le voisinage de S restent identiques, alors, il n'est pas nécessaire de considérer à nouveau S pour rechercher des valeurs SNS-éliminables. C'est l'objet de la ligne 3. Nous utilisons le même mécanisme d'horodatage que dans l'algorithme 18

L'algorithme 17 nous permet de calculer la paire de surcoût *pcost* de t' par rapport à t. Puisque nous savons que le WCN est TC1$_r$-cohérent, nous avons la garantie que la paire de surcoût de t' par rapport à t dans toute contrainte non-unaire $w_{S'}$ voisine à S est inférieur où égal à $(0, 0)$. Cela signifie que nous ne pourrons jamais compenser une paire de surcoût négative avec une une paire de surcoût positive (une fois que la paire de surcoût unaire a été pris en compte). Un grand avantage de cette observation est la possibilité d'utiliser les arrêts précoces de boucle au cours de tels calculs. Cette opération est effectuée aux lignes 2, 4 et 9. Les résidus sont un autre mécanisme introduit pour augmenter la performance de l'algorithme. Pour chaque sous-ensemble S, et pour tout couple (a, b) de valeurs de $l(S)$, nous stockons dans `residues`$[S, a, b]$ la contrainte $w_{S'}$ qui garantit que t' n'est pas SNST-éliminable par t, si elle existe. La contrainte résiduelle est prioritaire (lignes 3-4) ; de cette façon, si elle permet de compenser le surcoût initial unaire, elle nous évite tout travail supplémentaire. Elle est mise à jour aux lignes 7. Notons que nous pouvons initialiser le tableau `residues` avec n'importe quelle contrainte arbitraire et que l'algorithme 16 retourne nécessairement une paire de surcoût inférieure ou égale à $(0, 1)$ (ce qui explique l'initialisation de `pcst` à $(0, 1)$ à la ligne 1).

5.4 Conclusion

Dans ce chapitre, nous avons étudié la notion de substituabilité de tuples dans le cadre WCSP. Nous avons montré que la substituabilité de tuples est une généralisation de la substituabilité de valeurs vue au chapitre 3. Nous avons modifié la propriété *pcost* pour permettre d'identifier efficacement des tuples qui sont souples-substituables. Enfin, nous avons proposé un algorithme qui combine la cohérence de tuples TC et *pcost* pour éliminer efficacement les tuples souples-substituables.

Conclusion générale

Dans ce travail de thèse, nous nous sommes intéressés au problème de satisfaction de contraintes pondérées (WCSP). Ce cadre est une extension du formalisme CSP qui est présenté dans le chapitre 1 et est également un spécialisation du formalisme VCSP présenté au début du chapitre 2. Nous avons présenté différents algorithmes de filtrage connus aussi sous l'appellation cohérences souples : NC, AC, FAC, DAC, FDAC, EAC, EDAC, VAC, OSAC. Enfin, nous avons comparé ces cohérences en établissant deux types de liens qui peuvent exister entre elles. L'implication permet d'identifier les cohérences qui sont nécessairement vérifiées lorsque qu'une autre cohérence est établie. La $w_\emptyset$-supériorité permet d'identifier les cohérences qui, dans l'absolu, permettent d'obtenir de meilleures bornes $w_\emptyset$ que d'autres.

La seconde partie est constituée de trois chapitres. Au sein du chapitre 3, nous avons étudié la notion de la substituabilité de voisinage souple (SNS) dans le cadre WCSP. Nous avons présenté un opérateur efficace, appelée *pcost*, qui nous permet d'identifier les valeurs dites SNS-éliminables. Nous avons montré que dans le cas général, *pcost* est plus faible que SNS que nous avons démontré coNP-difficile. Nous montrons que sous certaines hypothèses, *pcost* est équivalent à SNS. Nous avons étudié également les liaisons entre SNS et certaines propriétés connues de cohérence d'arc souple. Enfin, nous avons développé un algorithme pour *pcost* qui bénéficie d'une complexité en temps raisonnable, et nous avons montré expérimentalement qu'il peut être associé à EDAC avec succès au cours de la recherche.

