Une approche déductive de la résolution de problèmes de satisfaction de contraintes

Carlos Castro

Département de formation doctorale en informatique
UFR STMIA

École doctorale IAE + M

Une approche déductive de la résolution de problèmes de satisfaction de contraintes

THÈSE

présentée et soutenue publiquement le 18 décembre 1998

pour l'obtention du

Doctorat de l'université Henri Poincaré – Nancy 1

(spécialité informatique)

par

Carlos Castro

Composition du jury

Président :	**François Fages**, Chargé de recherche CNRS, ENS Paris, France
Rapporteurs :	**Krzysztof R. Apt**, Prof. à l'Université d'Amsterdam et CWI, Pays Bas
	Jacques Cohen, Prof. à l'Université Brandeis, États-Unis
	Marie-Claude Portmann, Prof. à l'École des Mines de Nancy, France
Examinateurs :	**Alexander Bockmayr**, Prof. à l'Université Henri Poincaré, Nancy 1, France
	Claude Kirchner, Directeur de recherche à l'INRIA, France

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Remerciements

Je voudrais tout d'abord remercier chaleureusement Claude Kirchner pour la possibilité qu'il m'a donnée de travailler dans l'équipe PROTHEO, pour m'avoir fait connaître le monde de l'informatique théorique et pour avoir dirigé ma recherche toujours avec intérêt. Ses commentaires et conseils, représentant toute sa culture informatique, m'ont guidé tout au long de ma thèse et sans doute m'ont permis de la finir. Comme responsable scientifique de l'équipe, Claude m'a aussi donné tous les moyens nécessaires pour réaliser ma recherche.

Je tiens ensuite à remercier les membres du jury :

François Fages qui, à travers son cours à l'école d'été d'Aix-en-Provence en août 1997 et son livre sur la programmation logique par contraintes, m'a beaucoup appris et m'a fait l'honneur de présider le jury ;

Krzysztof R. Apt qui s'est intéressé à mon travail depuis longtemps et qui a fait un grand effort pour venir à Nancy ;

Jacques Cohen qui n'a pas hésité à participer au jury en acceptant la lourde tâche de rapporteur et qui a aussi fait l'effort de venir depuis les États-Unis ;

Marie-Claude Portmann qui a lu en détail le document et a fait des commentaires très pertinents, sans avoir la satisfaction d'assister à la soutenance ;

Alexander Bockmayr qui a bien voulu s'intéresser à ce travail et dont les travaux d'intégration entre recherche opérationnelle et résolution de contraintes m'ont été très utiles.

Je remercie également les membres de l'équipe PROTHEO :

Hélène Kirchner qui a lu mes articles et qui après avoir repris la direction de l'équipe m'a donné tous les moyens pour poursuivre ma recherche ;

Christophe Ringeissen avec qui j'ai discuté dans des innombrables occasions au sujet de cette thèse. Il m'a aidé à la formalisation présentée dans ce travail, il a lu mes travaux et il a rendu le français de ce document presque lisible. Je lui en suis vivement reconnaissant ;

Eric Domenjoud avec qui j'ai travaillé pour le projet industriel GIHP-INRIA. Il a aussi lu mes travaux, m'a beaucoup aidé à éclaircir des concepts fondamentaux en informatique et m'a expliqué des nuances du français ;

Pierre-Ettienne Moreau et Peter Borovanský pour leur aide concernant le langage ELAN ce qui m'a permis de réaliser l'implantation du système COLETTE ;

Göknil Ersoy et Hubert Dubois qui en utilisant le système COLETTE m'ont permis de l'améliorer ;

Jianyang Zhou pour partager avec moi son expérience dans la résolution de problèmes combinatoires.

Un grand merci à Eric Monfroy avec qui j'ai travaillé pour la définition d'un langage de stratégies. Ses commentaires concernant mon travail ont aussi été très utiles.

Je remercie Philippe Dosch pour son aide concernant le logiciel ILOG Views ce qui m'a permis de développer l'interface graphique du système COLETTE.

Je profite de l'occasion pour remercier Hubert Hoffmann pour m'avoir initié à la problématique des problèmes de satisfaction de contraintes.

J'aurai le temps dans le futur pour remercier mcm pour son soutien, sa compréhension, la lecture de mes papiers et son aide pour finir ce document et préparer ma soutenance.

Résumé

Dans cette thèse, nous modélisons la résolution de problèmes de satisfaction de contraintes (CSPs) comme un processus de déduction. Nous formalisons les techniques de résolution de CSPs sur des domaines finis en utilisant des règles de réécriture et des stratégies. Les règles expriment les transformations réalisées par ces techniques sur les contraintes et les stratégies contrôlent l'application de ces règles. Cette approche nous permet de vérifier facilement des propriétés telles que la correction, la complétude et la terminaison d'un solveur. L'utilisation d'un langage de stratégies puissant nous permet de décrire les heuristiques d'une manière très abstraite et flexible. En utilisant cette approche déductive, nous pouvons prouver l'existence ou l'inexistence d'une solution : il nous suffit de regarder le terme de preuve pour reconstruire exactement une preuve du calcul réalisé. Afin de valider notre approche, nous avons implanté le système COLETTE dans le langage ELAN.

Nous étudions tout d'abord le problème de la résolution d'un ensemble de contraintes élémentaires qui sont combinées par des opérateurs de conjonction. Nous nous intéressons ensuite à la résolution de problèmes d'optimisation en utilisant des techniques de résolution de CSPs. Nous abordons l'extension du langage de contraintes afin de considérer la combinaison des contraintes élémentaires avec les connecteurs logiques de disjonction, d'implication et d'équivalence. Le traitements des contraintes disjonctives est décrit en détail. Finalement, nous étudions la coopération de solveurs. En utilisant les opérateurs de stratégies concurrents du langage ELAN, nous décrivons d'une manière abstraite des schémas de coopération séquentiels et concurrents. Pour d'autres types de coopération, nous utilisons des primitives de bas niveau qui sont disponibles dans ELAN pour la communication entre processus.

Mots-clés: Problème de satisfaction de contraintes, Règle de réécriture, Stratégie, Système de calcul

Abstract

In this thesis, we model the resolution of constraint satisfaction problems (CSPs) as a deductive process. We formalise CSP solving techniques over finite domains using rewrite rules and strategies. The rules express the transformations over the set of constraints carried out by these techniques. The strategies control the application of these rules. This approach allows us to easily verify properties like correctness, completeness, and termination of a solver. The use of a powerful strategy language allows us to describe heuristics in an abstract and flexible way. Furthermore, using this deductive approach we can justify the existence or absence of a solution: we just have to look at the proof term in order to reconstruct exactly a proof of the computation carried out. To validate our approach, we have implemented the system COLETTE using the ELAN language.

Firstly, we study the resolution of a set of elementary constraints combined by conjunction operators. Next, we focus on the resolution of optimisation problems using CSP solving techniques. Then, we extend the constraint language in order to consider the combination of elementary constraints by the logical operators of disjunction, implication, and equivalence. The treatment of disjunctive constraints is described in detail. Finally, we study the cooperation of solvers. Using the concurrent strategy operators of the ELAN language we describe in an abstract way some schemes of sequential and concurrent cooperation. For other types of cooperation, we use low-level primitives for process communication available in ELAN.

Keywords: Constraint Satisfaction Problem, Rewrite Rule, Strategy, Computational System

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Sommaire

Le problème de satisfaction de contraintes (CSP [1]) peut être défini très simplement comme le problème d'affectation d'un ensemble de variables à partir d'un ensemble de valeurs possibles, de façon telle qu'un ensemble de contraintes portant sur ces variables soit satisfait. Dans le sens le plus large du terme, ce problème se trouve à l'origine de nombreux problèmes mathématiques. Ses applications incluent des domaines divers tels que la reconnaissance de formes et l'ordonnancement d'activités en milieu industriel. Selon le domaine des objets considérés, diverses approches ont été proposées par plusieurs communautés scientifiques. Le problème de l'existence de solutions à un CSP est NP-complet. Afin de faire face à cette complexité, des techniques qui permettent de réduire l'espace de recherche ont été proposées. En particulier, depuis les années soixante-dix, il y a eu un développement important des techniques de résolution de contraintes sur des variables portant sur de domaines finis et leurs applications dans la résolution de problèmes réels a marqué une des contributions importantes de l'informatique dans la résolution de problèmes pratiques. Les travaux réalisés par la communauté d'intelligence artificielle (IA) ont été centrés principalement sur l'amélioration de techniques de résolution de CSP. Dans la communauté de programmation logique (LP [2]) on trouve des travaux d'intégration des techniques de résolution de contraintes dans le cadre de langages à la Prolog qui a été à l'origine du cadre de programmation logique avec contraintes (CLP [3]).

La motivation principale de ce travail est de considérer la résolution de CSPs comme un processus d'inférence. L'objectif principal de cette thèse est d'étudier les techniques de résolution de CSPs du point de vue des systèmes de calcul afin d'avoir un cadre formel général pour les

1. Constraint Satisfaction Problem.
2. Logic Programming.
3. Constraint Logic Programming.

exprimer d'une façon claire et précise et ainsi prouver leurs propriétés fondamentales, mais aussi les améliorer par l'introduction de stratégies spécifiques.

1.1 Applications

La littérature classique sur les CSP les définit comme un triplet $\langle X, D, C \rangle$, où X est un ensemble fini de variables x_i, D est l'ensemble des domaines D_{x_i} chacun contenant les valeurs prises par la variable x_i, et C est un ensemble de contraintes c_i qui contraint les valeurs prises par les variables. Le problème de satisfaction de contraintes consiste à déterminer une ou toutes les combinaisons de valeurs qui satisfont toutes les contraintes. Afin d'éclaircir le type de problèmes que sont traités dans le cadre des CSPs, on présente quelques exemples de problèmes qui sont considérés typiquement comme des problèmes de satisfaction de contraintes.

Les techniques de recherche opérationnelle (RO) ont atteint un très haut niveau de développement, leur utilisation a été largement diffusée avec le développement d'ordinateurs de plus en plus puissants et la popularisation des ordinateurs personnels. Malgré cela, un grand nombre de problèmes combinatoires ne peuvent toujours pas être résolus efficacement en raison de leur très haute complexité. Les problèmes présentés ici sont des exemples d'applications normalement traitées avec des techniques de RO et qui ont été résolues efficacement en utilisant des techniques de résolution de contraintes.

1.1.1 Problème de coloriage de graphes

Un problème utilisé très fréquemment dans la littérature pour expliquer les concepts et les algorithmes de résolution de CSPs est le *problème de coloriage de graphes.* Ce problème joue aussi un rôle important en RO car de nombreux problèmes d'emploi du temps ou d'ordonnancement peuvent être exprimés comme des problèmes de coloriage de graphes.

Étant donné un graphe et un certain nombre de couleurs, il s'agit d'affecter une couleur à chaque nœud tel que deux nœuds adjacents n'aient pas la même couleur. Une instance de ce problème est le *problème de coloriage de cartes.* Dans ce cas, le problème consiste à affecter une couleur à chaque zone d'une carte en utilisant un nombre limité de couleurs et en respectant la contrainte que deux zones adjacentes n'aient pas la même couleur. La figure 1.1 présente un exemple de ce type de problème [107].

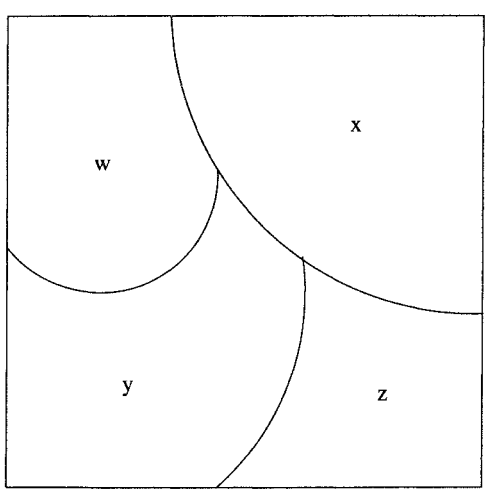

FIG. 1.1 – *Problème de coloriage de cartes*

Comme il s'agit d'une instance particulière du problème de coloriage de graphes, il peut être représenté par le graphe de la figure 1.2, où chaque nœud représente une zone de la carte et chaque arc indique que les deux nœuds qu'il connecte représentent deux zones adjacents.

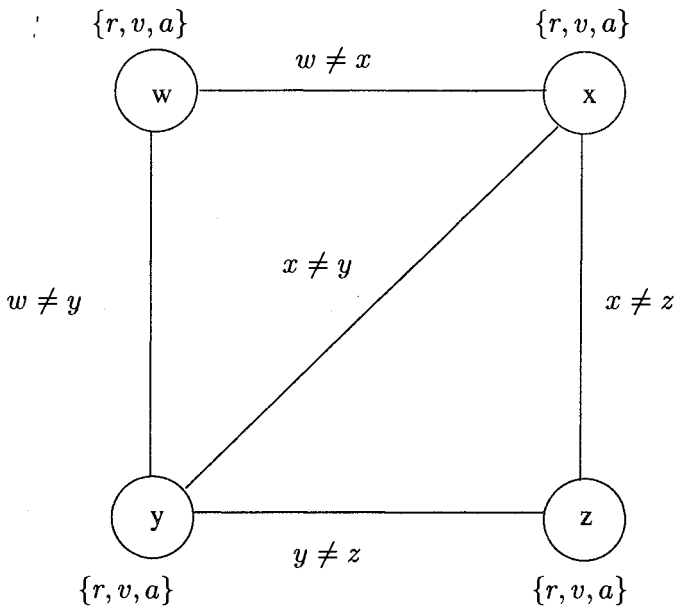

FIG. 1.2 – *Représentation par des graphes du problème de coloriage de cartes*

Si on suppose que chaque zone peut être colorée avec les couleurs rouge (r), vert (v), ou azur (a), sa formulation comme un CSP est la suivante

$$\begin{aligned} X &= \{w, x, y, z\} \\ D_w = D_x = D_y = D_z &= \{r, v, a\} \\ C &= \{w \neq x,\ w \neq y,\ x \neq y,\ x \neq z,\ y \neq z\} \end{aligned}$$

Du au fait que dans chaque contrainte intervient au plus deux variables et que le domaine associé à chaque variable est un ensemble fini de valeurs, ce problème est dit *CSP binaire sur des domaines finis*, une classe particulière de CSP très largement étudiée.

Une extension du problème de coloriage de graphes consiste à déterminer le nombre minimum de couleurs que l'on nécessite pour colorer tous les nœuds en respectant les contraintes (le nombre chromatique du graphe). Ce type d'extension du CSP est appelée problème d'optimisation de satisfaction de contraintes.

1.1.2 Ordonnancement d'activités

Étant donnés un ensemble d'activités et un ensemble de relations de précédence entre ces activités, où chaque activité i est caractérisée par sa durée d_i, une contrainte de précédence entre les activités i et j impose que l'activité i commence après la fin de l'activité j. Le problème consiste à déterminer le temps minimum dans lequel toutes les activités peuvent être terminées. Ce problème est connu par la communauté RO comme le *problème du chemin critique* et il est résolu généralement en utilisant une technique spécifique : PERT[4] [57].

4. Program Evaluation and Review Technique.

Pour expliquer comment ce problème peut être résolu en utilisant des techniques de résolution de contraintes on considère l'exemple du tableau 1.1 [40], où on présente les activités, leur durée et les relations de précédence entre les activités.

Activité	Durée	Précédence
A	7	-
B	3	A
C	1	B
D	8	A
E	2	D,C
F	1	D,C
G	1	D,C
H	3	F
J	2	H
K	1	E,G,J

TAB. 1.1 – *Problème d'ordonnancement d'activités*

Ce type de problème est traditionnellement représenté par un graphe orienté comme celui de la figure 1.3, où chaque arc représente une activité et les nœuds expriment les relations de précédence entre les activités.

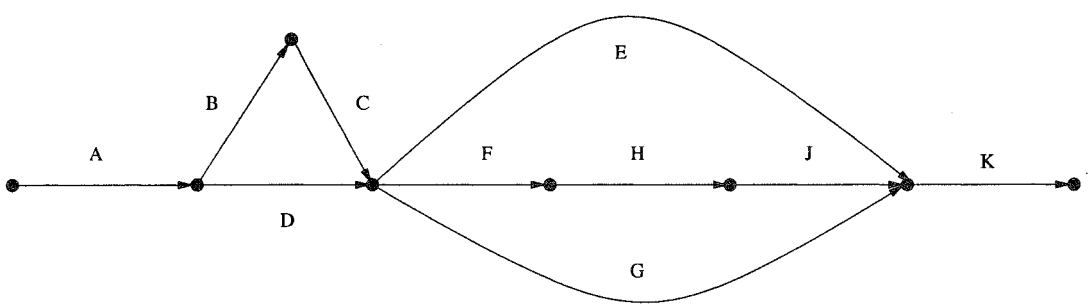

FIG. 1.3 – *Représentation par des graphes du problème d'ordonnancement d'activités*

Si on associe à chaque activité i une variable S_i qui représente le temps de début de l'activité, une contrainte de précédence entre les activités i et j peut être exprimée par l'inégalité $S_i \geq S_j + d_j$. Pour représenter la fin de la dernière activité on ajoute une nouvelle variable S_{fin}. Ainsi, l'ensemble de contraintes de ce CSP est le suivant

$$
\begin{aligned}
S_B &\geq S_A + 7 \\
S_D &\geq S_A + 7 \\
S_C &\geq S_B + 3 \\
S_E &\geq S_C + 1 \\
S_E &\geq S_D + 8 \\
S_G &\geq S_C + 1 \\
S_G &\geq S_D + 8 \\
S_F &\geq S_D + 8
\end{aligned}
$$

$$\begin{aligned} S_F &\geq S_C + 1 \\ S_H &\geq S_F + 1 \\ S_J &\geq S_H + 3 \\ S_K &\geq S_G + 1 \\ S_K &\geq S_E + 2 \\ S_K &\geq S_J + 2 \\ S_{fin} &\geq S_K + 1 \end{aligned}$$

La résolution de ce problème par des techniques de CSP peut être vue comme un processus en trois étapes

- une propagation de contraintes initiale ;
- l'affectation de la valeur la plus petite possible à la variable S_{fin} ;
- une deuxième étape de propagation de contraintes.

Comme il est nécessaire de définir un domaine pour chaque variable et la somme de toutes les durées est 29, on peut définir, par exemple, un domaine $D_{S_i} = [0, \ldots, 30]$ pour chaque variable S_i.

Ce problème est un autre exemple d'un CSP binaire sur des domaines finis. Néanmoins, du fait que les domaines sont ordonnés, on verra que les contraintes peuvent être analysées en se basant seulement sur les bornes des domaines des variables.

La propagation de contraintes initiale élimine de chaque domaine les valeurs nécessaires pour que chaque contrainte soit satisfaite. Ainsi, si on analyse la contrainte

$$S_B \geq S_A + 7$$

on peut réaliser que la borne inférieure de la variable S_B doit être supérieure ou égale à la somme de la borne inférieure de S_A et 7 et la borne supérieure de S_A doit être inférieure ou égale à la somme de la borne supérieure de S_B et -7. De cette façon, on obtient les domaines suivants

$$D_{S_A} = [0, \ldots, 23] \quad \text{et} \quad D_{S_B} = [7, \ldots, 30]$$

À partir de $S_D \geq S_A + 7$, et en faisant le même type d'analyse, on obtient $D_{S_D} = [7, \ldots, 30]$ (le domaine de S_A ne change pas). Si on continue ce processus jusqu'à qu'aucun domaine puisse être modifié, on obtient

$$\begin{aligned} S_A &\in [0, \ldots, 8] \\ S_B &\in [7, \ldots, 19] \\ S_C &\in [10, \ldots, 22] \\ S_D &\in [7, \ldots, 15] \\ S_E &\in [15, \ldots, 27] \\ S_F &\in [15, \ldots, 23] \\ S_G &\in [15, \ldots, 28] \\ S_H &\in [16, \ldots, 24] \\ S_J &\in [19, \ldots, 27] \\ S_K &\in [21, \ldots, 29] \\ S_{fin} &\in [22, \ldots, 30] \end{aligned}$$

La deuxième étape affecte la variable S_{fin} par la valeur la plus petite dans son domaine, 22. Finalement, après cette affectation, la deuxième propagation de contraintes est réalisée de la même manière que la première, et on obtient

$$\begin{aligned} S_A &= 0 \\ S_B &\in [7,\ldots,11] \\ S_C &\in [10,\ldots,14] \\ S_D &= 7 \\ S_E &\in [15,\ldots,19] \\ S_F &= 15 \\ S_G &\in [15,\ldots,20] \\ S_H &= 16 \\ S_J &= 19 \\ S_K &= 21 \\ S_{fin} &= 22 \end{aligned}$$

En termes d'un problème de chemin critique, les variables correspondant aux *activités critiques* (les activités A, D, F, H, J et K) ont été instanciées à sa seule valeur possible, le reste correspond aux activités avec une certaine *ampleur*. Il est intéressant de noter que même si la technique PERT résout d'une façon efficace ce type de problèmes, les techniques de résolution de CSP peuvent aussi le résoudre efficacement mais en utilisant une approche plus générale.

Nous voyons ici l'application de deux concepts fondamentaux des techniques de résolution de CSPs : la propagation de contraintes et l'énumération de valeurs. Dûe à la structure particulière du problème, un certain ordre dans l'application de ces deux concepts, une tactique, nous a donné le résultat attendu.

1.1.3 Ordonnancement disjonctif

Le problème d'ordonnancement disjonctif peut être considéré comme une extension du problème d'ordonnancement d'activités. Les caractéristiques de ce problème sont les suivantes

- il existe un ensemble de *jobs* job_i qui doivent être terminés en un temps plus tardif donné ;
- chaque *job* est composé d'un ensemble de tâches $tâche_{ij}$ qui sont réalisées en un temps d_{ij} par une machine $machine_j$;
- il existe un ensemble de relations de précédence entre les tâches d'un même *job* ;
- chaque machine ne peut réaliser qu'une tâche à la fois.

Le but consiste à déterminer le temps minimum dans lequel tous les *jobs* peuvent être terminés en respectant les contraintes.

Dans le tableau 1.2, on présente les données d'un problème d'ordonnancement disjonctif consistant en 2 jobs et 3 machines.

Job	Machine			Temps
	1	2	3	au plus tard
1	94	66	10	225
2	53	26	15	225

TAB. 1.2 – *Problème d'ordonnancement disjonctif*

Si on représente par une variable $tâche_{ij}$ le temps de début de la tâche du *job* i sur la machine j, ce problème peut être modélisé par l'ensemble de contraintes suivant.

- Les valeurs possibles pour chaque variable

$$\forall i \in \{1,2\}\ \forall j \in \{1,2,3\} : \quad tâche_{ij} \in [0,\ldots,225]$$

- Le temps maximum permis pour la fin de chaque job

$$\begin{aligned} tâche_{13} + 10 &\leq 225 \\ tâche_{23} + 15 &\leq 225 \end{aligned}$$

- Les contraintes de précédence entre les tâches d'un même job

$$\begin{aligned} tâche_{11} + 94 &\leq tâche_{12} \\ tâche_{12} + 66 &\leq tâche_{13} \\ tâche_{21} + 53 &\leq tâche_{22} \\ tâche_{22} + 26 &\leq tâche_{23} \end{aligned}$$

- Les contraintes de capacité des machines

$$\begin{aligned} tâche_{11} \geq tâche_{21} + 53 \quad \vee \quad tâche_{21} \geq tâche_{11} + 94 \\ tâche_{12} \geq tâche_{22} + 26 \quad \vee \quad tâche_{22} \geq tâche_{12} + 66 \\ tâche_{13} \geq tâche_{23} + 15 \quad \vee \quad tâche_{23} \geq tâche_{13} + 10 \end{aligned}$$

La différence principale de ce problème par rapport au problème d'ordonnancement d'activités consiste en le partage des machines par les tâches. Cette situation est représentée par les contraintes disjonctives. La précédence entre les tâches qui utilisent une même machine n'est pas connue à l'avance, elle ne sera connue qu'à partir des solutions du problème.

Si on considère seulement les contraintes de précédence entre les tâches d'un même job, les temps possibles d'exécution de chaque tâche sont représentés dans la figure 1.4, où pour chaque tâche on montre sa durée, colorée en gris, et son temps de début le plus tôt et son temps de fin le plus tard.

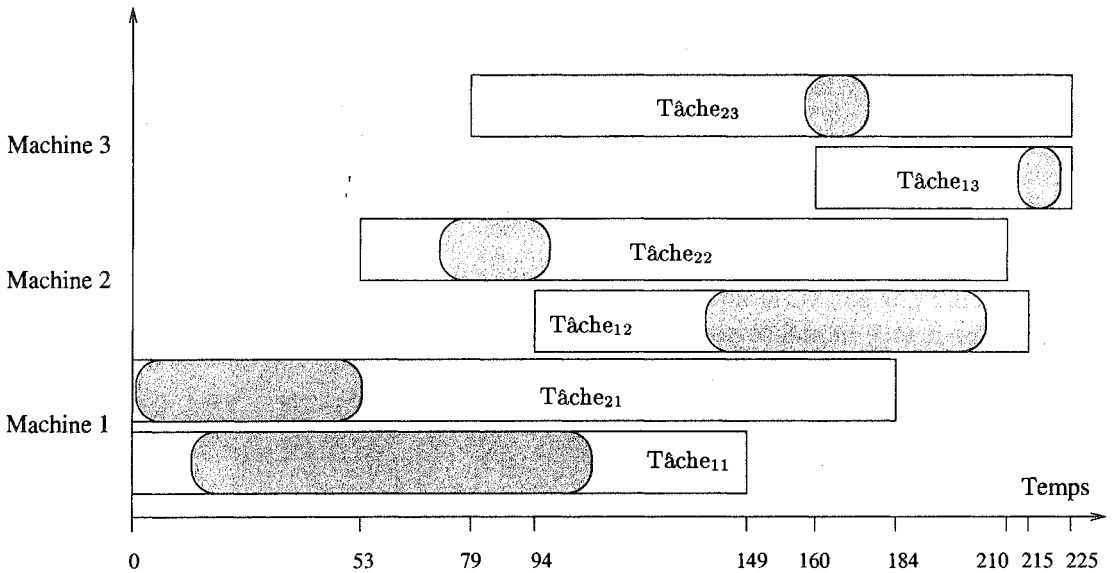

FIG. 1.4 – *Ordonnancements possibles des tâches après l'analyse des contraintes de précédence*

Mais cette analyse n'est pas suffisante pour satisfaire toutes les contraintes, le partage de ressources rend ce problème encore plus difficile. Les techniques pour le traitement des contraintes disjonctives, développées par la communauté CLP, permettent de résoudre ce type de situations et rivalisent avec les meilleurs algorithmes de RO pour les problèmes d'ordonnancement disjonctif [17]. Un bon résumé de l'utilisation de l'approche contraintes dans la résolution de problèmes d'ordonnancement se trouve dans [115].

1.1.4 Puzzles

En raison de leur nature combinatoire et de leur large diffusion, les puzzles sont utilisés fréquemment comme des exemples pour tester l'efficacité des algorithmes de résolution de CSPs. Bien qu'il ne s'agisse pas directement de problèmes industriels, la problématique qu'implique leur résolution systématique est commune à tous les problèmes combinatoires. Largement connus sont le *problème du zèbre* [36], le problème crypto-arithmétique *SEND+MORE=MONEY* [109] et le *problème des N reines*. Ce dernier exemple est présenté ci-dessous.

Le problème des N reines consiste à placer N reines sur un échiquier de manière que, en considérant les règles du jeux d'échec, elles ne puissent pas s'attaquer. Celui-ci est un CSP où interviennent seulement de contraintes unaires et binaires et c'est un exemple excellent pour montrer les diverses façons avec lesquelles un problème peut être modélisé. Par la suite, on présente une version qui considère 8 reines [107]. La figure 1.5 présente une solution à ce problème.

Si on associe une variable Q_i à chaque file de l'échiquier, l'ensemble de contraintes qui modélisent ce problème est le suivant.

- Chaque variable peut prendre une valeur parmi les huit colonnes

$$\forall i \in [1,\ldots,8]: \quad Q_i \in \{1,2,3,4,5,6,7,8\}$$

Le fait que deux reines ne peuvent pas être placées sur la même file est implicite dans la définition des variables.

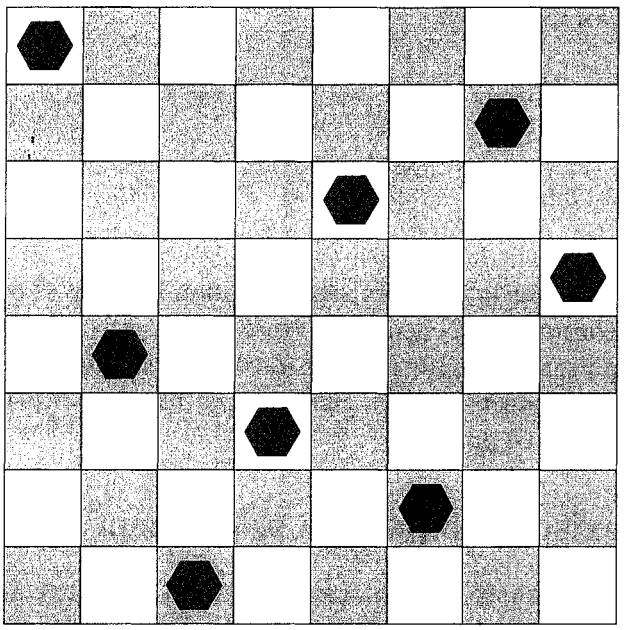

FIG. 1.5 – *Une solution du problème des 8 reines*

- Le fait que deux reines ne peuvent pas être placées sur la même colonne peut être représenté par

$$\forall i, j \in [1, \ldots, 8]; i \neq j : \quad Q_i \neq Q_j$$

- Pour ne pas permettre que deux reines soient sur la même diagonale on définit l'ensemble de contraintes suivant

$$\forall i, j \in [1, \ldots, 8]; i \neq j : \quad Q_i - Q_j \neq i - j \quad \text{et} \quad Q_i - Q_j \neq j - i$$

Une formulation alternative peut être obtenue en considérant que la variable associée à chaque reine représente explicitement chacune des soixante-quatre positions de l'échiquier. Dans ce cas nous pouvons modéliser le problème avec l'ensemble de contraintes suivant.

- Chaque variable peut prendre une valeur parmi les 64 positions

$$\forall i \in [1, \ldots, 8] : \quad Q_i \in \{1, 2, \ldots, 64\}$$

- Si on suppose que les positions sont numérotées de *gauche à droite*, et de *haut en bas*, la file et la colonne d'une reine Q_i peuvent être obtenues à partir de

$$\forall i \in [1, \ldots, 8] : \quad File_i = (Q_i \ \ div \ \ 8) + 1 \quad \text{et} \quad Colonne_i = (Q_i \ \ mod \ \ 8) + 1$$

- Les contraintes sur les files

$$\forall i, j \in [1, \ldots, 8]; i \neq j : \quad File_i \neq File_j$$

- Les contraintes sur les colonnes

$$\forall i, j \in [1, \ldots, 8]; i \neq j : \quad Colonne_i \neq Colonne_j$$

- Les contraintes sur les diagonales

$$\forall i, j \in [1, \ldots, 8]; i \neq j :$$

$$File_i - File_j \neq Colonne_i - Colonne_j \text{ et } File_i - File_j \neq Colonne_j - Colonne_i$$

Cet exemple montre comment la représentation utilisée peut influer dans la complexité du problème à résoudre. En utilisant la première représentation il existe 8^8 combinaisons de valeurs possibles à analyser, par contre, avec la deuxième représentation il en existe 64^8. Cette différence est basée sur le fait que la première représentation encode implicitement les contraintes relatives au fait que deux reines ne peuvent pas être sur la même file. Dans [94], Nadel présente neuf possibilités de modélisation de ce problème.

1.2 Approches existantes pour le traitement de CSPs

Depuis la fin des années soixante, la communauté IA s'est intéressée au problème de satisfaction de contraintes. Les travaux de Fikes [42] et de Waltz [117] marquent le début des efforts de cette communauté. L'idée principale d'éliminer des valeurs prises par les variables, sans modifier l'ensemble de solutions, fut plus tard formalisée par Montanari [88] et Mackworth [76]. Depuis, un grand nombre d'algorithmes ont été proposés, toujours suivant la même idée. Ces algorithmes dits de vérification de consistance ont été incorporés dans des algorithmes de recherche exhaustive créant ainsi les techniques hybrides du style *Forward Checking* et *Intelligent Backtracking*, entre autres. Ces techniques ont été incorporées dans des produits commerciaux comme, par exemple, la librairie ILOG Solver de la société ILOG, qui peut être intégrée dans des applications écrites en C [58]. Un modèle plus riche, considérant une approche à objets et des règles de propagation, est la base du langage CLAIRE [18].

Dans le domaine LP, les contraintes furent introduites par Jaffar et Lassez avec la définition formelle de CLP. Dans ce cadre, l'opération d'unification de la LP traditionnelle est remplacée par la satisfaction de contraintes dans une structure mathématique fixée quelconque représentant le domaine du discours. Pendant l'exécution d'un programme logique traditionnel, les contraintes sont accumulées et résolues par un solveur externe qui peut être considéré comme une boite noire. Afin de permettre à l'utilisateur de définir ses propres règles de propagation, Frühwirth introduit les *Constraint Handling Rules* (CHR) dans le système ECLiPSe [45].

Récemment, Apt [2] a modélisé la résolution de contraintes comme un processus de déduction où le calcul est associé à une preuve constructive d'une formule, la requête. Ce travail est centré sur la proposition d'un ensemble de règles d'inférence générales pour traiter des CSPs et ne concerne pas le problème de l'enchaînement des inférences.

Concernant le traitement de la disjonction dans les systèmes de contraintes, la première approche utilisée fut proposée par Van Hentenryck dans le cadre CLP : une composante de la disjonction est choisie et postée *a priori* durant le processus de résolution de contraintes, si l'ensemble de contraintes ainsi obtenu est inconsistant un retour en arrière est réalisé et une autre composante est choisie et postée [109]. Cette approche est basée sur le mécanisme de retour en arrière des langages à la Prolog et peut être vue comme une énumération des disjonctions.

Une approche plus élaborée dite *disjonction constructive* a été proposée plus tard par Van Hentenryck, Saraswat et Deville [111] : toutes les composantes d'une disjonction sont analysées pour extraire des informations concernant les valeurs qui sont impossibles dans toutes les alternatives.

Le problème d'optimisation de satisfaction de contraintes est largement étudié par la communauté RO. Le travail de Bockmayr et Kasper, où est présenté un cadre unifié pour le traitement de ce problème du point de vue RO, d'une part, et des contraintes, d'autre part, nous semble être la meilleure référence disponible sur le sujet [9]. Tsang [107] et Fages [41] décrivent aussi l'approche contraintes. Brièvement, l'approche contraintes est basée sur la résolution de CSPs dynamiques : à partir d'une solution possible initiale, on essaie de l'améliorer par l'ajout d'une nouvelle contrainte qui impose une meilleure borne pour la fonction d'optimisation.

Toujours dans le cadre CLP, la collaboration de solveurs a été proposée comme une façon d'attaquer des problèmes qui soit ne peuvent pas être résolus avec un seul solveur, soit sont résolus plus efficacement quand on les attaque avec plusieurs solveurs [3, 81, 85]. L'intégration de solveurs requiert des mécanismes appelés *primitives de collaboration* par Monfroy. La séquentialité, le parallélisme et la concurrence ont été proposées comme primitives de collaboration dans le langage **BALI**.

1.3 Rôle des transformations et du contrôle

Les notions de transformation et de contrôle sont présentes depuis longtemps dans la conception des programmes en informatique. Quand on analyse les approches utilisées par les communautés IA et CLP, on peut réaliser que les modèles de programmation sous-jacents sont ceux de *algorithms + data structure = programs* et de *algorithm = logic + control*, respectivement.

Quand on veut prouver, d'un point de vue formel, des propriétés telles que la correction, la complétude et la terminaison, on remarque la nécessité de faire la différence entre les notions de transformation et de contrôle.

D'ailleurs, la résolution de contraintes, étant une activité notamment heuristique, a besoin d'un mécanisme flexible pour le prototypage de nouvelles heuristiques.

Dans ce contexte, l'idée de regarder la résolution de contraintes comme un processus de déduction est intéressant. D'une part, en ce qui concerne la formalisation des transformations, les inférences, et d'autre part, en ce qui concerne la possibilité d'expérimenter des heuristiques à partir d'un ordre différent d'application des inférences.

La principale motivation pour ce travail est de modéliser la résolution de contraintes dans un cadre formel permettant, d'une part, de développer des solveurs corrects, et d'autre part, d'avoir la flexibilité nécessaire pour le prototypage sûr, simple et rapide des heuristiques de résolution de CSPs.

Cet intérêt nous a amené à considérer la possibilité de modéliser un solveur de contraintes comme étant un système de calcul. Les systèmes de calcul, nés au sein de la communauté de déduction automatique, se basent sur la notion de théorie de réécriture pour exprimer les transformations, les inférences, et sur le concept de stratégies qui permettent d'exprimer l'ordre d'application des inférences.

1.4 Objectifs de la thèse

L'objectif général de cette thèse est de valider l'hypothèse qu'un solveur de contraintes peut être consideré comme un système de calcul, i.e., un ensemble de règles de réécriture plus des stratégies qui contrôlent l'ordre d'application des règles afin de mettre l'ensemble de contraintes sous une forme normale.

Ce but général est décomposé dans les objectifs spécifiques suivants.

- Formaliser les algorithmes de résolution de CSPs en faisant une distinction claire entre les actions et le contrôle.

  Compte tenu que le développement des techniques de résolution de contraintes a été fait par deux communautés scientifiques différentes, la communauté IA et la communauté CLP, qui utilisent des paradigmes de programmation différents, il n'est pas toujours évident de reconnaître les principes généraux concernant la résolution de contraintes. Notre premier but est d'arriver à identifier les concepts fondamentaux derrière ces deux approches. À partir de l'analyse de ces techniques, notre objectif est de les exprimer sous une forme unifiée qui fasse une distinction claire entre les transformations et le contrôle.

- Valider dans le cas spécifique des CSPs, l'idée de Kirchner, Kirchner et Vittek qu'un solveur de contraintes peut être vu comme un système de calcul.

  Lors de l'introduction de la notion de systèmes de calcul, ces auteurs postulent la possibilité de voir un solveur de contraintes comme étant un système de calcul. Cette proposition peut être acceptée intuitivement très facilement. L'utilisation de règles de transformation pour exprimer des opérations sur des objets est largement connue. Si on ajoute en plus la possibilité de contrôler les règles en utilisant un langage de stratégies, l'idée devient encore plus réalisable. Malgré cela, la spécification d'un solveur de contraintes nécessite un formalisme capable d'exprimer aussi bien les opérations de transformations que le contrôle. Dans le cadre des systèmes de calcul, les questions qui se posent sont : les règles de réécriture ont-elles le pouvoir suffisant pour exprimer facilement les opérations basiques que l'on fait sur les contraintes ? Est-ce que les stratégies permettent de contrôler l'application d'un ensemble de règles de réécriture d'une façon telle que l'on puisse simuler les heuristiques utilisées pour la résolution de contraintes ?

- Étudier la possibilité de développer une application en dehors du cadre de la réécriture.

  Les applications développées jusqu'au présent dans le cadre des systèmes de calcul pourraient être considérées comme fondamentalement syntaxiques. Le problème d'unification a été particulièrement bien traité et l'approche a montrée toute sa puissance. Notre intérêt est de voir la faisabilité d'appliquer cette approche principalement syntaxique, les techniques de réécriture, pour résoudre des problèmes que l'on pourrait considérer comme plus proches des problèmes pratiques. Ce but est notamment pratique : même si on peut considérer le cadre de la réécriture comme assez puissant pour formaliser les principes généraux de la résolution de contraintes, notre objectif est d'étudier si les considérations pratiques liées au développement d'un produit industriel peuvent être traitées par ces techniques.

- Étudier la possibilité d'utiliser le cadre des systèmes de calcul pour faire de la collaboration de solveurs.

  Au cours des dernières années la nécessité de faire collaborer plusieurs solveurs pour traiter des problèmes qui ne peuvent pas être résolus ou résolus

efficacement par un solveur simple a été signalée. Ainsi, un nouveau problème qui se pose est celui de comment pouvoir coordonner l'interaction entre des solveurs qui travaillent ensemble pour résoudre un problème globalement. La concurrence implicite de la logique de réécriture a été mise en évidence et on pourrait essayer de formaliser non seulement un solveur simple mais aussi une collaboration de plusieurs solveurs sous la forme de règles de réécriture et de stratégies. Bien que cette voie laisse prévoir *a priori*, en général, des résultats positifs, il est intéressant d'étudier en détail les possibilités réelles qu'offrent les systèmes de calcul pour exprimer tout le contrôle nécessaire pour simuler la collaboration de solveurs. Quelques schémas de collaboration ont déjà été proposés [85] : le schéma de *solveurs séquentiels* a lieu quand la sortie d'un solveur devient l'entrée d'un autre solveur et ainsi de suite, le schéma de *solveurs concurrents* a lieu quand les solveurs tournent de façon concurrente et dès qu'un résultat est obtenu tous les solveurs sont arrêtés et éventuellement relancés à nouveau à partir de ce résultat. On peut aussi imaginer d'autres schémas de collaboration, et notre but sera donc d'étudier la possibilité de simuler tous ces schémas dans le cadre des systèmes de calcul.

1.5 Description de la thèse

Le plan de ce document est le suivant.

Le **chapitre 2** a pour but de rappeler les concepts fondamentaux utilisés au cours de cette thèse. Nous commençons par les concepts concernant l'algèbre universelle. Nous présentons ensuite la logique de réécriture et les systèmes de calcul qui constituent la base logique de ce travail. Nous présentons les quatre composants qui définissent la logique de réécriture : sa syntaxe, son système de déduction, son modèle et une relation de satisfaisabilité. À partir de la notion de théorie de réécriture et du concept de stratégie, nous rappelons la notion de système de calcul. Ce chapitre se termine avec la présentation des aspects relatifs à notre travail du langage utilisé pour le prototypage des systèmes proposés dans cette thèse : le langage ELAN.

Le **chapitre 3** présente tout d'abord un résumé des techniques existantes pour la résolution de CSPs sur les domaines finis et ensuite un système de calcul proposé pour la résolution de conjonctions de contraintes élémentaires.

Nous abordons ici le problème de la résolution d'un ensemble de contraintes élémentaires qui sont combinées seulement par des opérateurs de conjonction. À partir de l'analyse des algorithmes de résolution de CSPs, nous proposons cinq règles de réécriture qui expriment les transformations fondamentales réalisées par ces algorithmes. En utilisant le langage de stratégies d'ELAN, nous simulons plusieurs heuristiques utilisées par la communauté CSP pour trouver une solution ou toutes les solutions. Ces exemples montrent que les heuristiques basées sur un *backtracking* chronologique peuvent être simulées très facilement, mais des stratégies basées sur un *backtracking* intelligent sont difficiles à simuler car elles réalisent des sauts dans l'arbre de recherche qui ne sont pas évidents à décrire avec le langage de stratégies lequel est basé sur un mécanisme de retour en arrière.

Le **chapitre 4** est dédié à l'étude d'une extension des CSPs : le problème d'optimisation de satisfaction de contraintes.

Dans ce chapitre, nous rappelons les approches de programmation linéaire entière (ILP[5]) et de contraintes pour la résolution du problème d'optimisation sur des variables entières. À partir de ces deux approches, nous modélisons le problème d'optimisation sous la forme d'un système de calcul. Nous abordons aussi quelques considérations liées à l'implantation de ce système.

Le **chapitre 5** est destiné à l'étude de la disjonction dans le contexte des CSPs.

Nous nous intéressons à l'extension du langage de contraintes présenté dans le chapitre 3 afin de considérer la combinaison des contraintes élémentaires avec les connecteurs logiques de disjonction, d'implication et d'équivalence. Pour cela nous ramenons ce problème à la résolution de problèmes en forme normale conjonctive. Dans ce cadre, nous trouvons un problème additionnel, lequel consiste à gérer la disjonction entre contraintes élémentaires ce qui rend le problème encore plus dur. Dans ce chapitre, nous présentons les approches existantes pour traiter la disjonction : l'utilisation de variables binaires proposée par la communauté de programmation linéaire, la gestion de points de choix et la disjonction constructive utilisées par la communauté LP. À partir de la notion de disjonction constructive, nous proposons une nouvelle approche qui peut être considérée comme une extension basée principalement sur des considérations d'implantation de la disjonction constructive.

Le **chapitre 6** est consacré à l'étude de la coopération de solveurs dans le cadre des systèmes de calcul.

Vu que l'efficacité des solveurs travaillant de manière isolée peut être améliorée et qu'il y a des cas où l'utilisation d'un seul solveur ne donne pas de résultats satisfaisants, l'idée de faire collaborer plusieurs solveurs commence a être largement étudiée par la communauté CSP. Dans ce chapitre, nous nous intéressons au cas de solveurs qui travaillent sur un même domaine d'interprétation, ce type de collaboration est appelé coopération de solveurs. À partir des solveurs qui sont proposés dans des chapitres précédents, nous simulons la coopération de plusieurs d'entre eux. En utilisant les opérateurs de stratégies concurrents du langage ELAN, nous sommes capables de simuler des solveurs séquentiels et des solveurs concurrents d'une manière très abstraite. Cependant, pour d'autres types de coopération, nous sommes obligés d'utiliser des primitives de bas niveau qui ont été conçues pour la communication entre processus.

Le **chapitre 7** contient les conclusions et les perspectives de ce travail. Nous résumons les apports de ce travail et nous énonçons un certain nombre de perspectives de recherche qui nous semblent intéressantes à considérer.

Finalement, dans l'annexe de ce document nous présentons des jeux d'essais que nous avons réalisé pour tester le système COLETTE que nous avons implanté pour valider les idées proposées.

5. Integer Linear Programming.

Chapitre 2

Cadre logique

Sommaire

Au cours du développement du génie logiciel, diverses approches ont été proposées pour définir la notion de programme informatique. Dans les années soixante-dix, Wirth [121] introduit l'équation

$$\text{Algorithms} + \text{Data Structures} = \text{Programs}$$

Vers la fin des années soixante-dix, Kowalski [71] présente la notion de

$$\text{Algorithm} = \text{Logic} + \text{Control}$$

Dans les années quatre-vingt-dix, Kirchner, Kirchner et Vittek [63] introduisent le concept de système de calcul, où un programme est vu comme étant un ensemble de règles de réécriture contrôlées par des stratégies

$$\text{Programme} = \text{Règles} + \text{Stratégies}$$

Ce cadre, apparu dans le domaine de la déduction automatique, permet de lier des règles de transformation à des règles de réécriture, et en plus, nous donne la possibilité d'exprimer leur enchaînement en utilisant la notion de stratégie.

Dans ce chapitre, nous présentons l'outil utilisé pour modéliser la manipulation de CSPs : les systèmes de calcul. Nous commençons par rappeler les concepts d'algèbre universelle dont nous aurons besoin tout au long de ce travail. Basés sur la notion de logique de réécriture, nous rappelons ensuite le concept de système de calcul. Nous présentons finalement une description du langage ELAN, qui sera utilisé pour mettre en œuvre les systèmes proposés.

2.1 Définitions de base

Les concepts et les définitions présentés dans cette section sont basés sur [48, 67, 61, 75, 109].

Définition 1 (Signature) *On appelle* arité *une fonction qui affecte un entier non-négatif arité(f) à chaque symbole de fonction f. Soit $\mathcal{F}_n$ l'ensemble de tous les symboles de fonction d'arité n. L'ensemble de tous les symboles de fonction est défini par $\mathcal{F} = \bigcup_{n\geq 0}\mathcal{F}_n$. Les éléments d'arité zéro sont appelés* constantes. *$\mathcal{F}$ est généralement appelé un ensemble de symboles de fonctions indexés ou une signature non-sortée ou monosortée.*

Les constantes sont désignées par les lettres $a, b, c, \ldots$, et les symboles de fonctions par f, g. Tous ces symboles peuvent être utilisés avec des indices.

Définition 2 (Terme du premier ordre) *Étant donnés un ensemble de symboles de fonctions $\mathcal{F}$ et un ensemble dénombrable de symboles de variables $\mathcal{X}$, l'ensemble de termes du premier ordre $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est le plus petit ensemble qui contient $\mathcal{X}$ et tel que la chaîne $f(t_1, t_2, \ldots, t_n)$ appartient à $\mathcal{T}(\mathcal{F}, \mathcal{X})$, où $f \in \mathcal{F}_n$ et $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ pour tout $i \in [1, \ldots, n]$. Un terme sans variable est dit* terme clos *et un terme qui contient des variables est dit* ouvert. *Un terme ouvert est* linéaire *si chacune de ses variables n'y apparaît qu'une fois.*

Les variables sont désignées par x, y, z, et les termes par s, t. Tous ces symboles peuvent inclure des indices. $\mathcal{V}ar(t)$ désigne l'ensemble des variables dans t. L'ensemble de termes clos est désigné par $\mathcal{T}(\mathcal{F})$.