Au sein du chapitre 4, nous avons présenté un nouveau type de propriétés pour le cadre WCSP à savoir la cohérence de tuples notée TC. La propriété TC se base sur la notion de tuple. Elle consiste à maintenir un support pour chaque tuple sur l'ensemble des contraintes qui portent sur ce dernier. Pour établir TC sur un WCN donné, nous avons proposé une nouvelle opération (EPT) appelée `TupleProject` qui permet d'effectuer des transferts de coûts entre tuples. Deux versions d'algorithmes TC1_r et TC2_r ont été proposés pour établir la propriété TC. Nous avons proposé également une version "optimale" de cette propriété, OTC, qui peut être perçue comme une généralisation de OSAC (Optimal Soft Arc Consistency).

Au sein du chapitre 5, nous avons étendu la notion de substituabilité vue au chapitre 3 aux tuples. Nous nous somme concentrés sur la notion de substituabilité de tuples au voisinage (SNST) dans le cadre WCSP. Nous avons généralisé la propriété *pcost* pour permettre d'identifier les tuples souples-substituables au voisinage. Enfin, nous avons proposé un algorithme qui permet d'identifier et éliminer les tuples souples-substituables au voisinage.

Pour finir, nous rappelons sur la figure 5.2 le lien d'implication et de $w_\emptyset$-supériorité que nous avons établi tout au long de ce mémoire entre les différentes propriétés.

Perspectives de travaux futurs

Dans le chapitre 4, nous avons présenté une généralisation de la cohérence AC aux tuples. Le principe est d'établir un support pour chaque tuple d'une arité inférieure ou égale à un paramètre prédéfini r. En pratique, établir de tels supports est coûteux. Pour raffiner ce travail,

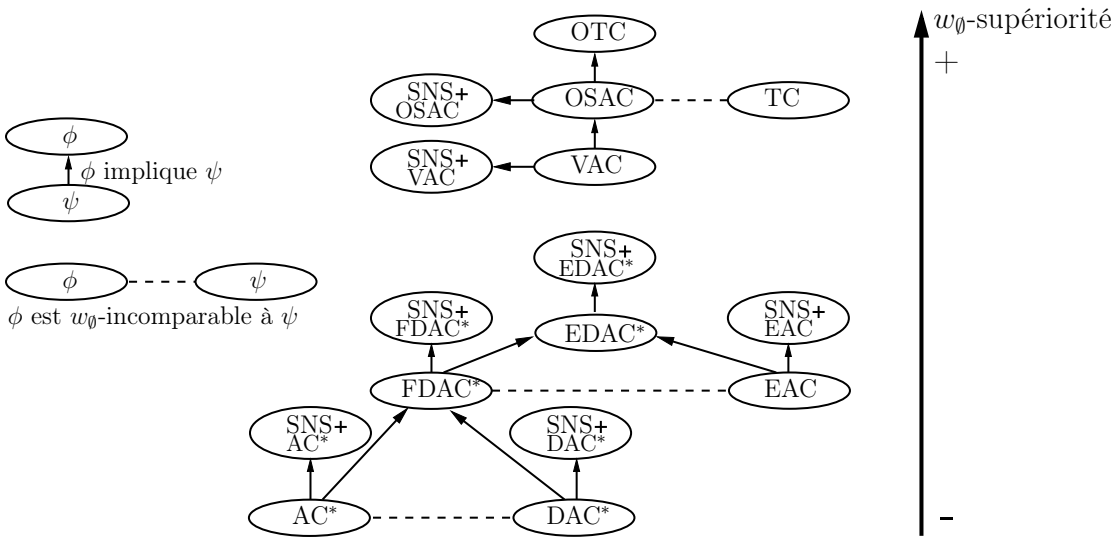


FIGURE 5.2 – Les relations entre les différents types de cohérence

nous envisageons d'établir un support uniquement pour un ensemble limité de tuples potentiellement intéressant. En d'autres termes, il faut établir un support pour chaque tuple qui par la suite permettra d'augmenter la borne inférieur $w_\emptyset$. Ce travail peut être également étendu en essayant de généraliser les autres cohérences tels que FDAC, EAC et EDAC aux tuples. La partie expérimentale a montré aussi que l'ajout systématique des contraintes ternaires qui portent sur un ensemble de variables qui forment une clique est très coûteux en terme de temps CPU. Il serait intéressant de trouver une heuristique qui permettra d'ajouter un nombre limité de contraintes ternaires susceptibles par la suite d'augmenter la borne inférieure $w_\emptyset$.