Exemple 1 *Étant donnés $\mathcal{F} = \{+, *, a\}$ et $\mathcal{X} = \{x\}$, avec arité(+)=arité(*)=2 et arité(a)=0,*

$$a + a * a$$

est un terme clos et

$$a * x + a * a$$

est un terme linéaire ouvert.

Définition 3 (Substitution) *Une* substitution *est une application sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$ déterminée uniquement par l'image d'un ensemble fini de variables. Elle est notée $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. L'application d'une substitution $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ sur un terme t est définie récursivement de la manière suivante*

- *si t est une variable x_i pour un certain $i \in [1, \ldots, n]$, alors $\sigma(t) = t_i$;*
- *si t est une variable $x \neq x_i$ pour tout $i \in [1, \ldots, n]$, alors $\sigma(t) = t$;*
- *si t est un terme $f(u_1, \ldots, u_k)$, alors $\sigma(t) = f(\sigma(u_1), \ldots, \sigma(u_k))$.*

Exemple 2 *Étant donnés le terme $t = a * x + a * y$ et la substitution $\sigma = \{x \mapsto a + a, y \mapsto x\}$, l'application de σ sur t donne comme résultat le terme*

$$\sigma(t) = a * a + a + a * x$$

Définition 4 (Signature $\Sigma = (\mathcal{F}, \mathcal{P})$) *Soit p un symbole de prédicat et* arité *une fonction qui affecte un entier non-négatif arité(p) à chaque symbole de prédicat p. Soit $\mathcal{P}_n$ l'ensemble de tous les symboles de prédicats d'arité n. L'ensemble de tous les symboles de prédicats est défini par $\mathcal{P} = \bigcup_{n \geq 0} \mathcal{P}_n$. La notion de signature peut être étendue afin d'inclure des symboles de prédicats. Étant donnés un ensemble de symboles de fonctions $\mathcal{F}$ et un ensemble de symboles de prédicats $\mathcal{P}$, une signature Σ est l'ensemble de tous les symboles de fonctions de $\mathcal{F}$ et tous les symboles de prédicats de $\mathcal{P}$. Cette signature est désignée par $\Sigma = (\mathcal{F}, \mathcal{P})$.*

Les symboles de prédicats sont désignés par les lettres p, q. Tous ces symboles peuvent avoir des indices.

Définition 5 (Formule du premier ordre bien formée) *Étant donnés une signature $\Sigma = (\mathcal{F}, \mathcal{P})$ et un ensemble dénombrable de symboles de variables $\mathcal{X}$, une* Σ-formule du premier ordre bien formée *est définie inductivement de la manière suivante*

- *si $t_1, \ldots, t_n$ sont des termes et p est un symbole de prédicat d'arité $n > 0$, alors $p(t_1, \ldots, t_n)$ est une Σ-formule (appelée une formule atomique ou simplement un atome) ;*
- *si A et B sont des Σ-formules, alors $(\neg A), (A \wedge B), (A \vee B), (A \rightarrow B)$, et $(A \leftrightarrow B)$ sont aussi des Σ-formules ;*
- *si A est une Σ-formule et x est une variable, alors $\forall x A$ et $\exists x A$ sont des Σ-formules.*

Lorsqu'il n'y a pas de confusion, le mot *formule* est utilisé pour désigner une Σ-formule. La portée de $\forall x$ dans $\forall x A$ et de $\exists x$ dans $\exists x A$ est A.

Une *occurrence* de la variable x est liée si elle apparaît dans la portée d'un quantificateur $\forall x$ ou $\exists x$. Toute autre occurrence d'une variable est *libre*. Une formule est *complètement quantifiée* si toutes ses variables sont quantifiées. Une Σ-théorie est un ensemble de Σ-formules complètement quantifiées.

Exemple 3 *Étant donnés $\Sigma = (\mathcal{F}, \mathcal{P})$ et $\mathcal{X} = \{x, y\}$, où $\mathcal{F} = \{+, *, a\}$ et $\mathcal{P} = \{=, \geq\}$ avec arité($+$) = arité($*$) = arité($=$) = arité($\geq$) = 2 et arité(a) = 0*

- *$a + x = a * a$ est une formule atomique ouverte ;*
- *$\exists x\ (y + x) \wedge (x * a)$ est une formule où les occurrences de x sont liées, mais l'occurrence de y est libre ;*
- *$\exists x, y\ (y + a) \vee (x * a)$ est une formule complètement quantifiée et donc elle appartient à la Σ-théorie.*

Définition 6 (Σ-structure) *Étant donnée une signature Σ, une* Σ-structure *$\mathcal{D}$ est une paire $\mathcal{D} = (D, I)$, où D est un ensemble non-vide, dit le* domaine *de la structure, et I est une fonction, dite l'interprétation, qui affecte les fonctions et les relations dans D aux symboles en $\mathcal{F}$ et $\mathcal{P}$, respectivement. Cette affectation des fonctions et des relations est définie de la manière suivante*

- *pour tout symbole de fonction f d'arité $n > 0$, une interprétation $I(f)$ dans D est une fonction d'arité n de D^n vers D ;*
- *pour tout symbole de fonction f d'arité $n = 0$, i.e., une constante, une interprétation $I(f)$ dans D correspond à l'affectation d'un élément de D ;*
- *pour tout ensemble de symboles de fonctions $\mathcal{F}$, une interprétation $I(\mathcal{F})$ de $\mathcal{F}$ dans l'ensemble D est une application qui associe à chaque symbole de fonction dans $\mathcal{F}$ une interprétation dans D ;*

- *pour tout symbole de prédicat p d'arité $n \geq 0$, une interprétation $I(p)$ dans D est une relation d'arité n dans D^n ;*
- *pour tout ensemble de symboles de prédicats $\mathcal{P}$, une interprétation $I(\mathcal{P})$ de $\mathcal{P}$ dans l'ensemble D est une application qui associe à chaque symbole de prédicat dans $\mathcal{P}$ une interprétation dans D.*

Les interprétations $I(f)$, $I(\mathcal{F})$, $I(p)$ et $I(\mathcal{P})$ sont aussi désignées par $f_{\mathcal{D}}$, $\mathcal{F}_{\mathcal{D}}$, $p_{\mathcal{D}}$ et $\mathcal{P}_{\mathcal{D}}$, respectivement.

Exemple 4 *Soit $\Sigma = (\mathcal{F}, \mathcal{P})$ une signature et $\mathcal{X} = \{x\}$ un ensemble de symboles de variables, où $\mathcal{F} = \{+, *, a\}$ et $\mathcal{P} = \{=\}$, avec arité(+) = arité(*) = arité(=) = 2 et arité(a)=0. Soit $\mathcal{D} = (\mathbb{N}, I)$ une Σ-structure, où $\mathbb{N}$ est l'ensemble des nombres naturels, $+_{\mathcal{D}}$ est l'addition, $*_{\mathcal{D}}$ est la multiplication, $a_{\mathcal{D}}$ est le nombre naturel 1 et $=_{\mathcal{D}}$ est la relation d'égalité. Toutes les formules construites à partir de $\mathcal{D}$ sont des équations sur les nombres naturels, comme par exemple*

$$a_{\mathcal{D}} +_{\mathcal{D}} x =_{\mathcal{D}} a_{\mathcal{D}} *_{\mathcal{D}} a_{\mathcal{D}}$$

Définition 7 (Interprétation) *Soit $\mathcal{D} = (D, I)$ une Σ-structure et $\mathcal{X}$ un ensemble de symboles de variables.*

- *Une* assignation des variables *de $\mathcal{X}$ par rapport à I est une fonction α qui affecte à chaque variable dans $\mathcal{X}$ un élément de D. L'assignation d'une variable est notée $\alpha(x)$ et l'ensemble de toutes les assignations de $\mathcal{X}$ dans $\mathcal{D}$ est noté $\alpha_{\mathcal{D}}^{\mathcal{X}}$.*
- *Une* assignation d'un terme *non-variable par rapport à I est définie récursivement par*

$$\alpha(f(t_1, \ldots, t_n)) = f_{\mathcal{D}}(\alpha(t_1), \ldots, \alpha(t_n))$$

où $f_{\mathcal{D}}$ est l'interprétation du symbole de fonction f d'arité n. L'assignation d'un terme t par rapport à I et α est désignée par $\alpha(t)$.

- *La valeur de vérité,* vrai *(**V**) ou* faux *(**F**), d'une formule dans $\mathcal{D}$ est obtenue de la manière suivante*

  - *si la formule est un atome $p(t_1, \ldots, t_n)$, alors sa valeur de vérité est obtenue récursivement par*

$$\alpha(p(t_1, \ldots, t_n)) = p_{\mathcal{D}}(\alpha(t_1), \ldots, \alpha(t_n))$$

  où $p_{\mathcal{D}}$ est l'interprétation du symbole de prédicat p d'arité n.
  - *si la formule a la forme $(\neg A), (A \wedge B), (A \vee B), (A \rightarrow B)$, ou $(A \leftrightarrow B)$, alors sa valeur de vérité est donnée par le tableau présenté dans le tableau 2.1 ;*

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
V	**V**	**F**	**V**	**V**	**V**	**V**
V	**F**	**F**	**F**	**V**	**F**	**F**
F	**V**	**V**	**F**	**V**	**V**	**F**
F	**F**	**V**	**F**	**F**	**V**	**V**

TAB. 2.1 – *Valeur de vérité des connecteurs logiques*

- *si la formule a la forme $\exists x A$, alors sa valeur de vérité est vrai s'il existe $d \in D$ tel que A a la valeur de vérité vrai par rapport à I et $\alpha_{|x \mapsto d}$, où $\alpha_{|x \mapsto d}$ est α sauf que x est affecté à d ; sinon, sa valeur de vérité est faux ;*
- *si la formule a la forme $\forall x A$, alors sa valeur de vérité est vrai si, pour tout $d \in D$, A a une valeur de vérité vrai par rapport à I et $\alpha_{|x \mapsto d}$; sinon, sa valeur de vérité est faux.*

L'interprétation d'une formule A par rapport à I et α est désignée par $\alpha(A)$.

Définition 8 (Modèles) *Soit Σ une signature et A une formule.*

- *Étant données une Σ-structure $\mathcal{D}$ et une affectation α,* $\mathcal{D}$ satisfait A avec α *si*

$$\alpha(A) = \mathbf{V}$$

Ceci est aussi noté

$$(\mathcal{D}, \alpha) \models A$$

- *Étant donnée une Σ-structure $\mathcal{D}$, la formule A est* satisfaisable dans $\mathcal{D}$ *s'il existe une affectation α telle que*

$$\alpha(A) = \mathbf{V}$$

- *La formule A est* satisfaisable *s'il existe une Σ-structure $\mathcal{D}$ dans laquelle A est satisfaisable.*
- *Étant donnée une Σ-structure $\mathcal{D}$, la formule A est* valide dans $\mathcal{D}$ *(ou vrai en $\mathcal{D}$) si pour toute affectation α*

$$\alpha(A) = \mathbf{V}$$

Ceci est désigné par

$$\mathcal{D} \models A$$

Dans ce cas, $\mathcal{D}$ est appelée un modèle *de A.*
- *La formule A est* valide *(ou* universellement valide*) si elle est valide dans chaque Σ-structure $\mathcal{D}$. Ceci est désigné par*

$$\models A$$

- *Étant donnée une Σ-théorie T, T est* satisfaisable *s'il existe une structure $\mathcal{D}$ et une affectation α telle que*

$$(\mathcal{D}, \alpha) \models A$$

pour toute formule $A \in T$.
- *Étant données une Σ-structure $\mathcal{D}$ et une Σ-théorie T, $\mathcal{D}$ est un* modèle *de T si $\mathcal{D}$ est un modèle de chaque formule en T. Ceci est désigné par*

$$\mathcal{D} \models T$$

- *Étant donnée une Σ-théorie T, T est* valide *si*

$$\mathcal{D} \models T$$

pour toute Σ-structure $\mathcal{D}$. Ceci est désigné par

$$\models T$$

- *Étant données une Σ-théorie T et une formule B, B est une* conséquence sémantique *de T, désignée par*

$$T \models B$$

si pour chaque Σ-structure $\mathcal{D}$, pour chaque affectation α et pour toute formule $A \in T$ si

$$(\mathcal{D}, \alpha) \models A$$

alors

$$(\mathcal{D}, \alpha) \models B$$

2.2 Logique de réécriture

Une logique est définie en général par une syntaxe, un système de déduction, une classe de modèles et une relation de satisfaisabilité. Dans cette section, nous présentons ces quatre composantes dans le cas de la logique de réécriture.

Syntaxe

La syntaxe nécessaire pour définir une logique est spécifiée par sa signature qui nous permet de construire des formules.

Définition 9 (Signature de la logique de réécriture) *Soit $\mathcal{X}$ un ensemble de variables et $\mathcal{L}$ un ensemble de symboles appelés étiquettes. La signature de la logique de réécriture est un triplet*

$$\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{E})$$

où $\mathcal{S}$ est un ensemble de sortes, $\mathcal{F}$ est un ensemble de symboles de fonctions et $\mathcal{E}$ est un ensemble d'axiomes équationnels dans $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Les axiomes équationnels dans $\mathcal{E}$ doivent être interprétées comme étant des axiomes exprimés sur la signature. Les formules $sen(\Sigma)$ formées sur la signature Σ sont définies comme des *séquents* $Seq(\Sigma)$ de la forme suivante

$$\pi : [t]_{\mathcal{E}} \rightarrow [t']_{\mathcal{E}}$$

où $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $\pi \in \mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{;\})$.

π est appelé un terme de preuve et l'ensemble de tous ces termes de preuve $\mathcal{T}(\mathcal{F} \cup \mathcal{L} \cup \{;\})$ est désigné par Π.

Le sens informel du séquent $\pi : [t]_{\mathcal{E}} \rightarrow [t']_{\mathcal{E}}$ est que π permet de dériver les termes de la classe d'équivalence $[t']_{\mathcal{E}}$ à partir des termes de la classe d'équivalence $[t]_{\mathcal{E}}$ et que le terme de preuve π représente une preuve de cette dérivation.

Système de déduction

Afin de construire le système de déduction de la logique de réécriture, on introduit d'abord la notion de *théorie de réécriture.*

Définition 10 (Théorie de réécriture) *Une* théorie de réécriture *est définie par un 4-uplet* $\mathcal{TR} = (\Sigma, \mathcal{L}, \mathcal{X}, \mathcal{R})$, *où* $\Sigma = (\mathcal{S}, \mathcal{F}, \mathcal{E})$ *est une signature composée des sortes* $\mathcal{S}$, *des symboles de fonctions* $\mathcal{F}$ *et des équations* $\mathcal{E}$ *dans* $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $\mathcal{X}$ *est un ensemble infini de variables,* $\mathcal{L}$ *est un ensemble d'étiquettes des règles et* $\mathcal{R}$ *est un ensemble de règles de réécriture étiquetées de la forme*

$$[\ell] \quad l \Rightarrow r$$

où l'étiquette $\ell \in \mathcal{L}$, *les membres gauche et droit* $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ *tels que* $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$ *et l'arité de l'étiquette* ℓ *est égale au nombre de variables distinctes dans cette règle.*

L'ensemble d'équations $\mathcal{E}$ définit une relation de congruence modulo laquelle la réécriture par les règles de $\mathcal{R}$ est réalisée. Typiquement, l'ensemble $\mathcal{E}$ contient des équations qui ne sont pas orientables, i.e., transformables en un système de réécriture terminant. Cependant, la terminaison, et aussi la confluence, peuvent être des propriétés souhaitables pour certains sous-ensembles de règles de $\mathcal{R}$.

La relation de déduction $\vdash$ est donc définie comme suit.

Définition 11 (Système de déduction) *Étant donnée une théorie de réécriture étiquetée* $\mathcal{TR}$, *le séquent* $\pi : [t]_\mathcal{E} \to [t']_\mathcal{E}$ *se déduit à partir de* $\mathcal{TR}$ *si* π *est obtenu en appliquant un nombre fini de fois* les règles de déduction de la logique de réécriture *données dans la figure 2.1. Ceci est désigné par*

$$\mathcal{TR} \vdash \pi : [t]_\mathcal{E} \to [t']_\mathcal{E}$$

Réflexivité

$$\Rightarrow$$
$$[t]_\mathcal{E} : [t]_\mathcal{E} \to [t]_\mathcal{E}$$
si $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

Congruence

$$\pi_1 : [t_1]_\mathcal{E} \to [t'_1]_\mathcal{E}, \ldots, \pi_n : [t_n]_\mathcal{E} \to [t'_n]_\mathcal{E}$$
$$\Rightarrow$$
$$f(\pi_1, \ldots, \pi_n) : [f(t_1, \ldots, t_n)]_\mathcal{E} \to [f(t'_1, \ldots, t'_n)]_\mathcal{E}$$
si $f \in \mathcal{F}_n$

Remplacement

$$\pi_1 : [t_1]_\mathcal{E} \to [t'_1]_\mathcal{E}, \ldots, \pi_n : [t_n]_\mathcal{E} \to [t'_n]_\mathcal{E}$$
$$\Rightarrow$$
$$\ell(\pi_1, \ldots, \pi_n) : [l(t_1, \ldots, t_n)]_\mathcal{E} \to [r(t'_1, \ldots, t'_n)]_\mathcal{E}$$
si $[\ell(x_1, \ldots, x_n)]l(x_1, \ldots, x_n) \to r(x_1, \ldots, x_n) \in \mathcal{R}$

Transitivité

$$\pi_1 : [t_1]_\mathcal{E} \to [t_2]_\mathcal{E}, \pi_2 : [t_2]_\mathcal{E} \to [t_3]_\mathcal{E}$$
$$\Rightarrow$$
$$\pi_1; \pi_2 : [t_1]_\mathcal{E} \to [t_3]_\mathcal{E}$$

FIG. 2.1 – *Règles de déduction de la logique de réécriture*

Modèle

Le modèle de la logique de réécriture présenté ici est basé sur une axiomatisation algébrique des séquents de réécriture. En particulier, on s'intéresse à une *sémantique algébrique.*

La sémantique algébrique permet de décrire l'idée intuitive d'un système de réécriture: les états du système sont des classes d'équivalence de termes modulo $\mathcal{E}$ et les transitions sont des réécritures utilisant les règles du système de réécriture. Ainsi, l'espace des calculs de la théorie de réécriture $\mathcal{TR}$ peut être choisi comme un modèle de la logique de réécriture. Cet espace des calculs est déterminé par l'ensemble des termes de preuves π calculés dans les séquents $\pi : [t]_\mathcal{E} \to [t']_\mathcal{E}$ modulo une équivalence de calcul. Cette équivalence est donnée par $\mathcal{E}$ et un ensemble $\mathcal{E}_\Pi$ d'axiomes équationnels sur les termes de preuves décrits dans la figure 2.2, où

- les deux premiers axiomes décrivent les équations habituelles d'associativité et d'identité ;
- l'axiome de préservation de composition décrit une équivalence entre la composition de plusieurs pas de réécriture dans le contexte « f » et la composition de chaque pas de réécriture dans ce contexte ;
- l'axiome de préservation d'identités $\mathcal{E}$ décrit la stabilité par contexte de $\mathcal{E}$;
- l'équivalence induit par les cinq premières équations définit des termes de preuve équivalents tels que les dérivations correspondantes diffèrent uniquement par l'ordre de réduction de radicaux ;
- l'axiome de permutation parallèle décrit la réduction *simultanée* de radicaux compatibles. Ceci peut être simulé par une composition d'exécution séquentielle [47]. Intuitivement, la réécriture au sommet par une règle ℓ et une réécriture en dessous sont des processus indépendants ce qui permet ainsi leur exécution dans n'importe quel ordre.

Associativité	$\forall \pi_1, \pi_2, \pi_3 \in \Pi$ $\pi_1; (\pi_2; \pi_3) = (\pi_1; \pi_2); \pi_3$
Identités	$\forall \pi : [t]_\mathcal{E} \to [t']_\mathcal{E}$, $\pi; [t']_\mathcal{E} = \pi$, et $[t]_\mathcal{E}; \pi = \pi$
Préservation de composition	$\forall f \in \mathcal{F}_n, n = \text{arité}(f), \forall \pi_1, \ldots, \pi_n, \pi'_1, \ldots, \pi'_n$: $f(\pi_1; \pi'_1, \ldots, \pi_n; \pi'_n) = f(\pi_1, \ldots, \pi_n); f(\pi'_1, \ldots, \pi'_n)$
Préservation d'identités	$\forall f \in F_n, n = \text{arité}(f)$: $f([t_1]_\mathcal{E}, \ldots, [t_n]_\mathcal{E}) = [f(t_1, \ldots, t_n)]_\mathcal{E}$
Axiomes de E	$\forall u = v \in \mathcal{E}, \forall \pi_1, \ldots, \pi_n$: $u(\pi_1, \ldots, \pi_n) = v(\pi_1, \ldots, \pi_n)$
Permutation parallèle	$\forall [\ell]\ l \to r \in \mathcal{R}, \forall \pi_1 : [t_1]_\mathcal{E} \to [t'_1]_\mathcal{E}, \ldots, \pi_n : [t_n]_\mathcal{E} \to [t'_n]_\mathcal{E}$ $\ell(\pi_1, \ldots, \pi_n) = \ell([t_1]_\mathcal{E}, \ldots, [t_n]_\mathcal{E}); r(\pi_1, \ldots, \pi_n)$ et $\ell(\pi_1, \ldots, \pi_n) = l(\pi_1, \ldots, \pi_n); \ell([t'_1]_\mathcal{E}, \ldots, [t'_n]_\mathcal{E})$

FIG. 2.2 – *Équivalence des termes de preuves* – $\mathcal{E}_\Pi$

Le modèle considéré est un ensemble quotient noté

$$\mathcal{T}_{\mathcal{TR}} = \{\pi \mid \mathcal{TR} \vdash \pi : [t] \to [t']\}/(\mathcal{E} \cup \mathcal{E}_\Pi)\}$$

Satisfaisabilité

La relation de satisfaisabilité $\models \subseteq \mathcal{T}_{\mathcal{TR}} \times Seq(\Sigma)$ doit être compatible avec les morphismes des signatures. Elle est définie dans [82].

2.3 Systèmes de calcul

Les systèmes de calcul furent introduits par Kirchner, Kirchner et Vittek dans [63], où ils présentent une version plus élaborée des idées qu'ils avaient proposées originalement dans [62]. Un système de calcul enrichit le formalisme de la logique de réécriture avec une notion de *stratégie*: un *système de calcul* est composé d'une théorie de réécriture et d'un *système de stratégies*. Les stratégies contrôlent l'application des règles de réécriture en spécifiant des parcours dans l'arbre de toutes les dérivations possibles et de cette façon décrivent quels sont les nœuds considérés comme des résultats d'un calcul. Elles sont utilisées, d'une part, pour décrire le déroulement de preuves qui nous intéressent et, d'autre part, pour restreindre l'espace de recherche de ces preuves.

La première composante d'un système de calcul est une théorie de réécriture $\mathcal{TR}$ à partir de laquelle on définit la notion de calcul.

Définition 12 (Pas de réécriture simple) *Étant donnés une théorie de réécriture $\mathcal{TR}$ et un séquent $\pi : [t]_\mathcal{E} \to [t']_\mathcal{E}$ avec le terme de preuve $\pi = t[\ell(\sigma x)]_\omega$, un* pas de réécriture simple *est défini par*

$$[t]_\mathcal{E} \Rightarrow_{\ell,\sigma,\omega} [t']_\mathcal{E}$$

Cette définition correspond exactement à la notion traditionnel d'un pas de réécriture dans la position ω en utilisant la règle étiquetée avec l'étiquette ℓ et le match σ.

En plus, on s'intéresse à une représentation canonique de tous les calculs qui sont équivalents modulo les axiomes $\mathcal{E}_{\Pi(R)}$. Puisque tout séquent peut être décomposé en une composition de séquents élémentaires (séquentiels)

$$\forall \pi : [t]_\mathcal{E} \to [t']_\mathcal{E}$$

soit

$$\pi = [t]_\mathcal{E} = [t']_\mathcal{E}$$

soit

$$\exists n \in \mathbb{N} \text{ tel que } [t]_\mathcal{E} = [t_0]_\mathcal{E} \Rightarrow_{\ell_0} [t_1]_\mathcal{E} \Rightarrow_{\ell_1} [t_2]_\mathcal{E} \ldots \Rightarrow_{\ell_{n-1}} [t_n]_\mathcal{E} = [t']_\mathcal{E}$$

et

$$\pi =_{A(;)} (\pi_0; \pi_1; \pi_2; \ldots; \pi_{n-1})$$

où $A(;)$ désigne l'associativité du « ; ».

Pour un terme de preuve π, $[t_n]_\mathcal{E}$ est appelé le *résultat de l'application de* π *sur* $[t_0]_\mathcal{E}$ et il est aussi désigné par $[t]_\mathcal{E} \stackrel{\pi}{\Rightarrow} [t']_\mathcal{E}$. La relation d'équivalence générée par $(E \cup \mathcal{E}_{\Pi(R)})$ sur les termes de preuve induit une équivalence sur les calculs: deux calculs sont équivalents, ils amènent au même résultat, si leurs termes de preuve sont équivalents.

En général, on ne s'intéresse pas à tous les calculs, on s'intéresse seulement à ceux guidés par une stratégie, c'est-à-dire une description de la séquence de pas de réécriture élémentaires permis par les calculs. D'un point de vue formel, une stratégie est un ensemble de termes de preuve, i.e., un sous-ensemble des termes de preuve Π, qui est clos par concaténation.

La relation de transition $\pi : [t]_\mathcal{E} \Rightarrow [t']_\mathcal{E}$ peut être étendue pour les stratégies.

Définition 13 (Stratégie) *Soient* $S \subseteq \Pi$ *et* $t, t' \in \mathcal{T}(\mathcal{F})$. *La relation*

$$S : [t]_\mathcal{E} \Rightarrow [t']_\mathcal{E}$$

est vraie s'il existe un terme de preuve $\pi \in S$ *tel que*

$$\pi : [t]_\mathcal{E} \Rightarrow [t']_\mathcal{E}$$

Le résultat de l'application d'une stratégie S *sur un terme* t, *désigné fonctionnellement par* $S(t)$, *est défini comme suit*

$$S(t) = \{[t']_\mathcal{E} | \exists \pi \in S, [t]_\mathcal{E} \stackrel{\pi}{\Rightarrow} [t']_\mathcal{E}\}$$

La relation $S : t \Rightarrow t'$ exprime la dérivabilité du terme t en t' suivant une certaine stratégie S. À partir de cette définition, on peut noter que l'application d'une stratégie sur un terme peut retourner plusieurs résultats.

Une première façon de décrire une stratégie est d'énumérer extensivement le sous-ensemble des termes de preuve. Évidement, cette approche n'est pas satisfaisante en pratique, donc le problème est de définir un langage permettant de décrire des sous-ensembles des termes de preuve. La différence entre la représentation d'une stratégie comme un ensemble de termes de preuve et une expression dans un formalisme de stratégies reflète la différence entre la vue sémantique des stratégies et la vue syntaxique des stratégies exprimées sous la forme d'un programme dans un langage de stratégies. Dans la section 2.4, on présentera le langage de stratégies utilisé dans ce travail.

On peut maintenant définir formellement la notion de système de calcul.

Définition 14 (Système de calcul) *Un* système de calcul *est composé d'une théorie de réécriture* $\mathcal{TR} = (\Sigma, \mathcal{L}, \mathcal{X}, \mathcal{R})$ *et d'une stratégie* S.

2.4 Langage ELAN

Le langage ELAN a été conçu au sein du projet PROTHEO à Nancy au début des années quatre-vingt-dix. Sa première version est décrite dans le travail de Vittek [113] et son implantation est détaillée dans [64]. Au cours des années, le langage a évolué et depuis le début de l'année 1998 la version 3.0 est disponible [14]. C'est cette version du langage que nous utiliserons tout au long de ce travail.

ELAN a été conçu comme un cadre logique pour le prototypage de systèmes de calcul. Du point de vue de la programmation, le langage offre la possibilité de spécifier des systèmes de

calcul composés de théories de réécriture multi-sortées, chacune décrite par une signature et par un ensemble de règles de réécriture et de stratégies d'exécution.

- La signature définit les sortes et les symboles de fonctions utilisés dans la description de la théorie. ELAN permet d'utiliser des symboles libres et associatifs-commutatifs, qui peuvent être spécifiés en utilisant une notation *mix-fix*.
- L'ensemble de règles de réécriture est composé de règles non-nommées et de règles nommées ou étiquetées.
  - Les règles non-nommées sont utilisées pour la normalisation de termes. Leur application n'est pas contrôlée par l'utilisateur, elles sont exécutées avec une stratégie pré-définie dans le langage. Cette stratégie pré-définie est la stratégie de normalisation *leftmost-innermost*. Puisque la stratégie de normalisation est pré-définie dans ELAN, elle n'est pas spécifiée dans la théorie de réécriture de l'utilisateur, l'ensemble de règles non-nommées doit être confluent et terminant.
  - L'ensemble de règles nommées, qui n'est pas nécessairement confluent et terminant, peut être contrôlé par des *stratégies élémentaires*. Les deux raisons principales pour leur utilisation sont
    - si l'ensemble de règles n'est pas terminant, l'utilisateur a la possibilité de restreindre l'ensemble de dérivations à un sous-ensemble de dérivations finies, afin d'éviter des dérivations infinies ;
    - si l'ensemble de règles n'est pas confluent, l'utilisateur a la possibilité de spécifier certains sous-ensembles de toutes les dérivations possibles et obtenir ainsi un sous-ensemble de tous les résultats possibles.
- Les stratégies sont utilisées en ELAN de trois façons différentes
  - pour séparer dans un programme la partie calcul de la partie contrôle ;
  - pour exprimer des dérivations non-déterministes et concurrentes ;
  - pour spécifier des procédures de normalisation particulières.

Un des avantages de la séparation entre le calcul et le contrôle est mise en valeur surtout au niveau de la compréhension des programmes. Cela veut dire que la partie du calcul est souvent technique et difficile à présenter en détail, tandis que le contrôle, exprimé en utilisant les constructions de concaténation, d'itération et de choix, est souvent plus facile à comprendre.

Parmi plusieurs caractéristiques, le calcul non-déterministe d'ELAN le différencie d'autres systèmes basés sur la réécriture. L'avantage de cette option est que cela permet de travailler avec des systèmes de réécriture non-confluents.

Le style de programmation en ELAN, basé sur le paradigme des systèmes de calcul, unifie certaines caractéristiques de la programmation fonctionnelle et logique. La programmation par réécriture est similaire à l'approche fonctionnelle restreinte au premier ordre. Cependant, la possibilité de spécifier des sous-ensembles de dérivations par un langage de stratégies joue le rôle du non-déterminisme de la programmation logique.

La variété des applications qui ont été implantées en ELAN illustre la généralité du paradigme des systèmes de calcul et montre l'expressivité et la puissance du langage comme un outil de programmation. Parmi elles, on peut citer

- une implantation de la procédure de résolution de contraintes d'ordre pour la preuve de terminaison basée sur l'ordre général sur les chemins [49] ;

- deux implantations de la procédure de complétion de Knuth-Bendix [68, 65] ;
- une implantation du prouveur de prédicats B [30] ;
- la combinaison d'algorithmes d'unification dans des théories arbitraires [101] ;
- un algorithme d'unification d'ordre supérieur [10] ;
- la SLD-résolution [113] ;
- CLP [66] ;
- la réécriture du premier ordre [69] et d'ordre supérieur.

La première version d'ELAN avait offert un interpréteur implanté en C++ qui avait été choisi comme langage d'implantation surtout pour des raisons d'efficacité et de portabilité. De nouvelles techniques de compilation de systèmes de calcul furent étudiées et maintenant il existe un compilateur du langage implanté en Java qui permet d'utiliser des symboles associatifs-commutatifs [91, 92].

Dans le reste de cette section, nous présentons brièvement le langage ELAN. Nous illustrons la syntaxe des trois composantes d'un système de calcul : signatures, règles de réécriture et stratégies. Nous décrivons aussi informellement la sémantique opérationnelle du langage. Notre intérêt est seulement de présenter les diverses constructions du langage et d'illustrer la flexibilité pour le prototypage de solveurs de contraintes. Une description formelle et détaillée du langage est donnée dans [11], une sémantique du point de vue fonctionnelle est présentée dans [13] et tous les détails nécessaires pour l'utilisation du langage peuvent être trouvés dans [14].

2.4.1 Signatures ELAN

ELAN permet de définir des signatures multi-sortées qui sont spécifiées par un ensemble de sortes $\mathcal{S}$. L'exemple 5 présente la déclaration de cinq sortes utilisées pour traiter un problème de contraintes.

Exemple 5 (Déclaration de sortes en ELAN) *La déclaration des sortes* `int`, `var`, `term`, `bool`, *et* `constraint`, *peut être faite en* ELAN *de la manière suivante*

```
sort
      int var term bool constraint;
end
```

Une fois déclarées les sortes de la signature, on peut définir les symboles de fonctions indexés qui appartiennent à l'ensemble de symboles de fonctions $\mathcal{F}$ de la signature. Chaque symbole, défini par son profil en notation *mix-fix*, peut être décoré par des attributs sémantiques comme étant un symbole libre ou associatif-commutatif (`(AC)`) et il peut aussi être décoré par des attributes syntaxiques tels que

- sa priorité syntaxique (e.g. `pri 10`) ;
- sa visibilité dans d'autres modules (e.g. `global/local`) ;
- son associativité syntaxique par défaut à gauche (`assocLeft`) ou à droite (`assocRight`) ;
- le fait d'être synonyme avec un autre symbole (e.g. `alias`).

Dans le langage, il existe également des symboles pré-définis attachés à des procédures écrites en C++ et décorés par leurs identificateurs (e.g. `code 18`).

L'exemple 6 montre la définition de symboles de fonctions avec des attributs syntaxiques et sémantiques.

Exemple 6 (Définition de symboles de fonctions en ELAN) *À partir de la déclaration de sortes de l'exemple 5, on peut spécifier que toute constante de la sorte* `int` *est un* `term` *et que toute constante de la sorte* `var` *est aussi un* `term`. *On peut également définir que toute combinaison de deux objets de la sorte* `term` *par les symboles* `(*)` *et* `(+)` *est aussi un* `term`. *En plus, on peut spécifier des attributes syntaxiques et sémantiques. Tous cela est fait de la manière suivante*

```
operators
global

      @       : (int)          term;
      @       : (var)          term;
      @ (*) @ : (term term)    term   assocRight (AC)   pri 40;
      @ (+) @ : (term term)    term   assocRight (AC)   pri 10;

end
```

Le langage permet aussi de spécifier un ensemble de *sélecteurs* pour chaque symbole de fonction $f \in \mathcal{F}$. Soit $f : (s_1 \dots s_n) \mapsto s$ un profil du symbole f. La construction syntaxique suivante spécifie un ensemble de noms de champs optionnels $field_1 \ \dots \ field_n$ du symbole f

$$f(@, \dots, @) : (field_1 : s_1 \ \dots \ field_n : s_n) \mapsto s$$

Pour cette définition du symbole f, ELAN offre n sélecteurs $@.field_i : (s) \mapsto s_i$ et n modificateurs $@[.field_i \leftarrow @] : (s\ s_i) \mapsto s$ définis par les règles suivantes

$$\begin{array}{ll} f(t_1, \dots, t_n).field_i & \Rightarrow \ t_i \\ f(t_1, \dots, t_n)[.field_i \leftarrow t] & \Rightarrow \ f(t_1, \dots, t_{i-1}, t, t_{i+1}, \dots, t_n) \end{array}$$

Exemple 7 (Définition de symboles de fonctions avec sélecteurs) *À partir des déclarations de sortes de l'exemple 5 et des définitions de symboles de fonctions de l'exemple 6, on peut définir les symboles de fonctions nécessaires pour représenter les symboles de prédicats qui nous permettent de construire les contraintes*

```
operators
global

      @       : (bool)                          constraint;
      @  =? @ : (left: term    right: term)     constraint;
      @ !=? @ : (left: term    right: term)     constraint;
      @ >=? @ : (left: term    right: term)     constraint;
      @  >? @ : (left: term    right: term)     constraint;
      @ <=? @ : (left: term    right: term)     constraint;
      @  <? @ : (left: term    right: term)     constraint;

end
```

Ainsi, on peut utiliser les sélecteurs pour obtenir, par exemple, le membre gauche de la contrainte d'inégalité ci-dessous

```
c1 = 2 (*) 3 <=? 5 (+) 3
```

par

```
c1.left
```

ce qui nous donne le term `2 (*) 3` *de sorte* `term`. *On peut aussi échanger les membres gauche et droite de la contrainte d'égalité suivante*

```
c2 = 2 (*) 3 =? 5 (+) 3
```

par

```
c2[.left <- c2.right][.right <- c2.left]
```

ce qui nous donne comme résultat la contrainte suivante

```
c2 = 5 (+) 3 =? 2 (*) 3
```

toujours de sorte `constraint`.

2.4.2 Règles de réécriture

Il existe deux types de règles souvent introduites dans une théorie de réécriture: les règles non-conditionnelles et les règles conditionnelles. La toute première version du langage ELAN introduit en plus la notion d'*affectation locale* pour des variables non-instanciées pendant le filtrage [64]. Cela nous permet d'appliquer une stratégie sur un terme autre que celui de tête et aussi de garder la valeur d'une variable lorsqu'elle est utilisée plusieurs fois dans une règle.

La syntaxe des règles conditionnelles avec des affectations locales est la suivante

$$[\ell] \quad l \Rightarrow \quad r \quad \text{if-where}$$

où

- $\ell \in \mathcal{L}$ est l'étiquette de la règle (qui est vide dans le cas d'une règle non-nommée) ;
- l et r sont des termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ représentant les membres gauche et droit de la règle ;
- if-where ::= $\{$**if** v | **where** $y := (S)u$ | **where** $y := ()u\}^*$ où
  - **if** v est une condition booléenne ;
  - **where** $y := (S)u$ est une affectation de la variable $y \in \mathcal{X}$ par le résultat de l'application de la stratégie S sur le terme $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$;
  - **where** $y := ()u$ est une affectation de la variable $y \in \mathcal{X}$ par le résultat de la normalisation du terme u.

Exemple 8 (Définition de règles en ELAN) *À partir des sortes et des symboles de fonctions définis dans les exemples 5 et 6, respectivement, la règle* `AddCoefficients` *ci-dessous définit la simplification d'un terme.*

```
rules for term
        x          : var;
        v, v1, v2  : int;
        t          : term;
global

[AddCoefficients]
        t (+) v1 (*) x (+) v2 (*) x
        =>
        t (+) v (*) x
        where   v := () v1 + v2
end
```

La variable v est affectée par le résultat de la normalisation appliquée au terme `v1 + v2` *laquelle dans ce cas utilise la définition du symbole* `+` *qui se trouve dans un module, inclut dans le langage, définissant les entiers.*

Pour des raisons de confort et d'efficacité d'exécution, le langage dispose de trois extensions pour la construction de règles : les *affectations généralisées*, la *factorisation de règles* et une construction de *règles avec des conditions structurées.*

Affectation généralisée L'*affectation généralisée* est une construction syntaxique

$$\textbf{where}\ \ (sort)\ \ p{:=}(S)u$$

où p est un terme non-clos de sorte $sort \in \mathcal{S}$.

Le terme p, dit *motif*, est composé de constructeurs et de variables, où un constructeur est un symbole de fonction qui n'apparaît pas comme opérateur de tête dans un membre gauche d'une règle de réécriture.

Toutes les variables dans le motif p non encore instanciées, sont instanciées par le filtrage de ce motif p avec le résultat de l'application de la stratégie S au terme u, ou au cas où une stratégie S n'est pas spécifiée, le résultat de la normalisation du terme u.

Factorisation de règles de réécriture La construction de factorisation permet de mettre en facteur des parties communes de plusieurs règles ayant les mêmes membres gauche et droit. Cela permet, d'une part, de supprimer une ou plusieurs règles de réécriture, et d'autre part, d'éviter le filtrage et des exécutions communes dans plusieurs règles.

En général, la syntaxe des règles est la suivante

règle ::= $l \Rightarrow r$
if-where-choose

if-where-choose ::= {if-where|
choose
{**try** if-where-choose}$^+$
end}*

Cette construction de factorisation peut être enchaînée au même niveau que des conditions **if** et des affectations locales **where**, mais également, elle peut être imbriquée.

Règle avec une condition structurée La construction d'une *règle avec une condition structurée* généralise la condition exprimée par **if** v sous la forme suivante

[ℓ] l $\Rightarrow$ if-where-choose
switch
case c_1 **then** r_1 if-where-choose$_1$
... ...
case c_n **then** r_n if-where-choose$_n$
otherwise r_0 if-where-choose$_0$
end

En général, cette construction de règles avec des conditions structurées est récursive et permet d'imbriquer des conditions structurées.

condition-structurée ::= *r* if-where-choose |
if-where-choose
switch
{**case** condition **then** condition-structurée }$^+$
otherwise condition-structurée
end

2.4.3 Stratégies élémentaires d'ELAN

Le langage de *stratégies élémentaires* d'ELAN permet de contrôler l'application des règles nommées, de définir des exécutions non-déterministes et de spécifier des dérivations simultanées. Le langage permet aussi d'introduire le typage de stratégies construites à partir de règles de réécriture. Le typage de stratégies permet de vérifier si leur usage est correct. Une stratégie de sorte $\langle s \mapsto s \rangle$ est bien typée, si toutes ses sous-stratégies sont de la même sorte $\langle s \mapsto s \rangle$ et également, toutes les règles dans cette stratégie ont des membres gauches et droits de sorte s.

En général, la sorte $\langle s_1 \mapsto s_2 \rangle$ d'une stratégie élémentaire exprime le fait que cette stratégie, appliquée à un terme de sorte s_1, donne des résultats de sorte s_2.

Le langage de stratégies élémentaires offre

- une construction pour la concaténation de stratégies : « ; » ;
- six constructions de choix : **dk**, **dk one**, **dc**, **dc one**, **first** et **first one** ;
- quatre constructions d'itération : **repeat***, **repeat+**, **iterate*** et **iterate+** ;
- une construction de normalisation : **normalise** ;
- deux constructions pour les stratégies identité et échec : **id** et **fail**, respectivement.

La syntaxe et la sémantique opérationnelle des stratégies élémentaires, de façon informelle, sont les suivantes.

- **Construction de concaténation**

  ; La concaténation de stratégies S_1 ; S_2 correspond à l'axiome de transitivité de la logique de réécriture. Pour typer une concaténation, il faut que les deux stratégies S_1 et S_2 soient de la même sorte, qui devient également la sorte de cette concaténation.

- **Constructions de choix**

  dk La stratégie **dk**(S_1 , ... , S_n) donne tous les résultats de l'application de toutes les stratégies $S_1, \ldots, S_n$. Si toutes les stratégies S_i échouent alors la stratégie **dk** échoue.

  dk one La stratégie **dk one**(S_1 , ... , S_n) donne le premier résultat de l'application de chaque stratégie $S_1, \ldots, S_n$. Si toutes les stratégies S_i échouent alors la stratégie **dk one** échoue.

  dc La stratégie **dc**(S_1 , ... , S_n) donne tous les résultats de l'application d'une des stratégies $S_1, \ldots, S_n$ laquelle est choisie de manière aléatoire. Si toutes les stratégies S_i échouent alors la stratégie **dc** échoue.

  dc one La stratégie **dc one**(S_1 , ... , S_n) donne le premier résultat de l'application d'une des stratégies $S_1, \ldots, S_n$ laquelle est choisie de manière aléatoire. Si toutes les stratégies S_i échouent alors la stratégie **dc one** échoue.

first La stratégie **first**(S_1 , ... , S_n) donne tous les résultats de l'application de la première stratégie $S_1, \ldots, S_n$ qui est applicable (en ordre textuel). Si toutes les stratégies S_i échouent alors la stratégie **first** échoue.

first one La stratégie **first one**(S_1 , ... , S_n) donne le premier résultat de l'application de la première stratégie $S_1, \ldots, S_n$ qui est applicable (en ordre textuel). Si toutes les stratégies S_i échouent alors la stratégie **first one** échoue.

- **Constructions d'itération**

repeat* La stratégie **repeat** $*(S)$ correspond à $S^i = \overbrace{S \; ; \; \ldots \; ; \; S}^{i}$ si S^{i+1} n'est plus applicable. Si la stratégie S échoue alors la stratégie **repeat*** retourne le terme d'entrée, elle n'échoue jamais.

repeat+ La stratégie **repeat** $+(S)$ correspond à la concaténation S ; **repeat** $*(S)$. Si la stratégie S échoue alors la stratégie **repeat+** aussi échoue.

iterate* La stratégie **iterate** $*(S)$ correspond à **dk**(**id** , S , S ; S , S ; S ; S , ...). Si la stratégie S échoue alors la stratégie **iterate*** retourne le terme d'entrée, elle n'échoue jamais.

iterate+ La stratégie **iterate** $+(S)$ correspond à la concaténation S ; **iterate** $*(S)$. Si la stratégie S échoue alors la stratégie **iterate+** aussi échoue.

- **Construction de normalisation**

normalise La stratégie **normalise**(S) normalise un terme par rapport à une sous-stratégie S. Nous l'expliquons plus précisément par la suite.

- **Constructions d'identité et d'échec**

id La stratégie **id** correspond à l'identité, elle retourne le même terme d'entrée et pourtant elle peut toujours être appliquée.

fail La stratégie **fail** correspond à un échec, elle échoue toujours.

L'exemple 9 montre la définition d'une stratégie en ELAN.

Exemple 9 (Définition de stratégies en ELAN) *En utilisant la règle* `AddCoefficients` *définie dans l'exemple 8, la sous-stratégie* `first one(AddCoefficients)` *retourne le premier résultat de l'application de cette règle (si elle est applicable). La stratégie* `ReduceCoefficients`*, en utilisant l'opérateur* `repeat*`*, applique cette sous-stratégie jusqu'à ce qu'elle ne puisse plus être appliquée.*

```
stratop
global
      ReduceCoefficients : < term -> term >     bs;
end

strategies for term
implicit

[] ReduceCoefficients =>
   repeat* (first one (AddCoefficients))
end
```

Le langage ELAN utilise implicitement une stratégie de normalisation avec l'ensemble de toutes les règles non-nommées. Cette normalisation se déclenche automatiquement après chaque application d'une règle de réécriture. Parfois, l'utilisateur peut avoir besoin de déclencher explicitement une normalisation particulière définie par un sous-ensemble de règles de réécriture ou il peut avoir besoin d'utiliser plusieurs types de normalisations dans le même système de réécriture.

Le langage offre la stratégie **normalise** (et son synonyme **normalize**) comme une stratégie élémentaire. Cette stratégie permet de déclencher explicitement une normalisation de type *leftmost-innermost* (par défaut) spécifiée par un ensemble de sous-stratégies, éventuellement, de règles nommées. Cette stratégie parcourt un terme et applique les sous-stratégies à tous les sous-termes autant que c'est possible.

La syntaxe générale de la stratégie **normalise** est la suivante :

$$\textbf{normalise}(S_1 : s_1 \ , \ \ldots \ , \ S_n : s_n)$$

où $S_1, \ldots, S_n$ sont des sous-stratégies de sortes $\langle s_1 \mapsto s_1 \rangle, \ldots, \langle s_n \mapsto s_n \rangle$, respectivement.

Supposons que $\langle s \mapsto s \rangle$ soit la sorte de la stratégie $S = \textbf{normalise}(S_1 : s_1 \ , \ \ldots \ , \ S_n : s_n)$. La description informelle de la stratégie S est la suivante : la stratégie S appliquée à un terme t de sorte s cherche une forme normale du terme t par rapport aux sous-stratégies $S_1, \ldots, S_n$. Ce processus de normalisation itère l'application d'une sous-stratégie S_i aux radicaux, i.e., aux sous-termes réductibles, de sorte s_i du terme t jusqu'à ce qu'il ne contienne plus que des radicaux irréductibles.

L'ordre, dans lequel les radicaux du terme t sont réduits, peut être important selon les stratégies S_i. Par défaut, la stratégie **normalise** utilise l'ordre *leftmost-innermost* qui peut être modifié par des stratégies d'évaluation attachées aux symboles de fonctions. Par exemple, il est possible de paramétrer la stratégie **normalise** pour obtenir une version *leftmost-outermost.*

Il existe également une abréviation syntaxique **normalise**(S) d'une construction générale **normalise**$(S : s)$, si $n = 1$ et $s_1 = s$. Dans ce cas, la stratégie **normalise**(S) de sorte $\langle s \mapsto s \rangle$ applique aux radicaux de sorte s du terme t la sous-stratégie S de la même sorte $\langle s \mapsto s \rangle$.