Dans le chapitre 5, nous avons présenté une généralisation de la notion de substituabilité aux tuples. Pour pouvoir faire une étude expérimentale, il serait intéressant de proposer une nouvelle forme de la substituabilité de tuples qui se limite à étudier la notion de substituabilité uniquement sur les tuples qui permettront par la suite de filtrer les domaines de variables.

Bibliographie

Aggoun, A. and Beldiceanu, N. (1992). Extending chip in order to solve complex scheduling and placement problems. In *JFPL*, pages 51–.

Bartak, R. and Erben, R. (2004). A new algorithm for singleton arc consistency. In *Proceedings of FLAIRS'04*, pages 257–262.

Bellicha, A., Capelle, C., Habib, M., Kokény, T., and Vilarem, M. (1994). CSP techniques using partial orders on domain values. In *Proceedings of ECAI'94 workshop on constraint satisfaction issues raised by practical applications*.

Benson, B. and Freuder, E. (1992). Interchangeability preprocessing can improve forward checking search. In *Proceedings of ECAI'92*, pages 28–30.

Bessiere, C. (1994). Arc consistency and arc consistency again. *Artificial Intelligence*, 65 :179–190.

Bessiere, C., Cardon, S., Debruyne, R., and Lecoutre, C. (2011). Efficient algorithms for singleton arc consistency. *Constraints*, 16(1) :25–53.

Bessiere, C. and Debruyne, R. (2004). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of CP'04 workshop on constraint propagation and implementation*, pages 17–27.

Bessiere, C. and Debruyne, R. (2005). Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI'05*, pages 54–59.

Bessiere, C., Freuder, E., and Régin, J. (1999). Using constraint metaknowledge to reduce arc consistency computation. *Artificial Intelligence*, 107 :125–148.

Bessiere, C. and Régin, J. (1996). MAC and combined heuristics : two reasons to forsake FC (and CBJ ?) on hard problems. In *Proceedings of CP'96*, pages 61–75.

Bessiere, C. and Régin, J. (2001). Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315.

Bessiere, C., Régin, J., Yap, R., and Zhang, Y. (2005). An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2) :165–185.

Bessiere, C. and van Hentenryck, P. (2003). To be or not to be ... a global constraint. In *Proceedings of CP'03*, pages 789–794.

Bistarelli, S., Faltings, B., and Neagu, N. (2002). Interchangeability in soft CSPs. In *Proceedings of CSCLP'02*, pages 31–46.

Bistarelli, S., Montanari, U., and Rossi, F. (1995). Constraint solving over semirings. In *Proceedings of IJCAI'95*, pages 624–630.

Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., and Fargier, H. (1999). Semiring-based CSPs and valued CSPs : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240.

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004a). Boosting systematic search by weighting constraints. In *Proceedings of ECAI'04*, pages 146–150.

Boussemart, F., Hemery, F., Lecoutre, C., and Sais, L. (2004b). Support inference for generic filtering. In *Proceedings of CP'04*, pages 721–725.

Cabon, B., de Givry, S., Lobjois, L., Schiex, T., and Warners, J. (1999). Radio Link Frequency Assignment. *Constraints*, 4(1) :79–89.

Chmeiss, A. and Jégou, P. (1996). Path-consistency : when space misses time. In *Proceedings of AAAI'96*, pages 196–201.

Chmeiss, A. and Jégou, P. (1998). Efficient path-consistency propagation. *International Journal on Artificial Intelligence Tools*, 7(2) :121–142.

Choueiry, B., Faltings, B., and Weigel, R. (1995). Abstraction by interchangeability in resource allocation. In *Proceedings of IJCAI'95*, pages 1694–1710.

Cooper, M. (1997). Fundamental properties of neighbourhood substitution in constraint satisfaction problems. *Artificial Intelligence*, 90 :1–24.

Cooper, M. (2003). Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3) :311–342.

Cooper, M., de Givry, S., Sanchez, M., Schiex, T., and Zytnicki, M. (2008). Virtual arc consistency for weighted CSP. In *Proceedings of AAAI'08*, pages 253–258.

Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., and Werner, T. (2010). Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8) :449–478.

Cooper, M., de Givry, S., and Schiex, T. (2007). Optimal soft arc consistency. In *Proceedings of IJCAI'07*, pages 68–73.

Cooper, M. and Schiex, T. (2004). Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2) :199–227.

Cooper, M. C. (2005). High-order consistency in valued constraint satisfaction. *Constraints*, 10(3) :283–305.

Dahiyat, B. I. and Mayo, S. L. (1996). Protein design automation. *Protein Science*, 5(5) :895–903.

de Givry, S. (2004). Singleton consistency and dominance for weighted CSP. In *Proceedings of CP'04 Workshop on Preferences and Soft Constraints*.

de Givry, S., Heras, F., Zytnicki, M., and Larrosa, J. (2005). Existential arc consistency : Getting closer to full arc consistency in weighted CSPs. In *Proceedings of IJCAI'05*, pages 84–89.

de Givry, S., Prestwich, S. D., and O'Sullivan, B. (2013). Dead-end elimination for weighted csp. In *CP*, pages 263–272.

Debruyne, R. and Bessiere, C. (1997). Some practical filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI'97*, pages 412–417.

Dechter, R. and Pearl, J. (1988). Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34(1) :1–38.

Desmet, J., Maeyer, M. D., Hazes, B., and Lasters, I. (1992). The dead-end elimination theorem and its use in protein side-chain positioning. *Nature*, 356(6369) :539–542.

Dubois, D., Fargier, H., and Prade, H. (1993). The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In *Proc. of the 2nd IEEE Inter. Conf. on Fuzzy Systems (FUZZ-IEEE'93), San Francisco, CA, 28/03/93-01/04/93*.

Fargier, H., Dubois, D., and Prade, H. (1995). Problèmes de satisfaction de contraintes flexibles : Une approche égalitariste. *Revue d'Intelligence Artificielle*, 9(3) :311–354.

Fargier, H. and Lang, J. (1993). Uncertainty in constraint satisfaction problems : a probalistic approach. In *ECSQARU*, pages 97–104.

Favier, A., de Givry, S., Legarra, A., and Schiex, T. (2011). Pairwise decomposition for combinatorial optimization in graphical models. In *IJCAI*, pages 2126–2132.

Freuder, E. (1982). A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1) :24–32.

Freuder, E. (1991). Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233.

Freuder, E. and Sabin, D. (1997). Interchangeability supports abstraction and reformulation for multi-dimensional constraint satisfaction. In *Proceedings of AAAI'97*, pages 191–196.

Freuder, E. and Wallace, R. (1992). Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3) :21–70.

Frost, D. and Dechter, R. (1994). In search of the best constraint satisfaction search. In *AAAI*, pages 301–306.

Gaschnig, J. (1979). Performance measurement and analysis of search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon.

Georgiev, I., Lilien, R. H., and Donald, B. R. (2006a). Improved pruning algorithms and divide-and-conquer strategies for dead-end elimination, with application to protein design. In *ISMB (Supplement of Bioinformatics)*, pages 174–183.

Georgiev, I., Lilien, R. H., and Donald, B. R. (2006b). Improved pruning algorithms and divide-and-conquer strategies for dead-end elimination, with application to protein design. In *ISMB (Supplement of Bioinformatics)*, pages 174–183.

Ginsberg, M. (1993). Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1 :25–46.

Goldstein, R. F. (1994). Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophys J*, 66(5) :1335–1340.

Han, C. and Lee, C. (1988). Comments on Mohr and Henderson's path consistency. *Artificial Intelligence*, 36 :125–130.

Haralick, R. and Elliott, G. (1980). Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313.

Haselbock, A. (1993). Exploiting interchangeabilities in constraint satisfaction problems. In *Proceedings of IJCAI'93*, pages 282–287.

Janssen, P., Jegou, P., Nouguier, B., and Vilarem, M. C. (1989). A filtering process for general constraint-satisfaction problems : achieving pairwise-consistency using an associated binary representation. In *Tools for Artificial Intelligence, 1989. Architectures, Languages and Algorithms, IEEE International Workshop on*, pages 420–427.