Exemple 10 (Utilisation de la stratégie normalise) *Le même effet de la stratégie définie dans l'exemple 9,* `ReduceCoefficients`*, peut être obtenu en utilisant la stratégie* **normalise**.

```
stratop
global
       SimplifyTerms : < term -> term >       bs;
end

strategies for term
implicit

[] SimplifyTerms =>
   normalise (first one (AddCoefficients))
end
```

2.4.4 Modularité, visibilité et encapsulation

Un programme ELAN peut être composé de plusieurs modules paramétrés qui définissent des sortes, des symboles de fonctions, des règles et des stratégies. Dans le système de calcul utilisé dans l'évaluation d'un programme ELAN, toutes ces définitions sont mises à plat et la structure de modules disparaît. Cependant, pour des applications de taille moyenne, il est souhaitable

d'avoir la possibilité de restreindre la visibilité de certains objets définis dans un module ou bien de paramétrer certains objets de caractère général.

Toutes les règles de réécriture et les stratégies des modules d'un programme ELAN sont dans le système de calcul mises ensemble au même niveau. De ce point de vue, la possibilité d'organiser un programme en modules et d'encapsuler certains objets dans ces modules est une facilité syntaxique pour spécifier des systèmes de façon modulaire.

Dans ELAN, les déclarations de visibilité et d'encapsulation peuvent être faites à deux niveaux

- dans l'importation de modules (`local` ou `global`) ;
- dans la déclaration d'objets dans un module (`local` ou `global`).

Étant donnée la définition d'un symbole de fonction, d'une règle ou d'une stratégie O dans un module A

- si O est défini comme étant `local` dans le module A, alors O n'est pas visible dans un module B, quel que soit le type d'importation du module A dans le module B ;
- si O est défini comme étant `global` dans le module A et le type d'importation du module A dans un module B est `local`, alors O est vu comme étant défini localement dans le module B ;
- si O est défini comme étant `global` dans le module A et le type d'importation du module A dans un module B est `global`, alors O est vu comme étant défini globalement dans le module B.

Exemple 11 (Modularité en ELAN) *La définition de sortes de l'exemple 5, peuvent être faite d'une manière paramétrée de la façon suivante*

```
module GenericSignature[Type]

sort
      Type var term;
end

operators
global

       @ : (Type)   term;
       @ : (var)    term;

end
```

Ainsi, le module `GenericSignature` *peut être invoqué avec comme paramètre une instance spécifique de la sorte* `Type`

```
module IntegerSignature

import
global
      GenericSignature[int];
end

end
```

De cette façon, dans le module `IntegerSignature` *on connaît les sortes* `var` *et* `term`*, ainsi que toute constante de sorte* `var` *est aussi de sorte* `term` *et que toute constante de sorte* `int` *et aussi de sorte* `term`*.*

Chapitre 3

Problèmes de satisfaction de contraintes

Sommaire

Le problème de trouver une affectation à des variables de façon telle qu'un ensemble de contraintes soit satisfait est largement présent dans le développement des mathématiques. Le nom de *problème de satisfaction de contraintes* est apparu dans la communauté IA vers la fin des années soixante. Pour éviter l'explosion combinatoire lors d'une énumération exhaustive de toutes les combinaisons de valeurs possibles pour les variables, la notion de réduction du problème fut introduit. Pour des raisons de simplicité, les premières études furent centrées sur des problèmes dont les contraintes portent au plus sur deux variables. Depuis, un grand nombre de techniques de réduction du problème a été développé. Ces techniques ont été combinées avec le

schéma d'énumération exhaustive donnant ainsi origine aux techniques hybrides. Les techniques de résolution de CSPs, restreintes tout d'abord à des contraintes binaires, ont été étendues plus tard au cas général de contraintes contenant un nombre quelconque de variables.

Dans ce chapitre, nous commençons par donner une définition formelle d'un CSP. Nous présentons ensuite les outils qui sont utilisés pour la représentation graphique d'un CSP et nous rappelons les techniques de résolution d'un problème de satisfaction de contraintes. À partir de l'analyse de ces techniques, nous proposons un ensemble de cinq règles de réécriture et des stratégies qui décrivent les heuristiques de résolution de CSPs. Le système de calcul ainsi construit nous permet de traiter des conjonctions de contraintes élémentaires. On présente aussi le traitement d'un cas particulier de CSP : les CSPs décomposables. Nous présentons aussi quelques considérations d'implantation concernant l'utilisation de stratégies paramétrées permettant le prototypage d'heuristiques d'une manière plus flexible et l'utilisation d'opérateurs adéquats pour mieux gérer la pose de points de choix au cours d'une énumération. Finalement, afin de montrer le fonctionnement de notre système, nous illustrons la résolution d'un problème très simple de calcul de calendriers.

3.1 Définitions

Dans cette section, on présente une définition formelle du CSP basée sur les concepts de l'algèbre universelle. Même si la définition d'un CSP comme étant un triplet $\langle X, D, C\rangle$ est largement utilisée dans la littérature, la formalisation que l'on présente ici facilite sa compréhension d'un point de vue conceptuel et nous permet d'appliquer directement les concepts des systèmes de calcul.

Définition 15 (CSP) *Une* contrainte élémentaire $c^?$ *est une formule atomique construite à partir d'une signature* $\Sigma = (\mathcal{F}, \mathcal{P})$ *et d'un ensemble dénombrable de symboles de variables* $\mathcal{X}$. *Les contraintes élémentaires peuvent être combinées avec le connecteur de conjonction* $\wedge$. *On désigne l'ensemble de contraintes formées à partir de* Σ *et* $\mathcal{X}$ *par* $\mathcal{C}(\Sigma, \mathcal{X})$. *Étant donnés une signature* $\Sigma = (\mathcal{F}, \mathcal{P})$, *un ensemble de symboles de variables* $\mathcal{X}$ *et une structure* $\mathcal{D} = (D, I)$, *un* $\langle \Sigma, \mathcal{X}, \mathcal{D}\rangle$*-CSP est une contrainte* $C = (c_1^? \wedge \ldots \wedge c_n^?)$ *tel que* $c_i^? \in \mathcal{C}(\Sigma, \mathcal{X}); \forall i = [1, \ldots, n]$[6].

On désigne la contrainte C par $C = (c_1^? \wedge \ldots \wedge c_n^?)$ ou bien par $C = \{c_1^?, \ldots, c_n^?\}$. Le nombre de contraintes dans le problème est désigné par n ($n = \mathcal{C}ard(C)$). L'ensemble des variables libres dans une contrainte $c^?$ est désigné par $\mathcal{V}ar(c^?)$; elles représentent les variables contraintes par $c^?$. L'arité d'une contrainte $c^?$ est définie comme le nombre de variables libres apparaissant dans la contrainte

$$\text{arité}(c^?) = \mathcal{C}ard(\mathcal{V}ar(c^?))$$

De cette façon on travaille avec un ensemble de contraintes indexées $C = \bigcup_{i \geq 0} C_i$, où C_i est l'ensemble de toutes les contraintes d'arité i. On désigne par e le nombre de variables qui apparaissent dans l'ensemble de contraintes ($e = \mathcal{C}ard(\mathcal{V}ar(C))$). Finalement, on désigne par a la cardinalité du domaine D ($a = \mathcal{C}ard(D)$).

Dans ce travail, nous considérerons des CSPs dans lesquels le domaine de la structure est de cardinalité finie. Ce type particulier de CSP est appelé CSP sur des domaines finis.

6. Pour des raisons de clarté, dans ce travail les contraintes son distinguées syntaxiquement des formules par un point d'interrogation sur leur symbole de prédicat. Le symbole ? explicite le fait que l'on cherche les solutions de la formule c.

Définition 16 (Solution d'un CSP) *Une solution d'une contrainte $c^?$ est une application α de $\mathcal{X}$ vers D qui associe à chaque variable $x \in \mathcal{X}$ un élément dans D tel que $\alpha(c^?)$ est vraie dans $\mathcal{D}$. Une contrainte est satisfaisable dans D si elle a au moins une solution dans D. L'ensemble de toutes les solutions de $c^?$ est défini comme suit*

$$Sol_{\mathcal{D}}(c^?) = \{\alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} | \alpha(c^?) = \mathbf{V}\}$$

Une solution dans $\mathcal{D}$ d'un ensemble de contraintes C est une solution de toutes les contraintes $c_i^? \in C$. L'ensemble de toutes les solutions de C est défini par

$$Sol_{\mathcal{D}}(C) = \bigcap_{c_i^? \in C} Sol_{\mathcal{D}}(c_i^?)$$

Comme une solution est une affectation des variables, les contraintes jouent le rôle de filtres pour les combinaisons d'instanciations des variables qui apparaissent dans les contraintes.

Les tâches typiques concernant les CSPs sont de trouver une ou toutes les solutions ou bien de déterminer l'insatisfaisabilité de l'ensemble de contraintes. Dans la section 3.3, on présente des techniques pour répondre à ces questions.

Finalement, la définition suivante nous permettra de conceptualiser d'une manière claire et simple la réduction de l'ensemble de valeurs prises par les variables.

Définition 17 (Contrainte d'appartenance) *Étant donnés une variable $x \in \mathcal{X}$ et un ensemble non-vide $D_x \subseteq D$, la* contrainte d'appartenance *de x est définie par $x \in^? D_x$. Un $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP C' avec* contraintes d'appartenance *est $C' = C \bigcup \{x \in D_x\}_{x \in^? \mathcal{X}}$, où C est un $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP.*

Nous utiliserons ces contraintes d'appartenance pour stocker dynamiquement l'ensemble des solutions d'un CSP et pour expliciter la réduction de l'ensemble des valeurs possibles pour les variables réalisée pendant le processus de résolution. En pratique, les ensembles $D_{x_{x \in \mathcal{X}}}$ seront initialisés à D au début du processus de résolution du problème et ils seront éventuellement réduits durant ce processus.

En ajoutant ces contraintes d'appartenance, on rend explicite, d'un point de vue conceptuel, la vérification de consistance. Dans la littérature standard on utilise le terme *réduction du domaine* pour désigner l'élimination de valeurs pour les variables qui sont inconsistantes par rapport à une certaine contrainte. Puisque le domaine est fixé lors de l'interprétation, le terme réduction du domaine n'est pas approprié. En utilisant les contraintes d'appartenance le domaine reste inchangé et ce sont les ensembles associés à chaque variable à travers les contraintes d'appartenance qui changent durant le processus de résolution des contraintes.

Comme l'ensemble D_x, associé par la contrainte d'appartenance à la variable x du problème, stocke toutes les valeurs potentielles prises par la variable, on peut définir l'espace de recherche d'un CSP à partir de ces contraintes d'appartenance.

Définition 18 (Espace de recherche) *Étant donné $C \bigcup \{x \in D_x\}_{x \in^? \mathcal{X}}$ un CSP avec des contraintes d'appartenance, l'espace de recherche est défini comme le produit cartésien $D_{x_1} \times \ldots \times D_{x_n}$ de tous les ensembles D_{x_i}. Ainsi, la taille de l'espace de recherche est défini par $\mid D_{x_1} \mid \times \ldots \times \mid D_{x_n} \mid$ où $\mid D_{x_i} \mid$ désigne le nombre d'éléments dans D_{x_i}* [16].

L'espace de recherche d'un CSP peut être représenté par un arbre. Soit, par exemple, un CSP consistant en trois variables x, y et z, dont les contraintes d'appartenance sont, à un moment donné, $x \in \{1,2,3,4\}$, $y \in \{1,2,3\}$ et $z \in \{1,2\}$. Si l'on considère l'ordre d'énumération des variables x, y, z, l'espace de recherche correspondant est représenté dans la figure 3.1.

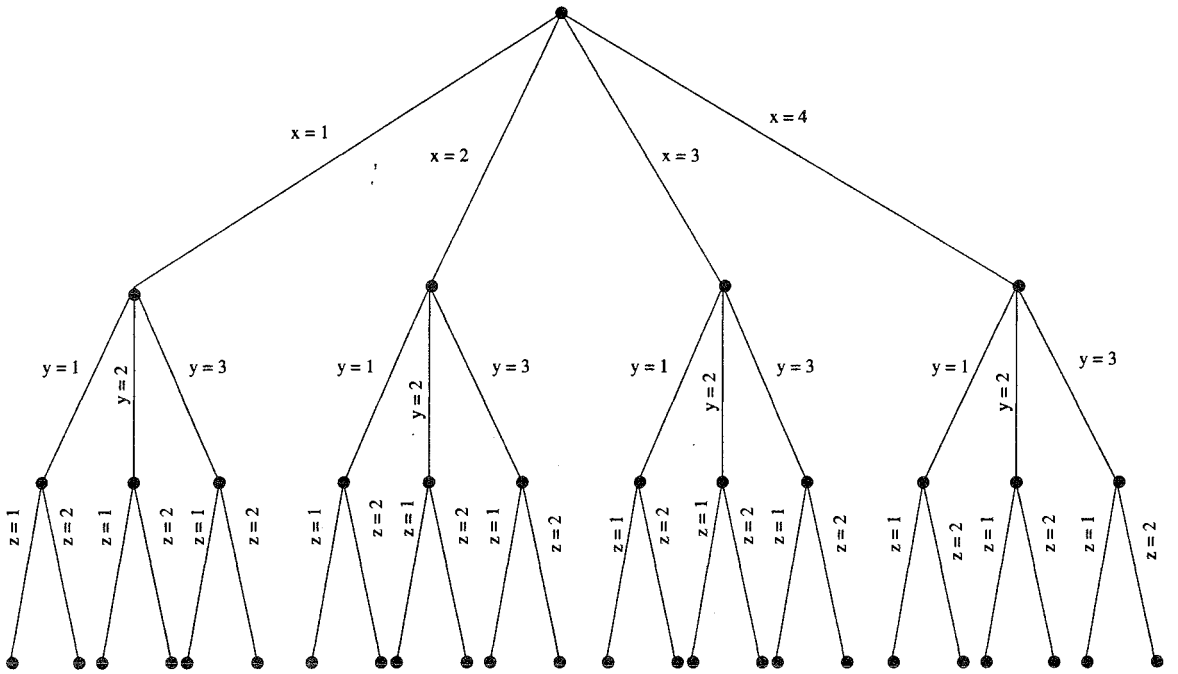

FIG. 3.1 – *Espace de recherche considérant un ordre d'énumération x, y, z*

Il est intéressant de noter que la forme de l'arbre d'énumération dépend de l'ordre d'énumération des variables. L'arbre de la figure 3.2 représente l'espace de recherche lors d'une énumération des variables dans l'ordre z, y, x.

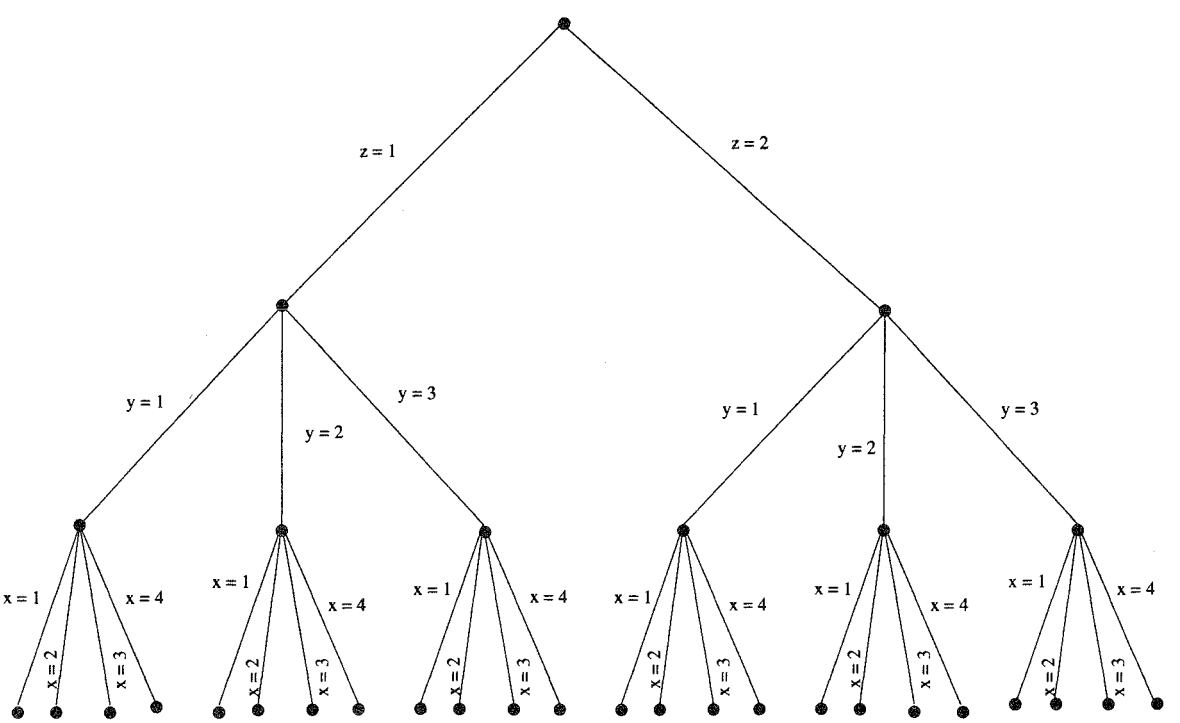

FIG. 3.2 – *Espace de recherche considérant un ordre d'énumération z, y, x*

À partir de ces deux exemples, on peut remarquer quelques propriétés de l'espace de recherche d'un CSP.

- La taille de l'espace de recherche est finie. Le nombre de feuilles de l'arbre d'énumération est

$$\mid D_{x_1} \mid \times \ldots \times \mid D_{x_n} \mid$$

Cette taille est indépendant de l'ordre d'énumération des variables. Cependant, l'ordre d'énumération des variables influe sur le nombre de nœuds internes de l'arbre d'énumération[7]. Par exemple, le nombre total de nœuds de l'espace de recherche présenté dans la figure 3.1 est 16, tandis que dans le cas de l'espace de recherche de la figure 3.2 est 8. En général, si l'on suppose que les variables sont énumérées dans l'ordre $x_1, \ldots, x_n$, le nombre de nœuds de l'espace de recherche est

$$1 + \sum_{i=1}^{n} (\Pi_{j=1}^{i} \mid D_{x_j} \mid)$$

À partir de cet expression, on peut déduire deux faits importants

  - si les variables sont énumérées en considérant la taille de leur domaines en ordre décroissant, alors le nombre de nœuds internes de l'espace de recherche sera maximal ;
  - si les variables sont énumérées en considérant la taille de leur domaines en ordre croissant, alors le nombre de nœuds internes de l'espace de recherche sera minimal.

- La profondeur de l'arbre d'énumération est égal au nombre de variables du problème, elle est donc fixe.
- Si nous fixons l'ordre d'énumération des variables, la topologie de tous les sous-arbres au même niveau de chaque branche est identique.

Les premiers travaux sur la résolution de CSP ont été réalisés dans le cas particulier des CSPs dans lesquels le domaine de la structure est un ensemble fini de valeurs et les contraintes ont des arités un ou deux uniquement. Ce type de problèmes est appelé CSP binaire. Évidemment, considérer des CSPs où au plus deux variables apparaissent dans chaque contrainte est très limité. Pour cette raison les techniques ont été étendues pour considérer le cas général où un nombre quelconque de variables intervient dans chaque contrainte.

3.2 Représentation graphique d'un CSP

Dans cette section, nous présentons les outils qui sont utilisés pour la représentation graphique des CSPs. Nous commençons par la représentation des CSPs binaires car ce type particulier de problème est à l'origine de certaines techniques de résolution de CSPs et leur nom se base sur cette représentation graphique. Nous présentons ensuite la représentation graphique d'un CSP d'arité quelconque.

7. Un nœud interne est un nœud qui a au moins un fils [1].

3.2.1 Le cas binaire

Pour la représentation graphique d'un CSP binaire, des graphes ont été utilisés [88, 76]. Pour cette raison les CSPs sont aussi dits *réseaux de contraintes.*

Un graphe G est associé à un CSP de la manière suivante

- G a un nœud pour chaque variable $x \in \mathcal{V}ar(C)$.
- Pour chaque variable $x \in \mathcal{V}ar(c^?)$ tel que $c^? \in C_1$, G a un arc qui va du nœud associé à x à lui-même. Chacun de ces arcs est étiqueté avec la contrainte associée.
- Pour chaque paire de variables $x, y \in \mathcal{V}ar(c^?)$ tel que $c^? \in C_2$, G a deux arcs orientés et opposés entre les nœuds associés à x et à y. Chacun de ces arcs est étiqueté avec la contrainte associée. La contrainte associée à l'arc (x, y) est similaire à la contrainte associée à l'arc (y, x) sauf que ses arguments sont échangés.

Le nombre de nœuds et le nombre d'arcs du graphe G est désigné par e et par n, respectivement, et *nœuds(G)* et *arcs(G)* désignent l'ensemble des nœuds et l'ensemble des arcs du graphe G, respectivement.

Exemple 12 *Soient* $\Sigma = (\{\leq, \neq\}, \mathcal{X} = \{x_1, x_2, x_3\}$, *et* $\mathcal{D} = (\{1, 2, 3, 4, 5\} \subset \mathbb{N}, \{\leq_\mathcal{D}, \neq_\mathcal{D}\})$, *où* $\leq_\mathcal{D}$ *est la relation plus petit égal et* $\neq_\mathcal{D}$ *est la relation de diségalité. Le* $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$*-CSP* $\{x_1 \leq^? 3, x_1 \neq^? x_2, x_1 \neq^? x_3, x_2 \neq^? x_3\}$ *est représenté par le graphe de la figure 3.3.*

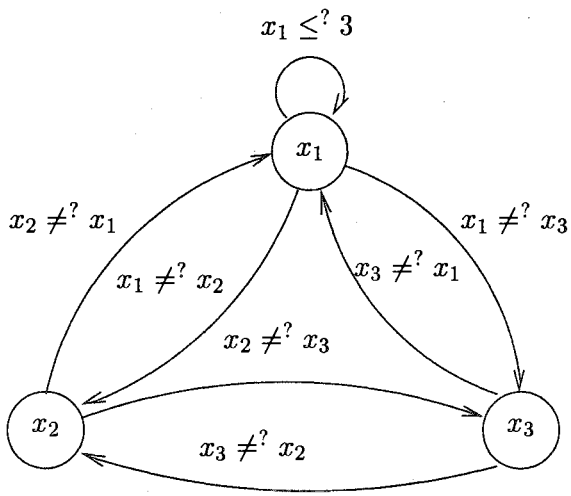

FIG. 3.3 – *Représentation par des graphes d'un CSP binaire*

3.2.2 Le cas général

En raison des limitations évidentes des graphes pour représenter des CSPs généraux, des hypergraphes avec des hyperarcs étiquetés sont utilisés pour leur représentation [89].

Définition 19 (Hypergraphe avec des hyperarcs étiquetés) *Soit* $L = \bigcup_{n>0} L_n$ *un ensemble indexé d'étiquettes. Un* hypergraphe avec des hyperarcs étiquetés $\langle N, E, k, l \rangle$ *sur* L *consiste en*

- *un ensemble fini de* nœuds *(ou* sommets*)* N *;*
- *un ensemble indexé fini d'*hyperarcs $E = \bigcup_{n>0} E_n$*, où* $\varepsilon \in E_n$ *implique que* ε *est un hyperarc lié à* n *nœuds ;*

- *une fonction indexée de connexions* $k : \bigcup_k (E_k \to N^k)$ *qui affecte une séquence de nœuds à chaque hyperarc* $\varepsilon \in E$;
- *une fonction d'étiquetage* $l : \bigcup_k (E_k \to L_k)$.