Karakashian, S., Woodward, R., Choueiry, B., Prestwhich, S., and Freuder, E. (2010). A partial taxonomy of substitutability and interchangeability. Technical Report arXiv :1010.4609, CoRR.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4) :373–396.

Koster, A. (1999). *Frequency assignment : Models and Algorithms*. PhD thesis, University of Maastricht, The Netherlands.

Lal, A., Choueiry, B., and Freuder, E. (2005). Neighborhood interchangeability and dynamic bundling for non-binary finite CSPs. In *Proceedings of AAAI'05*, pages 397–404.

Larrosa, J. (2002). Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53.

Larrosa, J. and Schiex, T. (2003). In the quest of the best form of local consistency for Weighted CSP. In *Proceedings of IJCAI'03*, pages 363–376.

Larrosa, J. and Schiex, T. (2004). Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2) :1–26.

Lawler, E. L. and Wood, D. E. (1966). Branch-and-Bound Methods : A Survey. *Operations Research*, 14 :699–719.

Lecoutre, C. and Cardon, S. (2005). A greedy approach to establish singleton arc consistency. In *Proceedings of IJCAI'05*, pages 199–204.

Lecoutre, C., Cardon, S., and Vion, J. (2007). Path consistency by dual consistency. In *Proceedings of CP'07*, pages 438–452.

Lecoutre, C. and Hemery, F. (2007). A study of residual supports in arc consistency. In *Proceedings of IJCAI'07*, pages 125–130.

Lecoutre, C. and Vion, J. (2008). Enforcing arc consistency using bitwise operations. *Constraint Programming Letters*, 2 :21–35.

Lee, J. and Leung, K. (2010). A stronger consistency for soft global constraints in weighted constraint satisfaction. In *Proceedings of AAAI'10*, pages 121–127.

Looger, L. L. and Hellinga, H. W. (2001). Generalized dead-end elimination algorithms make large-scale protein side-chain structure prediction tractable : implications for protein design and structural genomics. *Journal of molecular biology*, 307(1) :429–445.

Mackworth, A. (1977). Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118.

Martello, S. and Toth, P. (1990). *Knapsack Problems : Algorithms and Computer Implementations*. Wiley.

Minton, S., Johnston, M., Philips, A., and Laird, P. (1992). Minimizing conflicts : a heuristic repair method for constraint-satisfaction and scheduling problems. *Artificial Intelligence*, 58(1-3) :161–205.

Mohr, R. and Henderson, T. (1986). Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233.

Montanari, U. (1974). Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7 :95–132.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems : Networks of Plausible Inference*. Morgan Kaufmann.

Petcu, A. and Faltings, B. (2003). Applying interchangeability techniques to the distributed breakout algorithm. In *Proceedings of CP'03*, pages 925–929.

Pierce, N. A., Spriet, J. A., Desmet, J., and Mayo, S. L. (2000). Conformational splitting : A more powerful criterion for dead-end elimination. *Journal of Computational Chemistry*, 21(11) :999–1009.

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problems. *Computational Intelligence*, 9(3) :268–299.

Refalo, P. (2004). Impact-based search strategies for constraint programming. In *Proceedings of CP'04*, pages 557–571.

Régin, J. (1994). A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI'94*, pages 362–367.

Régin, J. (2003). *Constraints and Integer Programming combined*, chapter Global constraints and filtering algorithms. Kluwer.

Rossi, F., Petrie, C., and Dhar, V. (1990). On the equivalence of constraint satisfaction problems. In *Proceedings of ECAI'90*, pages 550–556.

Rossi, F., van Beek, P., and Walsh, T., editors (2006). *Handbook of Constraint Programming*. Elsevier.

Sandholm, T. (2002). Algorithm for optimal winner determination in combinatorial auctions. *Artif. Intell.*, 135(1-2) :1–54.

Schiex, T. (1992a). Possibilistic constraint satisfaction problems or "how to handle soft constraints ?". In *UAI*, pages 268–275.

Schiex, T. (1992b). Possibilistic constraint satisfaction problems or "how to handle soft constraints ?". In *In Proc. 8th Conf. of Uncertainty in AI*, pages 269–275.

Schiex, T. (2000). Arc consistency for soft constraints. In *CP*, pages 411–424.

Schiex, T., Fargier, H., and Verfaillie, G. (1995). Valued constraint satisfaction problems : Hard and easy problems. In *Proceedings of IJCAI'95*, pages 631–639.

Shapiro, L. G. and Haralick, R. M. (1981). Structural descriptions and inexact matching. *IEEE Trans. Pattern Anal. Mach. Intell.*, 3(5) :504–519.

Singh, M. (1996). Path consistency revisited. *International Journal on Artificial Intelligence Tools*, 5 :127–141.

Stergiou, K. and Walsh, T. (1999). Encodings of non-binary constraint satisfaction problems. In *Proceedings of AAAI'99*, pages 163–168.

van Hentenryck, P., Deville, Y., and Teng, C. (1992). A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57 :291–321.

La substituabilité et la cohérence de tuples pour les réseaux de contraintes pondérées

Résumé : Cette thèse se situe dans le domaine de la programmation par contraintes (CP). Plus précisément, nous nous intéressons au problème de satisfaction de contraintes pondérées (WCSP), qui est un problème d'optimisation pour lequel plusieurs formes de cohérences locales souples telles que, par exemple, la cohérence d'arc existentielle directionnelle (EDAC*) et la cohérence d'arc virtuelle (VAC) ont été proposées durant ces dernières années. Dans ce cadre, nous adoptons une perspective différente en revisitant la propriété bien connue de la substituabilité (souple). Tout d'abord, nous précisons les relations existant entre la substituabilité de voisinage souple (SNS) et une propriété appelée pcost qui est basée sur le concept de surcoût de valeurs (par le biais de l'utilisation de paires de surcoût). Nous montrons que sous certaines hypothèses, pcost est équivalent à SNS, mais que dans le cas général, elle est plus faible que SNS prouvée être coNP-difficile. Ensuite, nous montrons que SNS conserve la propriété VAC, mais pas la propriété EDAC. Enfin, nous introduisons un algorithme optimisé et nous montrons sur diverses séries d'instances WCSP l'intérêt pratique du maintien de pcost avec AC*, FDAC* ou EDAC*, au cours de la recherche. Nous introduisons un algorithme optimisé et nous étudions la relation existante entre SNS et les différentes cohérences. Nous présentons aussi un nouveau type de propriétés pour les WCSPs. Il s'agit de la cohérence de tuples (TC) dont l'établissement sur un WCN est effectué grâce à une nouvelle opération appelée TupleProject. Nous proposons également une version optimale de cette propriété, OTC, qui peut être perçue comme une généralisation de OSAC (Optimal Soft Arc Consistency). Enfin, nous étendons la notion de substituabilité souple aux tuples.

Mots clés : Programmation par contraintes,WCSP, contraintes souples, substituabilité souple, cohérences souples.

The substitutability and the tuples consistency for weighted constraint networks

Abstract : This thesis is in the field of constraint programming (CP). More precisely, we focus on the weighted constraint satisfaction problem (WCSP), which is an optimization problem for which many forms of soft local (arc) consistencies have been proposed such as, for example, existential directional arc consistency (EDAC) and virtual arc consistency (VAC) in recent years. In this context, we adopt a different perspective by revisiting the well-known property of (soft) substitutability. First, we provide a clear picture of the relationships existing between soft neighborhood substitutability (SNS) and a tractable property called *pcost* which allows us to compare the cost of two values (through the use of so-called cost pairs). We prove that under certain assumptions, *pcost* is equivalent to SNS but weaker than SNS in the general case since we show that SNS is coNP-hard. We also show that SNS preserves the property VAC but not the property EDAC. Finally, we introduce an optimized algorithm and we show on various series of WCSP instances, the practical interest of maintaining *pcost* together with AC*, FDAC* or EDAC*, during search. We also present a new type of properties for WCSPs called tuples consistency (TC). Enforcing TC is done through a new operation called TupleProject. Moreover, we propose an optimal version of this property, OTC, which can be seen as a generalization of OSAC (Optimal Soft Arc Consistency). Finally, we extend soft substitutability concept to tuples.

Keywords : Constraint programming, WCSP, soft constraints, soft substitutability, soft consistencies.