Définition 20 (Réseau de contraintes) *Un* réseau de contraintes $H = \langle N, E, k, l \rangle$ *est un hypergraphe avec des hyperarcs étiquetés, où les nœuds sont des variables, les hyperarcs sont des contraintes et la fonction d'étiquetage* l *associe à chaque* $e \in E$ *un élément* $c^? \in C$, *où* C *est l'ensemble indexé des contraintes d'un CSP.*

Exemple 13 *Soient* $\Sigma = (\{=, \leq, \neq\}, \mathcal{X} = \{x_1, x_2, x_3\}$, *et* $\mathcal{D} = (\mathbb{N}, \{=_\mathcal{D}, \leq_\mathcal{D}, \neq_\mathcal{D}\})$, *interprétés dans leur sens traditionnel dans les entiers naturels. Le* $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$*-CSP* $C = \{x_1 \leq^? 3, x_1 \neq^? x_2, x_1 \neq^? x_3, x_2 \neq^? x_3, x_1 + x_2 =^? x_3\}$ *est représenté par le réseau de contraintes de la figure 3.4.*

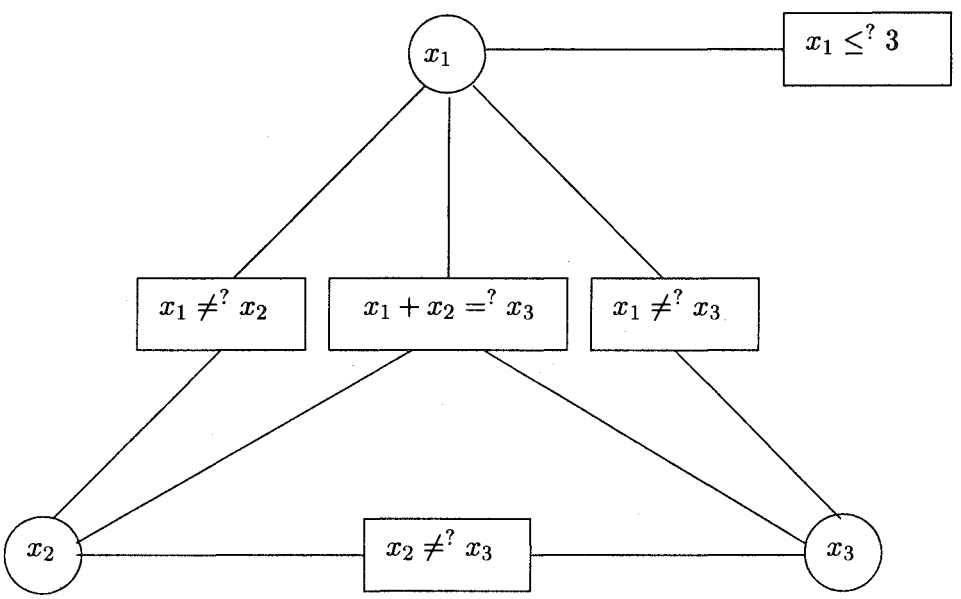

FIG. 3.4 – *Représentation par des hypergraphes d'un CSP général*

Le nombre de nœuds et le nombre d'hyperarcs de l'hypergraphe H est désigné par e et par h, respectivement, et *nœuds(H)* et *hyperarcs*(H) désignent l'ensemble des nœuds et l'ensemble des hyperarcs de l'hypergraphe H, respectivement.

3.3 Techniques de résolution

Les techniques de résolution utilisées par la communauté CSP peuvent être classées en des techniques de recherche exhaustive, des techniques de réduction de problème et des techniques hybrides. Un résumé de ces techniques se trouve dans [72].

3.3.1 Techniques de recherche exhaustive

Les techniques de recherche exhaustive consistent en l'exploration systématique de l'espace de recherche.

Générer puis tester

L'algorithme de force brute le plus simple, *générer puis tester*, connu aussi comme *recherche par essais et erreurs* [96], est basé sur l'idée de tester chaque combinaison possible de valeurs afin d'obtenir une solution à un CSP. L'algorithme instancie toutes les variables du problème en même temps et ensuite il évalue les contraintes en tenant en compte cette instanciation. Sa complexité en temps est donc $O(a^e n)$. Cette technique est basée sur une descente directe aux feuilles de l'arbre d'énumération et la vérification de la satisfaisabilité. Cet algorithme est correct mais il implique une explosion combinatoire évidente [77].

Backtracking

Afin d'améliorer le comportement naïf de *générer puis tester*, l'algorithme de recherche généralement utilisé pour résoudre des CSPs est celui dit de *Simple Backtracking*, appelé aussi *Standard Backtracking* ou *recherche en profondeur avec backtracking chronologique*, qui correspond à une heuristique de recherche générale qui a été largement utilisée dans le domaine de la résolution de problèmes[8]. Walker [114] fut le premier à établir cet algorithme dans sa forme générale. Le travail de Cohen [32] inclut une large liste de références sur les premiers travaux concernant le sujet.

Brièvement, l'algorithme instancie les variables dans un ordre prédéfini en affectant provisoirement des valeurs consistantes à une sous-séquence de variables $(x_1, \ldots, x_k)$ et en essayant d'y ajouter une nouvelle instanciation de x_{k+1} telle que l'ensemble complet de variables soit consistante. Une affectation de valeurs à un sous-ensemble de variables est consistante quand elle satisfait toutes les contraintes applicables à ce sous-ensemble.

Les contraintes qui contiennent seulement les premières k composantes sont généralement appelées *contraintes modifiées* [16]. Si l'affectation de x_{k+1} viole une contrainte, une affectation alternative de x_{k+1}, s'il est encore possible, est essayée. Si toutes les variables ont été instanciées et les contraintes sont satisfaites, le problème est résolu. Si dans une étape du processus toutes les affectations possibles d'une variable violent une contrainte, la dernière variable qui a été instanciée est revisitée, l'algorithme fait un retour en arrière à la variable la plus récente et une affectation alternative, s'il est encore possible, est essayée et le processus continue à partir de ce dernier point. Ce processus est réalisé jusqu'à ce qu'une solution soit trouvée ou bien que toutes les affectations possibles soient essayées et aient échouées.

Comme il est signalé par Bitner et Reingold, il est utile de représenter ce processus en terme du parcours d'un arbre comme celui présenté dans la figure 3.5, laquelle s'inspire de [7]. L'algorithme de *Backtracking* parcourt les nœuds suivant l'ordre montré par les lignes en pointillés : chaque branche représente l'instanciation d'une variable et chaque nœud représente le CSP obtenu à partir du CSP correspondant au nœud père plus cette instatiation. Puisque les branches de l'arbre sont parcourues d'abord en profondeur avant de retourner à explorer d'autres branches de l'arbre, ce processus est appelé *recherche en profondeur.*

On peut noter que cette technique, à la différence de *générer puis tester*, fait des calculs au niveau des nœuds internes de l'arbre d'énumération afin d'essayer d'éliminer les sous-arbres qui ne contiennent pas de solution.

Les mérites de l'algorithme de *Backtracking* sont

- son applicabilité générale ;
- sa faible demande d'espace de mémoire ;

8. Dans ce travail, nous l'appellerons simplement *Backtracking.*

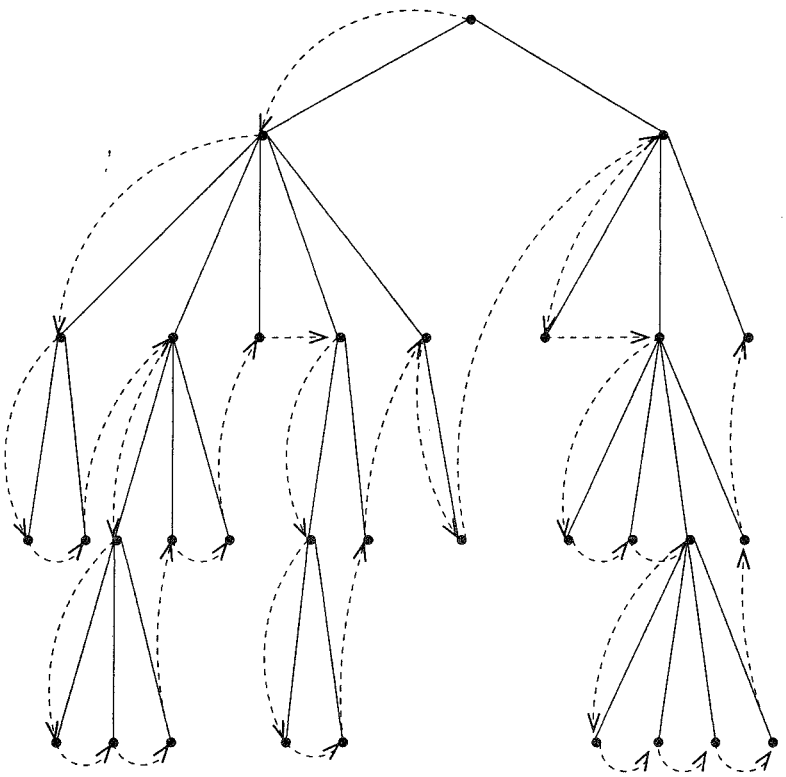

FIG. 3.5 – *Parcours d'un arbre en utilisant l'algorithme de* Backtracking

- l'élimination de sous-espaces de l'espace de recherche une fois qu'une inconsistance est détectée.

Une grande partie de l'espace de recherche que parcoure l'algorithme *générer puis tester* peut être éliminé ($| D_{x_{k+1}} | \times \ldots \times | D_{x_n} |$ combinaisons de valeurs) quand une inconsistance est détectée dans une solution partielle $x_1, \ldots, x_k$. Dans la mesure où les contraintes modifiées sont plus restrictives, l'espace de recherche sera plus petit. La meilleure contrainte modifiée est celle qui permet une seule solution partielle $(x_1, \ldots, x_k)$, laquelle peut être étendue à une solution $(x_1, \ldots, x_n)$. En général, il n'est pas possible de détecter une telle contrainte modifiée sans avoir d'abord à trouver toutes les solutions du problème considéré [16].

En général, il existe trois décisions à prendre lorsqu'on utilise l'algorithme de *Backtracking*

- la prochaine variable à instancier ;
- la valeur à affecter à la variable choisie ;
- la prochaine contrainte à analyser.

Ainsi, à partir de différents ordres d'instanciations des variables et des valeurs, des espaces de recherche différents peuvent être explorés. Dû au fait que les contraintes peuvent être propagées, les différents ordres dans lesquels les variables et leurs valeurs sont examinées peuvent influer sur l'efficacité d'un algorithme de recherche. Dans quelques problèmes, la vérification d'une contrainte peut être elle-même très chère en terme de calcul et de cette façon l'ordre dans lequel les contraintes sont examinées pourrait influer fortement sur l'efficacité d'un algorithme [107].

Néanmoins, indépendamment de l'ordre d'instanciation des variables, on peut observer presque toujours des comportements pathologiques. Bobrow et Raphael ont appelé cette classe de comportement *thrashing* [8]. Le *thrashing* peut être défini comme l'exploration répétée de sous-arbres de l'espace de recherche qui est parcouru par l'algorithme de *Backtracking* et qui sont différents seulement dans des caractéristiques non essentielles, telles que l'affectation de variables sans importance pour l'échec des sous-arbres [77]. Le pire des cas qui peut se produire dans l'algorithme de *Backtracking* est qu'une affectation soit insatisfaisable seulement à cause de l'inconsistance entre la dernière variable instanciée et les autres variables. Puisqu'il y a a^e combinaisons possibles de valeurs et que, dans le pire des cas, pour chaque combinaison toutes les contraintes doivent être vérifiées une fois, la complexité en temps de l'algorithme de *Backtracking* est $O(a^e n)$, i.e., le temps nécessaire pour trouver une solution tend a être exponentiel dans le nombre de variables [79, 107]. Cette analyse explique pourquoi Bruynooghe et Venken signalent comme le principal désavantage de *Backtracking* sa potentielle inefficacité [16]. Afin d'améliorer le comportement de ce type d'exploration, la notion de réduction de problème a été développée.

3.3.2 Techniques de réduction de problème

La réduction de problème, généralement appelée *maintien de consistance* ou *vérification de consistance* [107], consiste en des techniques pour transformer un CSP dans une représentation plus explicite. Elles sont utilisées principalement dans une phase de pré-traitement du problème afin d'améliorer la performance de la recherche exhaustive, mais elles peuvent être aussi integrées dans le propre algorithme de recherche.

L'analyse de complexité en temps de l'algorithme de *Backtracking* montre que son efficacité peut être améliorée si la taille du domaine, a, est réduite au minimum possible [79]. À partir de cette idée, les techniques de réduction de problème transforment un CSP dans un problème équivalent par la réduction de l'ensemble de valeurs prises par les variables dans le problème original. La notion de problèmes équivalents exprime le fait que deux problèmes ont un ensemble de solutions identiques.

Définition 21 (CSPs équivalents) *Étant donnés les CSPs P et P', avec les ensembles de solutions $Sol(P)$ et $Sol(P')$, respectivement, ils sont* équivalents *si $Sol(P) = Sol(P')$.*

Définition 22 (CSP réduit) *Un CSP P est* réduit *à P' si*

- *P et P' sont équivalents ;*
- *l'ensemble $D_x{}'$ associé à chaque contrainte d'appartenance $x \in^? D_x{}'$ dans P' est un sous-ensemble de l'ensemble D_x associé à chaque contrainte d'appartenance $x \in^? D_x$ dans P.*

Un problème réduit est plus facile à résoudre car les ensembles de valeurs prises par les variables dans le problème réduit ne sont pas plus grands que ceux dans le problème original. Ceci implique un nombre plus petit de combinaisons de valeurs à considérer.

Même si la simple réduction de problème ne produit pas de solutions, elle peut être extrêmement utile quand elle est utilisée avec des techniques de recherche exhaustive.

Ce type de techniques demande la capacité de reconnaître des combinaisons inconsistantes de valeurs. Plusieurs concepts de consistance ont été définis pour identifier dans l'espace de recherche des classes de combinaisons de valeurs qui ne peuvent pas apparaître simultanément dans aucun ensemble de valeurs qui satisfait l'ensemble des contraintes. Mackworth [76] propose trois niveaux de consistance : consistance de nœud, consistance d'arc et consistance de chemin. Ces noms viennent du fait que l'on utilise des graphes pour représenter les CSPs binaires [107].

Il est important de remarquer que les diverses formes des algorithmes de consistance doivent être vues comme des *algorithmes d'approximation*, dans le sens qu'elles imposent des conditions *nécessaires* mais pas toujours *suffisantes* pour l'existence d'une solution à un CSP [80].

Consistance locale dans le cas binaire

On présente dans la suite les définitions de consistance pour un réseau de contraintes binaires.

Définition 23 (Consistance de nœud) *Étant données une variable $x \in \mathcal{X}$, une contrainte d'appartenance $x \in^? D_x$ et une contrainte unaire $c^?(x) \in C$, le nœud associé à x est* consistant *(NC[9]) si*

$$\forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x \in^? D_x) \Rightarrow \alpha \in Sol_{\mathcal{D}}(c^?(x))$$

Exemple 14 *Soient $\Sigma = (\{\leq, \geq, \neq\})$, $\mathcal{X} = \{x_1, x_2\}$, et $\mathcal{D} = (\mathbb{N}, \{\leq_{\mathcal{D}}, \geq_{\mathcal{D}}, \neq_{\mathcal{D}}\})$, interprétés dans leur sens traditionnel dans les entiers naturels. Soit le $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP $C = \{x_1 \leq^? 3, x_2 \geq^? 1, x_1 \neq^? x_2\}$. Si on ajoute les contraintes d'appartenance $x_1 \in^? D_{x_1}$ et $x_2 \in^? D_{x_2}$ et ces ensembles D_{x_i} sont initialisés en $D_{x_1} = D_{x_2} = \{1, 2, 3, 4, 5\}$, le graphe qui représente ce CSP est présenté dans la figure 3.6.*

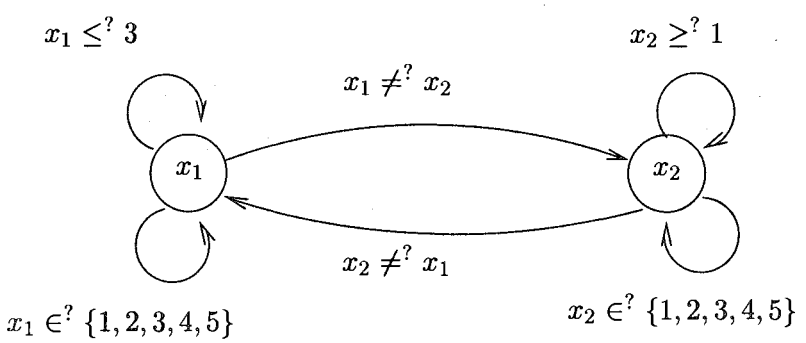

FIG. 3.6 – *Concept de consistance de nœud en CSPs binaires*

Le nœud associé à x_1 n'est pas consistant car la contrainte $x_1 \leq^? 3$ n'est pas satisfaite par toutes les affectations α qui satisfassent la contrainte $x_1 \in^? D_{x_1}$ (par exemple, $\alpha = \{x \mapsto 4\}$), mais le nœud associé à x_2 est consistant car pour toute affectation $\alpha \in Sol_{\mathcal{D}}(x_2 \in^? D_{x_2})$ la contrainte $\alpha(x_2) \geq 1$ est satisfaite.

Définition 24 (CSP consistant sur les nœuds) *Un réseau de contraintes est* consistant sur les nœuds *si tous ses nœuds sont consistants.*

Évidemment, le CSP de l'exemple 14 n'est pas consistant sur les nœuds.

La figure 3.7 présente l'algorithme NC-1 qui vérifie la consistance de nœud pour un ensemble de contraintes et qui s'inspire du travail de Mackworth [76]. On suppose qu'avant d'appliquer cet algorithme les ensembles D_x sont initialisés à D.

9. Node Consistency.

```
procedure NC-1 ;
1  debut
2    pour tout x ∈ 𝒳 faire
3      pour tout α ∈ Sol_𝒟(x ∈? D_x) faire
4        si ¬α(c?(x)) alors
5          D_x ← D_x\α(x) ;
6        fin_si
7      fin_pour
8    fin_pour
9  fin
```

FIG. 3.7 – *Algorithme NC-1*

La complexité en temps de l'algorithme NC-1 est $O(ae)$, c'est-à-dire la consistance de nœud peut être vérifiée en un temps linéaire dans le nombre de variables [79].

Définition 25 (Consistance d'arc) *Étant données les variables $x_i, x_j \in \mathcal{X}$, les contraintes d'appartenance $x_i \in^? D_{x_i}$ et $x_j \in^? D_{x_j}$ et les contraintes $c_i^?, c_j^?, c_k^? \in C$ telles que $\mathcal{V}ar(c_i^?) = \{x_i\}$, $\mathcal{V}ar(c_j^?) = \{x_j\}$ et $\mathcal{V}ar(c_k^?) = \{x_i, x_j\}$, l'arc associé à $c_k^?$ est* consistant *(AC[10]) si*

$$\forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} \quad \exists \alpha' \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i} \wedge c_i^?(x_i))$$

$$\Rightarrow \alpha' \in Sol_{\mathcal{D}}(x_j \in^? D_{x_j} \wedge c_j^?(x_j) \wedge c_k^?(\alpha(x_i), x_j))$$

Exemple 15 *Soient $\Sigma = (\{+, \neq, =\})$, $\mathcal{X} = \{x_1, x_2, x_3\}$, et $\mathcal{D} = (\mathbb{N}, \{+_{\mathcal{D}}, \neq_{\mathcal{D}}, =_{\mathcal{D}}\})$, interprétés dans leur sens traditionnel dans les entiers naturels. Soit le $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP $C = \{x_1 \neq^? x_2, x_1 + x_3 =^? 5\}$. Si on ajoute les contraintes d'appartenance $x_1 \in^? D_{x_1}, x_2 \in^? D_{x_2}$ et $x_3 \in^? D_{x_3}$ et ces ensembles D_{x_i} sont initialisés en $D_{x_1} = D_{x_2} = D_{x_3} = \{1, 2, 3, 4, 5\}$, le graphe qui réprésente ce CSP est présenté dans la figure 3.8*[11].

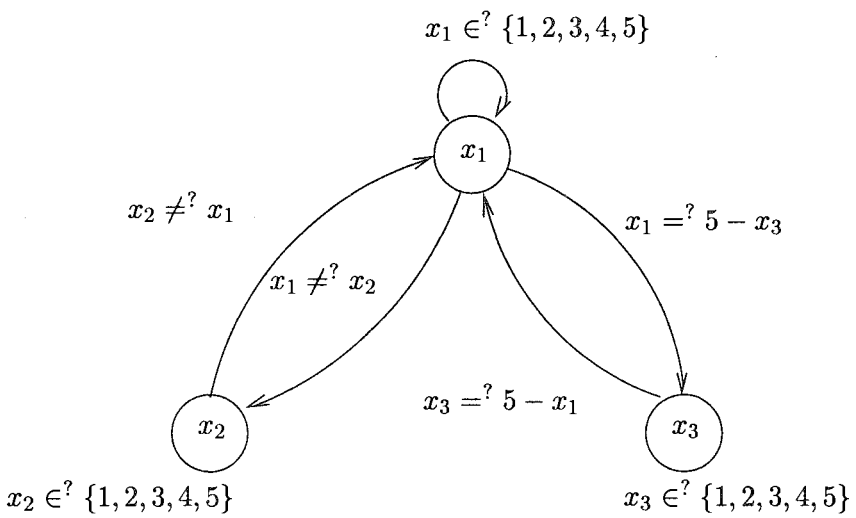

FIG. 3.8 – *Concept de consistance d'arc en CSPs binaires*

10. Arc Consistency.

11. Afin de remarquer le fait que le concept de consistance d'arc analyse les contraintes binaires dans leurs deux sens, dans la figure on exprime de manières différentes la contrainte $x_1 + x_3 =^? 5$.

L'arc (x_1, x_2) *associé à la contrainte* $x_1 \neq^? x_2$ *est consistant car pour toute affectation* $\alpha \in Sol_{\mathcal{D}}(x_1 \in^? D_{x_1})$ *il est possible de trouver une affectation* α' *qui satisfait* $\alpha(x_1) \neq^? x_2$, *mais l'arc* (x_1, x_3) *associé à la contrainte* $x_1 =^? 5 - x_3$ *n'est pas vu que pour certaines affectations* $\alpha \in Sol_{\mathcal{D}}(x_1 \in^? D_{x_1})$ *il est impossible de trouver une affectation* α' *telle que la contrainte* $\alpha(x_1) =^? 5 - x_3$ *soit satisfaite (par exemple,* $\alpha = \{x_1 \mapsto 5\}$*).*

Il est important de remarquer que la consistance d'arc est unidirectionnelle : si l'arc (i, j) est consistant cela n'implique pas que l'arc (j, i) le soit.

Exemple 16 *Dans l'exemple 15, si on initialise* $D_{x_1} = \{1, 2, 3, 4, 5\}$ *et* $D_{x_3} = \{1, 2, 3, 4\}$, *l'arc* (x_1, x_3) *est toujours inconsistant mais l'arc* (x_3, x_1) *devient consistant.*

Définition 26 (CSP consistant sur les arcs) *Un réseau de contraintes est* consistant sur les arcs *si tous ses arcs sont consistants.*

Évidemment, les deux derniers CSPs (exemples 15 et 16) ne sont pas consistants sur les arcs.

Les trois premiers algorithmes développés pour vérifier la consistance d'arc se basent sur l'opération élémentaire suivante proposée originalement par Fikes [42]. Étant données deux variables x_i et x_j et la contrainte $c^?$ telle que $\mathcal{V}ar(c^?) = \{x_i, x_j\}$, si $\alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$ et il n'existe pas d'application $\alpha' \in Sol_{\mathcal{D}}(x_j \in^? D_{x_j} \wedge c^?(\alpha(x_i), x_j))$ alors $\alpha(x_i)$ doit être éliminée de D_{x_i}. Lorsque ceci a été vérifié pour chaque $\alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$ alors l'arc (x_i, x_j) est consistant (mais cela ne veut pas dire que l'arc (x_j, x_i) est consistant). Cette idée est exprimée par la fonction *reviser* présentée dans la figure 3.9.

```
fonction reviser((x_i, x_j)) : booleen
1  debut
2    retourner ← F ;
3    pour tout α ∈ Sol_D(x_i ∈? D_{x_i}) faire
4      si Sol_D(x_j ∈? D_{x_j} ∧ c?(α(x_i), x_j)) = ∅ alors
5        D_{x_i} ← D_{x_i} \ α(x_i) ;
6        retourner ← V ;
7      fin_si
8    fin_pour
9  fin
```

FIG. 3.9 – *Fonction* reviser

La complexité en temps de la fonction *reviser* est $O(a^2)$, c'est-à-dire quadratique dans la taille de l'ensemble des valeurs prises par les variables [79].

Au moins une fois il faut appliquer la fonction *reviser* à chaque arc du graphe, mais il est évident que les applications de *reviser* aux arcs (x_j, x_k), $\forall x_k \in \mathcal{X}$, peuvent éliminer des valeurs en D_{x_j} qui sont nécessaires pour maintenir la consistance de l'arc (x_i, x_j). Appliquer une seule fois cette fonction n'est donc pas suffisant. Les trois premiers algorithmes développés pour vérifier la consistance d'arc utilisent la même fonction de base *reviser* mais ils différent dans la stratégie d'application.

La figure 3.10 présente AC-1, l'algorithme le plus simple pour vérifier la consistance d'arc, où Q correspond à l'ensemble de contraintes binaires à vérifier.

```
procedure AC-1;
1  debut
2    Q ← {(x_i, x_j) | (x_i, x_j) ∈ arcs(G), x_i ≠ x_j};
3    repeter
4      changement ← F;
5      pour tout (x_i, x_j) ∈ Q faire
6        changement ← changement ∨ reviser((x_i, x_j));
7      fin_faire
8    jusqu'a ¬changement
9  fin
```

FIG. 3.10 – *Algorithme AC-1*

L'algorithme AC-1 vérifie, en applicant la fonction *reviser*, chaque arc en une itération. Si au moins un ensemble D_x est modifié alors tous les arcs sont revisités. Ce processus est répété jusqu'à ce qu'aucun ensemble D_x soit modifié.

La complexité en temps de AC-1 est $O(a^3ne)$ [79]. Son inefficacité évidente est dûe au fait que la modification d'un seul ensemble D_x entraîne la révision de tous les arcs dans la prochaine itération, même s'ils n'ont pas de relation directe avec l'arc qui provoque la modification de D_x.

L'algorithme AC-1 peut évidemment être amélioré si après la première itération complète on revisite uniquement les arcs reliés directement aux ensembles D_x qui ont été modifiés. Cette idée fut implantée originalement par Waltz dans son algorithme de filtrage [117] et elle a servi d'inspiration pour l'algorithme AC-2 proposé par Mackworth [76]. L'algorithme AC-3 proposé aussi par Mackworth [76] utilise la même idée. La figure 3.11 présente une version de l'algorithme AC-3.

```
procedure AC-3;
1  debut
2    Q ← {(x_i, x_j) | (x_i, x_j) ∈ arcs(G), x_i ≠ x_j};
3    tantque Q ≠ ∅ faire
4      choisir et éliminer un arc (y_i, y_j) ∈ Q;
5      si reviser((y_i, y_j)) alors
6        Q ← Q ∪ {(y_k, y_i) | (y_k, y_i) ∈ arcs(G), y_k ≠ y_i, y_k ≠ y_j};
7      fin_si
8    fin_tantque
9  fin
```

FIG. 3.11 – *Algorithme AC-3*

Si l'on suppose que le graphe est connexe et la complexité de la fonction *reviser* est $O(a^2)$, la complexité en temps de AC-3 est $O(a^3n)$: la consistance d'arc peut donc être vérifiée en un temps linéaire dans le nombre de contraintes [79].

Mohr et Henderson ont proposé l'algorithme AC-4 dont la complexité est $O(a^2n)$ et ils ont prouvé qu'il est optimal en temps [83]. Le principal désavantage de cet algorithme est sa complexité moyenne qui est très proche de sa complexité dans le pire des cas. De plus, sa complexité en espace de mémoire est $O(a^2n)$, donc lorsque les problèmes ont beaucoup de solutions, c'est-à-dire lorsque les contraintes sont faibles, la consistance d'arc élimine peu ou pas de valeurs possibles, AC-3 montre des performances supérieures à celles de AC-4 malgré l'optimalité de ce

dernier [116]. Quand on fait face aux problèmes où le nombre de valeurs prises par les variables est grand et les contraintes sont faibles, AC-3 est souvent préféré par rapport à AC-4 en raison de l'utilisation de mémoire. Deux algorithmes AC-5 ont été proposés, l'un par Deville et Van Hentenryck [39] et l'autre par Perlin [97]. Ils utilisent la structure spécifique de certaines contraintes, mais dans le cas général ils ont équivalents à AC-3 ou AC-4. Bessière [5] a proposé l'algorithme AC-6 qui a la même complexité dans le pire des cas que AC-4 mais élimine le problème d'utilisation de mémoire, AC-6 a une complexité $O(an)$ en espace de mémoire. La principale limitation de AC-6 est sa complexité théorique quand il est inclut dans une procédure d'énumération. L'algorithme AC-6+, proposé aussi par Bessière, est une amélioration de AC-6 qui utilise une caractéristique directionelle des contraintes: une contrainte est bidirectionnelle si la combinaison de valeurs a pour une variable x_i et b pour une variable x_j est permise par la contrainte entre x_i et x_j si et seulement si b pour x_j et a pour x_i sont permis par la contrainte entre x_j et x_i [4]. Ce algorithme a été amélioré par Bessière et Régin avec l'algorithme AC-6++ [6]. Par coïncidence, dans le même workshop, Freuder a présenté son algorithme AC-7 qui utilise la même idée [44]. Comme notre intérêt dans ce travail est d'introduire un cadre général pour le traitement de CSPs, nous nous limitons à l'utilisation des algorithmes AC-1 et AC-3 pour expliquer notre formalisation car ils ont besoin d'une structure de données très simple.

Définition 27 (Consistance de chemin) *Étant données les variables* $x_1, \ldots, x_n \in \mathcal{X}$ *et les contraintes* $c_1^?, \ldots, c_n^?$, $c_{n+1}^?, \ldots, c_{2n}^? \in C$ *telles que* $\mathcal{V}ar(c_1^?) = \{x_1\}$,..., $\mathcal{V}ar(c_n^?) = \{x_n\}$, $\mathcal{V}ar(c_{n+1}^?) = \{x_1, x_2\}$,..., $\mathcal{V}ar(c_{2n}^?) = \{x_1, x_n\}$, *le chemin de longueur* $n-1$ *à travers les nœuds* $(x_1, \ldots, x_n)$ *est consistant (PC*[12]*) si*

$$\forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} \ \exists \alpha' \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x_1 \in^? D_{x_1} \ \wedge \ c_1^?(x_1) \wedge$$

$$x_n \in^? D_{x_n} \ \wedge \ c_n^?(x_n) \ \wedge \ c_{2n}^?(x_1, x_n))$$

$$\Rightarrow \alpha' \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i} \ \wedge \ c_i^?(x_i) \wedge$$

$$c_{n+1}^?(\alpha(x_1), x_2) \ \wedge \ c_{2n-1}^?(x_{n-1}, \alpha(x_n)) \ \wedge \ c_{n+j}^?(x_j, x_{j+1})$$

$$\forall i = 2, \ldots, n-1; j = 2, \ldots, n-2)$$

Exemple 17 *Soit* $\Sigma = (\{\neq\})$, $\mathcal{X} = \{x_1, x_2, x_3\}$, *et* $\mathcal{D} = (\mathbb{N}, \{\neq_{\mathcal{D}}\})$, *interprété dans son sens traditionnel dans les entiers naturels. Soit le* $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$*-CSP* $C = \{x_1 \neq^? x_2, x_1 \neq^? x_3, x_2 \neq^? x_3\}$. *Si on ajoute les contraintes d'appartenance* $x_1 \in^? D_{x_1}, x_2 \in^? D_{x_2}$ *et* $x_3 \in^? D_{x_3}$ *et ces ensembles* D_{x_i} *sont initialisés en* $D_{x_1} = D_{x_2} = \{1, 2\}$ *et* $D_{x_3} = \{1, 2, 3\}$, *le graphe qui représente ce CSP est présenté dans la figure 3.12*[13].

12. Path Consistency.

13. On montre seulement les arcs (i, j) pour $i < j$, les arcs opposés (j, i) ne sont pas montrés pour de raisons de clarté.

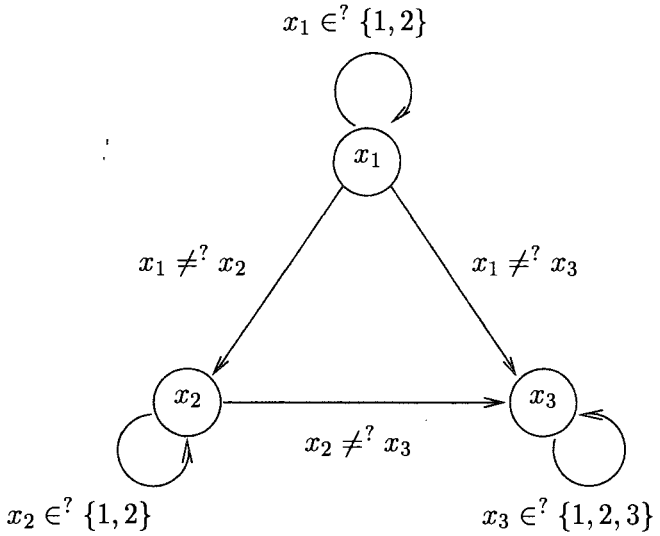

FIG. 3.12 – *Concept de consistance de chemin en CSPs binaires*

Le chemin de longueur 2 de x_1 à x_2 à travers x_3 est consistant car pour toute affectation $\alpha \in Sol_{\mathcal{D}}(x_1 \in^? D_{x_1} \wedge x_2 \in^? D_{x_2} \wedge x_1 \neq^? x_2)$ il est possible de trouver une affectation α' telle que $\alpha(x_1) \neq x_3 \wedge \alpha(x_2) \neq x_3$ soit satisfaite (dans ce cas toujours avec l'affectation $\alpha' = \{x_3 \mapsto 3\}$). Mais, le chemin de x_1 à x_3 à travers x_2 n'est pas consistant car pour certaines affectations $\alpha \in Sol_{\mathcal{D}}(x_1 \in^? D_{x_1} \wedge x_3 \in^? D_{x_3} \wedge x_1 \neq^? x_3)$ (spécifiquement $\alpha = \{x_3 \mapsto 1\}$ et $\alpha = \{x_3 \mapsto 2\}$) il n'est pas possible de trouver une l'affectation α' telle que $x_2 \in D_{x_2} \wedge \alpha(x_1) \neq x_3 \wedge x_2 \neq \alpha(x_3)$ soit satisfaite.

Il est important de noter que cette définition de consistance de chemin ne requiert pas que toutes les valeurs des variables satisfassent toutes les contraintes entre toutes les variables apparaissant dans le chemin.

Cette définition de consistance de chemin requiert que tous les nœuds $(x_1, \ldots, x_n)$ soient distincts. La définition proposée par Mackworth ne requiert pas que tous les nœuds $(x_1, \ldots, x_n)$ soient distincts, i.e., elle est applicable à des chemins simples et non simples. Ainsi, un chemin non simple pourrait être consistant quand les occurrences distinctes du même nœud dans le chemin correspondent à des instanciations différentes des variables associées. Évidemment, la différence entre les deux définitions est basée sur le fait de que la définition présentée dans ce travail instancie en même temps toutes les variables qui apparaissent dans le chemin.

Définition 28 (CSP consistant sur les chemins) *Un réseau de contraintes est* consistant sur les chemins *si pour toute paire de variables (x_i, x_j) tous les chemins entre les deux variables sont consistants.*

Plusieurs algorithmes de vérification de consistance de chemin ont été proposés. Montanari a proposé PC-1 et il a prouvé qu'un réseau de contraintes est consistant sur les chemins si chaque chemin de longueur 2 est consistant [88]. Mackworth a proposé l'algorithme PC-2 [76]. Mackworth et Freuder ont démontré plus tard que la complexité en temps de PC-1 et PC-2 est $O(a^5e^5)$ et $O(a^5e^3)$, respectivement [79]. Mohr et Henderson ont proposé l'algorithme PC-3 dont sa complexité en temps est $O(a^3e^3)$ [83]. Han et Lee [55] ont détecté une erreur dans l'algorithme PC-3 et ont proposé une solution avec l'algorithme PC-4, dont sa complexité en temps est aussi $O(a^3e^3)$. Ces deux algorithmes, PC-3 et PC-4, ont une complexité en espace de $O(a^3e^3)$.

Les concepts de consistance de nœud, consistance d'arc et consistance de chemin ont été généralisés par la définition de la k-consistance [43] : un CSP est k-consistant si pour tout sous-ensemble de k variables toutes les contraintes portant sur ces variables sont satisfaisables. Ainsi, les concepts de consistance de nœud, consistance d'arc et consistance de chemin correspondent, respectivement, à 1-consistance, 2-consistance et 3-consistance.

Consistance locale dans le cas général

Dans cette section, on presente la définition formelle de la technique de réduction de problème la plus utilisée dans le traitement de CSPs d'arité quelconque : la consistance d'arc.

Définition 29 (Consistance d'arc) *Étant données les variables* $x_1, \ldots, x_n \in \mathcal{X}$, *la contrainte* $c^?(x_1, \ldots, x_n) \in C$ *et les contraintes d'appartenance* $\bigwedge_{i=1,\ldots,n} x_i \in^? D_{x_i}$, *la contrainte* $c^?$ *est* consistant *par rapport aux ensembles* $D_{x_1}, \ldots, D_{x_n}$ *si*

$$\forall x_i \in \mathcal{X} \forall \alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha \in Sol_{\mathcal{D}}(x_i \in^? D_{x_i})$$

$$\Rightarrow \exists \alpha' \in \alpha_{\mathcal{D}}^{\mathcal{X}} : \alpha' \in Sol_{\mathcal{D}}(\bigwedge_{j=1,\ldots,n; j \neq i} x_j \in^? D_{x_j} \wedge c^?(x_1, \ldots, \alpha(x_i), \ldots, x_n))$$

Un réseau de contraintes est consistant sur les arcs *si tous ses arcs sont consistants.*

L'analyse de complexité des algorithmes de vérification de consistance montre qu'ils sont des algorithmes polynômiaux de bas degré $(1, 2, 3)$ qui permetent d'obtenir des versions réduites d'un CSP. L'ensemble de solutions du problème réduit est toujours égal à l'ensemble de solutions du CSP original. Plus d'effort est réalisé pour réduire un problème plus petit est l'espace de recherche qui est necessaire à parcourir pour trouver les solutions [80]. La conclusion générale que l'on peut tirer est qu'en réalisant une quantité limitée de calcul local, soit en un temps linéaire, quadratrique ou cubique, il est possible d'améliorer suffisamment la recherche réalisée par un algorithme du style de *Backtracking* en obtenant ainsi une amélioration substantielle dans la performance lors de la résolution de problèmes complexes. Néanmoins, il n'existe pas encore une théorie adéquate pour déterminer comment la nature particulière des contraintes influe dans la performance de ces techniques [78]. Malgré que d'un point de vue conceptuel la réduction de problème soit une technique élégante, la vérification de hauts niveaux de consistance est trop coûteuse et elle doit être combinée avec des techniques de recherche exhaustive. Pour cette raison nous nous limitons dans ce travail aux techniques de réduction de problème uniquement basées sur le concept de consistance d'arc.

3.3.3 Techniques hybrides

Nous avons vu que, d'une part, l'algorithme de *Backtracking* souffre du phénomène de *thrashing* et que d'autre part, les algorithmes de consistance peuvent être utilisés pour éliminer des inconsistances locales avant d'essayer d'obtenir une affectation complète. Du point de vue des algorithmes de vérification locale de consistance, on a intérêt à obtenir un algorithme complet, i.e., un algorithme qui donne toutes les solutions possibles du problème ; du point de vue de l'algorithme de *Backtracking*, on a intérêt à réduire le *thrashing*. Ainsi, afin de réduire le *thrashing* et d'obtenir un algorithme complet, les algorithmes de vérification de consistance présentés peuvent être intégrés dans l'algorithme de *Backtracking* : à chaque fois qu'une variable est instanciée, un nouveau problème de satisfaction de contraintes est créé ; un algorithme de

propagation de contraintes peut être appliqué pour éliminer les inconsistances locales de ces nouveaux problèmes de satisfaction de contraintes. Cette intégration des techniques de vérification de consistance dans l'algorithme de *Backtracking* a donné origine aux dites techniques hybrides.

Les techniques hybrides peuvent être décrites de la manière suivante [72].

Un nœud racine est créé pour résoudre le CSP original. À chaque fois qu'un nœud est visité, un algorithme de propagation de contraintes est utilisé pour obtenir le niveau désiré de consistance. Si dans un nœud, la cardinalité de l'ensemble de valeurs prises par chaque variable est égale à 1, et le CSP associé est consistant sur les arcs, alors le nœud représente une solution. Si durant le processus de propagation de contraintes dans un nœud, le domaine d'une des variables est vide, alors le nœud est éliminé de toute considération. Sinon, une des variables (dont la cardinalité de l'ensemble actuel de ses valeurs possibles est supérieure à 1) est choisie et un nouveau CSP est créé pour chaque affectation possible de cette variable. Chacun de ces nouveaux CSPs est dit nœud successeur du nœud représentant le CSP père (on peut noter que chaque nouveau CSP est plus petit que le CSP père car il faut choisir une assignation pour une variable de moins). L'algorithme de *Backtracking* visite ces nœuds dans la forme traditionnelle de recherche en profondeur jusqu'à ce qu'une solution est trouvée. La figure 3.13 présente graphiquement le comportement d'un algorithme hybride.

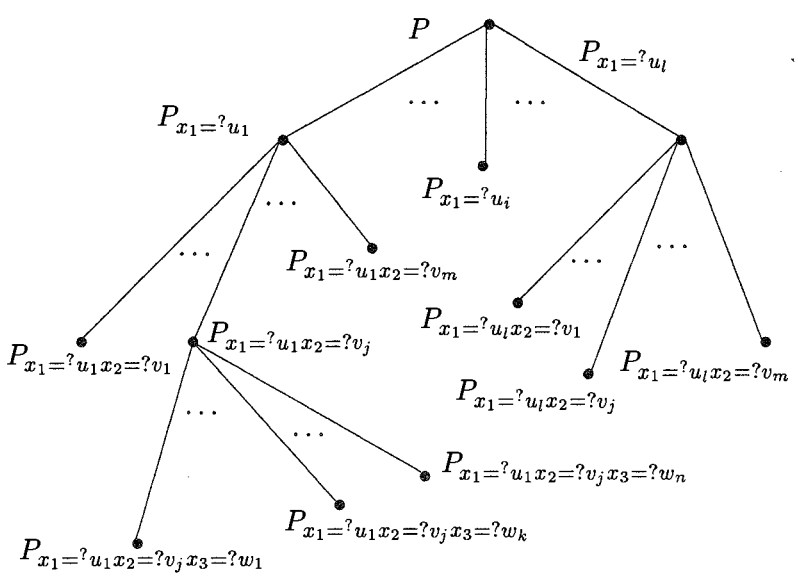

FIG. 3.13 – *Arbre d'énumération en utilisant des techniques hybrides*

On remarque que ces techniques font encore plus de calculs au niveau des nœuds internes de l'arbre d'énumération que la technique de *Backtracking.*

Un effort de recherche important a été dédié à l'étude d'algorithmes qui pour la plupart se trouvent dans le cadre général présenté pour les techniques hybrides. En particulier, Nadel [93] a comparé expérimentalement le comportement des algorithmes suivants: générer puis tester, *Simple Backtracking* , *Forward Checking*, *Partial Lookahead*, *Full Lookahead*, et *Really Full Lookahead.* La différence principale entre ces algorithmes se base sur la quantité de vérification de consistance d'arc qui est réalisée dans les nœuds de l'arbre d'énumération. Les résultats obtenus indiquent que le meilleur choix est de réaliser seulement une quantité limitée de propagation de contraintes.

Full Lookahead

L'algorithme de *Full Lookahead* représente la version la plus naïve des techniques hybrides : chaque fois qu'une variable est instanciée on vérifie la consistance de tout l'ensemble de contraintes.

Forward Checking

L'algorithme de *Forward Checking* est une version limitée de *Full Lookahead* : chaque fois qu'une variable est instanciée on vérifie la consistance de toutes les contraintes où cette variable apparaît. L'idée de cet algorithme est de vérifier seulement les valeurs des variables liées directement à la variable instanciée. Cette vérification de consistance ne prend pas en compte les autres contraintes. Si la vérification de consistance réduit l'ensemble des valeurs potentielles d'une variable x qui n'est pas liée directement à la variable instanciée, on ne vérifie pas la consistance pour les contraintes où la variable x apparaît.

Backjumping

Le comportement de l'algorithme de *Backjumping* est similaire à celui de l'algorithme de *Backtracking* sauf que le retour arrière est fait d'une manière différente. Cet algorithme essaie d'identifier les causes des échecs : lorsque l'instanciation d'une variable est impossible car elle ne satisfait pas une contrainte alors il analyse la contrainte afin de détecter les variables déjà instanciées qui participent dans la contrainte. L'algorithme réalise alors un saut dans l'arbre de recherche jusqu'à l'instanciation d'une de ces variables.

3.3.4 Critères de sélection des variables et des valeurs

Comme nous l'avons signalé dans la section 3.1, l'ordre d'énumération des variables a une influence significative quant au nombre de nœuds internes de l'arbre d'énumération. Lors de la recherche d'une solution par l'application des techniques de recherche exhaustive, l'ordre d'énumération des variables et des valeurs influe sur le nombre de retours arrière. D'ailleurs, ces ordres ont une influence lors de l'utilisation de techniques de vérification de consistance car ces techniques utilisent les instanciations partielles pour détecter des échecs et réduire ainsi l'espace de recherche. En plus, quand la vérification des contraintes est coûteuse, la simplification des contraintes résultant des instanciations partielles peut améliorer sensiblement l'efficaté des techniques de résolution de CSPs. En raison de ces considérations, plusieurs critères de sélection des variables et des valeurs ont été proposés. Dans cette section nous présentons quelques uns.

Critères de sélection des variables

Pour des raisons de simplicité, nous présentons la définition de quelques critères de sélection des variables dans le cas des CSPs binaires.

Degré avec précédence Le critère du degré avec précédence (MWO [14]) se base sur la topologie du graphe et les ordres possibles d'énumération des variables : quand le nombre de contraintes (arcs) qui contraignent une variable (nœud) est clairement supérieure au nombre de contraintes

14. Minimal Width Ordering.

qui contraignent une autre variable, ce critère propose d'énumérer d'abord la variable la plus contrainte.

Définition 30 *Étant donné un graphe G*

- *le degré avec précédence d'un nœud v par rapport à un ordre sur les éléments de* nœuds(G) *est égal au nombre de nœuds adjacents et précédents dans l'ordre à v ;*
- *le degré avec précédence d'un ordre est égal au degré maximum avec précédence de tous les nœuds par rapport à cet ordre ;*
- *le degré avec précédence d'un graphe est égal au degré minimum avec précédence de tous les ordres possibles.*

Ce critère propose donc d'énumérer les variables en considérant le degré avec précédence du graphe. L'intuition derrière ce critère est que le degré de liberté d'une variable est inversement proportionnelle au nombre de liaisons de cette variable vers des variables non-instanciées. Ainsi, les variables avec un degré de liberté majeur sont instanciée plus tard ce qui peut minimiser le nombre de retour arrière.

Le critère du degré maximum est une simplification du critère du degré avec précédence : il propose simplement d'énumérer la variable qui apparaît le plus dans l'ensemble de contraintes.

Minimum domaine Le principe général du *First Fail* propose de réaliser d'abord les tâches qui ont la plus forte probabilité d'échouer. Le critère du domaine minimum est une instance de ce principe : il propose d'énumérer la variable qui a associée le plus petit ensemble de valeurs potentielles à prendre.

Critères de sélection des valeurs

Lorsqu'on considère des domaines ordonnés, l'ordre d'instanciation des valeurs peut avoir une influence sur l'efficacité des techniques de résolution de CSPs. Néanmoins, cet influence est beaucoup moins claire que dans le cas de la sélection des variables car le nombre de nœuds internes de l'espace de recherche ne dépend pas de l'ordre d'instanciation des valeurs. Dans les systèmes que l'on propose dans ce travail, on considère simplement l'énumération des valeurs par la séparation de l'ensemble de valeurs en deux sous-ensembles. Les cas extrêmes étant : un des sous-ensembles contient uniquement l'élément le plus petit, un des sous-ensembles contient uniquement l'élément le plus grand.

3.4 Méthodologie pour la construction d'un système de calcul

Basés sur l'approche proposée par Jouannaud et Kirchner pour la résolution de problèmes d'unification, nous voyons la résolution de contraintes comme un processus de transformation d'un ensemble de contraintes jusqu'à obtenir une forme résolue à partir de laquelle une représentation de toutes les solutions peut être extraite facilement [59]. L'idée de base derrière cette approche est de faire une différence claire entre les transformations, exprimées par des règles, et leur utilisation, c'est-à-dire le contrôle. Les principales avantages d'une telle distinction sont de faciliter, d'une part, la conception et la compréhension des algorithmes et, d'autre part, la preuve de certaines propriétés. Dans ce travail, la méthodologie générale utilisée pour la construction des systèmes de calcul est la suivante.

1. On commence par définir une forme basique pour l'ensemble de contraintes. Cette forme basique représente des caractéristiques syntaxiques de l'ensemble de contraintes. En consi-

dérant toutes les contraintes qui peuvent être construites à partir de la signature considérée, nous définissons celles qui seront traitées par les transformations.

2. À partir de la forme basique, on définit une forme résolue qui représente le résultat que nous désirons obtenir. Cette forme résolue est une instance particulière de la forme basique à laquelle s'ajoute les caractéristiques sémantiques qui permettent de représenter la solution au problème. Les formes résolues caractérisent donc les ensembles de contraintes qui correspondent aux solutions du problème.
3. Nous définissons ensuite les règles qui expriment les transformations à réaliser sur l'ensemble de contraintes. Pour chaque ensemble de contraintes possible qui n'est pas en forme résolue nous concevons une règle qui transforme cet ensemble en un autre équivalent. Cette notion d'équivalence a un sens syntaxique et un autre sémantique : d'un point de vue syntaxique les règles doivent maintenir l'ensemble de contraintes dans la même forme syntaxique ; d'un point de vue sémantique les règles doivent maintenir l'ensemble de formes résolues. Ainsi, chaque règle définie doit vérifier les propriétés suivantes.

   (a) Maintenir le problème en forme basique.
   (b) Maintenir l'ensemble des solutions du problème, c'est-à-dire

$$Sol(P') = Sol(P)$$

   Cette preuve est normalement réalisée en vérifiant la correction et la complétude d'une règle.

      i. La *correction* d'une règle qui transforme un problème P en un problème P' est vérifiée si

$$Sol(P') \subseteq Sol(P)$$

      ii. La *complétude* d'une règle est vérifiée si

$$Sol(P) \subseteq Sol(P')$$

   Les propriétés de correction et de complétude d'une règle signifient que cette règle maintient l'ensemble des solutions du problème.

4. On prouve la terminaison de l'ensemble de règles définies pour une stratégie particulière ou une classe de stratégies déterminées. Autrement dit, on prouve que l'application des règles suivant une certaine stratégie termine forcément. Cette preuve se base généralement sur la définition de mesures de complexité appropriées. Si l'application des règles suivant cette stratégie spécifique fait décroître strictement ces mesures, alors le calcul termine.
5. Finalement, on prouve que les formes normales obtenues, par rapport à l'ensemble de règles définies, correspondent aux formes résolues. Cette preuve peut être réalisée en deux étapes.

   (a) On prouve que toute forme résolue est normale par rapport à l'ensemble de règles, c'est-à-dire qu'aucune règle ne peut être appliquée. Cette preuve est évidente car les règles sont définies pour transformer des ensembles de contraintes qui ne sont pas en forme résolue.

(b) On prouve que toute forme non résolue est réductible par une des règle définies. Cette preuve est moins facile car les règles sont définies justement afin de transformer tout ensemble de contraintes qui n'est pas en forme résolue en un autre équivalent. Si cette propriété n'est pas vérifiée alors il faudra considérer la définition d'une nouvelle règle qui s'occupe de ce cas particulier.

3.5 Système de calcul pour la résolution de conjonctions de contraintes élémentaires

Basé sur le travail de Comon, Dincbas, Jouannaud et Kirchner [33], nous avons proposé un système de calcul pour la résolution de CSPs binaires [20, 19]. Nous avons ensuite généralisé cette formalisation pour considérer des CSPs généraux [24]. Pour des raisons de simplicité, dans cette section nous présentons un système de calcul pour résoudre des conjonctions de contraintes élémentaires d'arité quelconque. Dans le chapitre 5, nous étendons le langage afin de considérer la combinaison de contraintes élémentaires avec les connecteurs de disjonction, d'implication et d'équivalence.

Nous commençons par définir les formes résolues qui représentent les résultats que nous désirons obtenir. Basé sur l'analyse des techniques de résolution de CSPs présentées dans la section 3.3, on propose ensuite un ensemble de cinq règles de réécriture qui expriment les transformations élémentaires réalisées par ces techniques. Finalement, nous présentons des stratégies qui nous permettent de décrire le comportement de quelques heuristiques de résolution de CSPs. On verra comment de simples modifications dans l'ordre d'application des règles nous permettent d'obtenir des différentes formes résolues ou une même forme résolue mais plus ou moins efficacement.

3.5.1 Forme résolue

La forme résolue utilisée est définie à partir de la forme basique suivante.

Définition 31 (Forme basique d'un CSP) *Une* forme basique *pour un CSP P est une conjonction de formules*

$$\bigwedge_{i\in I}(x_i \in^? D_{xi}) \wedge \bigwedge_{j\in J}(x_j =^? v_j) \wedge \bigwedge_{m\in M} c_m^?$$

équivalente à P, où $v_j \in D$ est une valeur quelconque dans le domaine d'interprétation considéré et $c_m^?$ est une contrainte élémentaire, telle que

- $\forall i,j \in I \cup J : x_i \neq x_j$
- $\forall i \in I : D_{xi} \neq \emptyset$
- $\forall m \in M : \mathcal{V}ar(c_m^?) \subseteq \{x_i | i \in I \cup J\}$

Les contraintes de la première et deuxième conjonction sont appelées contraintes d'*appartenance* et d'*égalité*, respectivement. À chaque variable est associée soit une contrainte d'appartenance soit une contrainte d'égalité. L'ensemble des valeurs associées à chaque variable dans les contraintes d'appartenance doit être non-vide et pour chaque variable apparaissant dans une contrainte de la troisième forme de conjonctions (les contraintes du problème initial) il doit y avoir une contrainte d'appartenance ou d'égalité associée. Les variables apparaissant uniquement dans des contraintes d'égalité sont appelées *variables résolues* et les autres, *variables non-résolues*.

A tout CSP P en forme basique, il est possible d'associer une *affectation basique* obtenue en affectant chaque variable x_j dans les contraintes d'égalité par la valeur associée v_j et chaque variable x_i dans les contraintes d'appartenance par une valeur quelconque dans l'ensemble des valeurs D_{x_i}.

Définition 32 (Affectation basique) *Une* affectation basique *pour un CSP P en forme basique est une application α de $\mathcal{X}$ vers D telle que*

$$\forall j \in J : \alpha(x_j) = v_j$$

$$\forall i \in I : \alpha(x_i) \in D_{x_i}$$

De la même manière que fut défini le concept de k-consistance pour des CSPs binaires, on peut définir différentes formes résolues qui prennent en compte le niveau de consistance imposé sur l'ensemble des contraintes. Ainsi, par exemple, un CSP P en *forme résolue binaire* est un système en forme basique tel que l'ensemble des contraintes est consistant sur les arcs. Un cas particulièrement important est celui où la vérification de consistance considère toutes les variables en même temps et de cette façon assure la consistance globale.

Définition 33 (Forme résolue d'un CSP) *La* forme résolue *d'un CSP P est une conjonction de formules en forme basique équivalente à P telle que toute affectation basique satisfait toutes les contraintes de P. Une affectation basique pour un CSP P en forme résolue est une* solution.

Comme les techniques de résolution de CSPs se basent sur l'énumération des solutions et non sur une représentation canonique de l'ensemble de toutes les solutions, nous définissons un cas particulier de forme résolue : la forme résolue par énumération.

Définition 34 (Forme résolue par énumération d'un CSP) *La* forme résolue par énumération *d'un CSP P est une forme résolue contenant une contrainte d'égalité pour chaque variable. Toute autre contrainte est satisfaite trivialement.*

Donc, chaque solution est représentée par une conjonction de e contraintes d'égalité, chacune associée à une variable du CSP, et de la contrainte $\mathbf{V}$ qui correspond à l'évaluation de l'ensemble de contraintes du problème original.

Dans la suite de ce travail, on suppose qu'au début du processus de résolution de contraintes on commence avec un CSP en forme basique, où une contrainte d'appartenance a été créée pour chaque variable apparaissant dans l'ensemble de contraintes initial et qu'aucune contrainte d'égalité n'est présente[15]. On utilise la constante $\mathbf{F}$ pour désigner un CSP insatisfaisable.

3.5.2 Règles de transformation

Les règles de transformation proposées peuvent être classées en

- des règles de simplification qui expriment la vérification de consistance locale ;
- des règles de propagation qui expriment la propagation des résultats obtenus lors de l'application des règles de transformation ;
- des règles d'énumération qui sont définies pour réaliser l'énumération des valeurs nécessaire pour une recherche exhaustive.

15. Ces contraintes d'égalité correspondent à celles de la forme $x_j =^? v_j$ utilisées dans la définition 31. Cela ne veut pas dire que des contraintes utilisant le prédicat d'égalité ne peuvent pas apparaître dans l'ensemble de contraintes initial.

Règles de simplification

L'idée fondamentale des algorithmes de vérification de consistance présentés dans la section 3.3 est de réduire l'ensemble de valeurs prises par les variables en préservant l'ensemble des solutions. Le pas d'inférence réalisé par les algorithmes de vérification de consistance d'arc peut être vu comme la transformation d'un état initial avec les contraintes $x_i \in^? D_{x_i}$ et $c^?$ en un état final où la contrainte $x_i \in^? D_{x_i}$ a été éliminée et une nouvelle contrainte $x_i \in^? D'_{x_i}$ a été créée, où D'_{x_i} correspond à D_{x_i} sans les éléments qui ne sont pas compatibles avec les valeurs dans D_{x_j} par rapport à $c^?$ pour toute $x_j \in \mathcal{V}ar(c); j \neq i$. Cette idée est exprimée par la règle d'inférence **ArcConsistency**, présentée dans la figure 3.14, où $RD(x_i \in^? D_{x_i}, c^?)$ désigne l'ensemble

$$D'_{x_i} = \{v_i \in D_{x_i} \mid (\exists v_1 \in D_{x_1}, \ldots, v_{i-1} \in D_{x_{i-1}}, v_{i+1} \in D_{x_{i+1}}, \ldots, v_n \in D_{x_n}) : c^?(v_1, \ldots, v_i, \ldots, v_n)\}$$

Dans ce cas on impose que $RD(x_i \in^? D_{x_i}, c^?) \neq D_{x_i}$ pour éviter des applications répétées de la même transformation. Dans cette règle, et dans les autres présentées dans le reste de ce travail, C représente tout le reste du CSP.

[ArcConsistency]
$x_i \in^? D_{x_i} \ \wedge \ c^? \ \wedge \ C$
$\Rightarrow$
$x_i \in^? RD(x_i \in^? D_{x_i}, c^?) \ \wedge \ c^? \ \wedge \ C$
if $RD(x_i \in^? D_{x_i}, c^?) \neq D_{x_i}$

FIG. 3.14 – *Règle pour l'élimination de valeurs*

On a $RD(x_i \in^? D_{x_i}, c^?) \subset D_{x_i}$ et par conséquent cette règle est correcte. Elle est aussi complète car pour toute $v_i \in D_{x_i} \backslash RD(x_i \in^? D_{x_i}, c^?)$, une application de la forme $(x_1, \ldots, x_i, \ldots, x_n) \mapsto (v_1, \ldots, v_i, \ldots, v_n)$ ne peut pas être une solution de $c^?$ par définition de $RD(x_i \in^? D_{x_i}, c^?)$.

Nous ne spécifions pas ici la façon précise par laquelle RD est calculé. Pour des contraintes sur les entiers bornés, sa définition spécifique correspond à celle des *domain reduction rules*[16] proposées par Apt dans [2]. Dans notre cas, la définition de RD correspond à celle de la fonction *revise domain* utilisée par les algorithmes de vérification de consistance d'arc, présentés dans la section 3.3.

Comme l'application de la règle **ArcConsistency** peut retourner un ensemble vide de valeurs pour une variable, on a besoin de traiter cette situation. La règle **Falsity**, présentée dans la figure 3.15, exprime cette idée.

[Falsity]
$x \in^? \emptyset \ \wedge \ C$
$\Rightarrow$
F

FIG. 3.15 – *Règle pour la détection de l'insatisfaisabilité*

La règle **Falsity** exprime évidemment l'insatisfaisabilité de l'ensemble de contraintes. Par définition, **F** représente un CSP insatisfaisable et $Sol_{\mathcal{D}}(x \in^? \emptyset \wedge C) = \emptyset$, donc cette règle est correcte et complète.

16. Règles de réduction des domaines.

Règles de propagation

Afin d'expliciter le fait que, comme résultat de l'application de la règle **ArcConsistency**, l'ensemble de valeurs prises par une variable a un seul élément, on définit la règle **Instantiation** présentée dans la figure 3.16.

[Instantiation]
$x \in^? \{v\} \ \wedge \ C$
$\Rightarrow$
$x =^? v \ \wedge \ C$

FIG. 3.16 – *Règle pour l'instanciation de variables*

La règle **Instantiation** correspond à l'affectation d'une variable. En raison de l'existence d'une seule application α satisfaisant $x \in^? \{v\}$, la contrainte d'appartenance peut être éliminée et une nouvelle contrainte d'égalité $x =^? v$ peut être ajoutée en préservant le CSP en forme basique. La règle **Instantiation** modifie seulement la représentation d'un CSP : $x \in^? \{v\}$ est équivalent à $x =^? v$, donc $Sol_{\mathcal{D}}(x \in^? \{v\} \wedge C) = Sol_{\mathcal{D}}(x =^? v \wedge C)$ et ainsi la règle est correcte et complète.

Une fois qu'une variable a été instanciée, on peut propager sa valeur à travers toutes les contraintes où la variable apparaît et de cette façon simplifier l'ensemble de contraintes. Cette propagation est exprimée par la règle **Elimination** présentée dans la figure 3.17, où $C\{x \mapsto v\}$ désigne la conjonction de contraintes obtenues à partir de C par le remplacement de toutes les occurrences de la variable x par la valeur v.

[Elimination]
$x =^? v \ \wedge \ C$
$\Rightarrow$
$x =^? v \ \wedge \ C\{x \mapsto v\}$
if $x \in Var(C)$

FIG. 3.17 – *Règle pour l'élimination de variables*

La règle **Elimination** est appliquée aux contraintes d'égalité qui peuvent être créées uniquement par l'application de la règle **Instantiation**. Vu que la règle **Instantiation** élimine la contrainte d'appartenance associée à la variable instanciée, il n'y a pas de contrainte d'appartenance associée à la variable que nous éliminons lors de l'application de la règle **Elimination**. Pour cette raison, une fois que l'on applique la règle **Elimination**, la variable devient une *variable résolue*. Par définition, x n'apparaît pas dans $C\{x \mapsto v\}$, donc toute application α qui est solution de $(x =^? v \wedge C\{x \mapsto v\})$ est telle que $\alpha(x) = v$. Si nous considérons le coté gauche de cette règle, pour toute application α qui est solution de $(x =^? v \wedge C)$ on a $\alpha(x) = v$ et donc cette règle est correcte et complète.

La règle **Elimination** réduit l'arité des contraintes où la variable apparaît : les contraintes unaires deviennent des formules closes dont les valeurs de vérité doivent être vérifiées, les contraintes binaires deviennent des contraintes unaires qui sont plus faciles à tester et ensuite de suite. Lors de l'évaluation des formules closes nous utilisons deux règles logiques additionnelles

- si l'une des formules closes est évaluée à **F** la règle **Elimination** retourne **F**, par l'application de la règle $C \wedge \mathbf{F} \Rightarrow \mathbf{F}$ (le CSP n'a pas de solution) ;
- lorsque les formules closes sont évaluées à **T** elles sont éliminées, par application de la règle $C \wedge \mathbf{T} \Rightarrow C$ et ainsi l'ensemble de contraintes C simplifié est retourné.

Il est important de remarquer la relation étroite entre les règles **Instantiation** et **Elimination**. D'un point de vue sémantique, la contrainte d'appartenance $x \in^? \{v\}$ est équivalent à la contrainte d'égalité $x =^? v$. Néanmoins, en utilisant la contrainte d'appartenance on n'explicite pas une information très utile : la variable peut prendre une seule valeur. Cette information additionnelle peut être utilisée par la règle **Elimination** pour simplifier l'ensemble de contraintes. Les avantages de cette approche ont été signalés depuis les premiers travaux sur la manipulation de formules mathématiques. Caviness mentionne que les expressions simplifiées demandent généralement moins d'espace de mémoire, leur calcul est plus rapide et plus simple et leur équivalence fonctionnelle est plus facile à identifier [28]. Par contre, il est nécessaire de signaler qu'en utilisant cette approche on perd des informations, en particulier, dans le cas de la résolution de CSPs dynamiques, car on ne sait plus où la variable apparaissait.

Règle d'énumération

L'énumération de valeurs peut être vue comme un processus de recherche dichotomique. À chaque étape l'ensemble des valeurs prises par une variable est divisé en deux sous-ensembles, ainsi on crée deux sous-problèmes. La solution du problème initial se trouvera donc dans l'union des solutions de chaque sous-problème. La règle **SplitDomain**, présentée dans la figure 3.18, exprime cette idée générale d'énumération laquelle est nécessaire pour réaliser une recherche exhaustive.

[SplitDomain]
$x \in^? D_x \ \wedge \ C$
$\Rightarrow$
$x \in^? D'_x \ \wedge \ C$ **or** $x \in^? D''_x \wedge \ C$
if $D'_x \cup D''_x = D_x$ **and** $D'_x \cap D''_x = \emptyset$ **and** $D'_x \neq \emptyset$ **and** $D''_x \neq \emptyset$

FIG. 3.18 – *Règle pour la séparation de l'ensemble de valeurs prises par une variable*

Comme la condition de la règle **SplitDomain** impose que $D'_x \cup D''_x = D_x$, cette règle n'élimine pas de solutions du problème initial, l'ensemble de solutions est seulement divisé en deux sous-ensembles, donc la règle **SplitDomain** est correcte et complète.

Lemme 1 *Les règles* **ArcConsistency**, **Falsity**, **Instantiation**, **Elimination** *et* **SplitDomain** *sont correctes et complètes.*

Preuve : Elle est directe à partir de l'examen de chaque règle.

□

Lemme 2 *L'application répétée de l'ensemble de règles* { **ArcConsistency**, **Falsity**, **Instantiation**, **Elimination**, **SplitDomain** } *termine.*

Preuve : Pour prouver la terminaison nous devons rappeler qu'au début du processus de résolution des contraintes une contrainte d'appartenance est créée par chaque variable qui apparaît dans l'ensemble de contraintes et qu'aucune contrainte d'égalité n'est présente. Nous définissons ainsi les trois mesures de complexité suivantes

- M_1, le nombre de contraintes d'appartenance ($|I|$) ;
- M_2, la cardinalité de l'espace de recherche ($| D_{x_1} | \times \ldots \times | D_{x_e} |$) ;

- M_3, la cardinalité de l'ensemble des variables qui apparaissent dans l'ensemble de contraintes $Card(Var(\bigwedge_{m \in M} c^?_m))$.

Nous pouvons maintenant analyser l'effet, sur ces mesures de complexité, de l'application de chaque règle proposée.

- $RD(x_i \in^? D_{x_i}, c^?) \subset D_{x_i}$ est la condition pour l'application de la règle **ArcConsistency**. De cette façon cette règle élimine au moins une valeur de l'ensemble de valeurs prises par une variable, donc elle fait décroître M_2. Comme nous avons un nombre fini e de variables et un nombre fini a de valeurs potentielles pour chaque variable, cette règle sera appliquée au plus $e \times a$ fois car les autres règles au pire conservent M_2. M_1 et M_3 restent inchangés après l'application de la règle **ArcConsistency**.
- La règle **Falsity** peut être appliquée au plus une fois car lorsqu'elle s'applique, l'ensemble de contraintes est réécrit à **F** et aucune règle ne peut plus être appliquée. Donc, l'application de cette règle fait décroître M_1, M_2 et M_3.
- La règle **Instantiation** décroît M_1, mais M_2 et M_3 restent inchangés. Puisque l'application de cette règle élimine la contrainte d'appartenance associée à la variable qui est instanciée et qu'il y a e contraintes d'appartenance au début du processus de résolution des contraintes, cette règle peut être appliquée au plus e fois car les autres règles au pire conservent M_1.
- La règle **Elimination** simplifie l'ensemble de contraintes par le remplacement d'une variable par sa valeur et de cette façon elle décroît M_3 et laisse M_1 et M_2 inchangées. Cette règle peut donc être appliquée au plus e fois, une fois par variable, car les autres règles au pire conservent M_3.
- À chaque fois que nous appliquons la règle **SplitDomain** on divise l'ensemble de valeurs prises par une variable et on obtient ainsi deux sous-problèmes. Dans chaque sous-problème l'ensemble initial D_x, associé à la variable qui est énumérée, est remplacé par D'_x et D''_x, respectivement. Vu que la condition pour l'application de cette règle impose que $D'_x \cup D''_x = D_x$, $D'_x \cap D''_x = \emptyset$, $D'_x \neq \emptyset$, et $D''_x \neq \emptyset$, on a donc $D'_x \subset D_x$ et $D''_x \subset D_x$. De cette façon, si on désigne par M'_1, M'_2 et M'_3 les mesures de complexité pour le premier sous-problème et par M''_1, M''_2 et M''_3 les mesures de complexité pour le deuxième sous-problème, nous avons $M'_2 < M_2$ et $M''_2 < M_2$, ainsi que $M_1 = M'_1 = M''_1$ et $M_3 = M'_3 = M''_3$. Cette règle peut être appliquée au plus $a-1$ fois pour chaque variable et si l'on considère toutes les variables elle peut être appliquée au plus $e \times (a-1)$ fois car les autres règles au pire conservent M_2.

Du fait que chaque règle décroît au moins l'une des mesures M_1, M_2 ou M_3 et qu'aucune règle n'augmente ces critères, l'itération de cet ensemble de règles terminera forcément. Le comportement des règles est résumée dans le tableau 3.1.

□

Règle	M_1	M_2	M_3
ArcConsistency	=	↓	=
Falsity	↓	↓	↓
Instantiation	↓	=	=
Elimination	=	=	↓
SplitDomain	=	↓	=

TAB. 3.1 – *Effet sur les mesures de complexité de l'application des règles*

Lemme 3 *Étant donné un CSP P en forme basique, en appliquant les règles de* { **ArcConsistency, Falsity, Instantiation, Elimination, SplitDomain** } *on obtient toujours un CSP en forme basique ou le CSP* **F**.

Preuve: Au début du processus de résolution de contraintes nous avons un ensemble de contraintes en forme basique $C \wedge x_i \in^? D_{x_i}$; $\forall i \in I$. Analysons maintenant l'effet de l'application de ces règles sur la forme des contraintes.

- La règle **ArcConsistency** élimine seulement des valeurs des ensembles D_{x_i}, donc les contraintes restent en forme basique.
- La règle **Falsity** réécrit un CSP à **F** lorsque l'un des ensembles D_{x_i} devient vide.
- La règle **Instantiation** élimine une contrainte d'appartenance et ajoute une contrainte d'égalité contenant la même variable. Comme la forme basique que nous avons définie impose que toute variable doit apparaître dans une contrainte d'appartenance ou dans une contrainte d'égalité, alors les contraintes restent toujours en forme basique après l'application de cette règle.
- À partir d'une contrainte d'égalité, la règle **Elimination** remplace dans C une variable par sa valeur en gardant toujours la contrainte d'égalité. Toutes les contraintes d'arité $e > 1$ dans C, où la variable apparaît, sont transformées en des contraintes d'arité $(e-1)$, donc l'ensemble de contraintes reste en forme basique. Toutes les contraintes d'arité $e = 1$ dans C, où la variable apparaît, sont évaluées après le remplacement de la variable. Les formules closes satisfaites trivialement sont éliminées et donc l'ensemble de contraintes reste en forme basique. Lorsqu'une formule close est fausse cette règle réécrit l'ensemble de contraintes C à **F**.
- La règle **SplitDomain** crée deux sous-problèmes, chacun a la même forme que le problème original sauf qu'une contrainte d'appartenance est modifiée par l'élimination de quelques valeurs, mais les deux sous-problèmes créés sont en forme basique.

□

Lemme 4 *Étant donné un CSP P en forme basique, les formes normales obtenues lors de l'application des règles de l'ensemble* { **ArcConsistency, Falsity, Instantiation, Elimination, SplitDomain** } *correspondent aux formes résolues par énumération de P plus le CSP* **F**.

Preuve: Il faut rappeler qu'une forme résolue par énumération est une forme résolue contenant une contrainte d'égalité pour chaque variable et que toute autre contrainte est satisfaite trivialement.

On commence par prouver que toute forme résolue par énumération est normale par rapport à l'ensemble de règles.

- Les règles **ArcConsistency**, **Falsity**, **Instantiation** et **SplitDomain** s'appliquent sur des contraintes d'appartenance, donc évidemment elles ne peuvent pas s'appliquer sur une forme résolue par énumération car cette forme ne contient pas de contraintes d'appartenance.
- La règle **Elimination** est la seule à s'appliquer sur une contrainte d'égalité, mais sa condition d'application impose que la variable apparaissant dans la contrainte d'égalité soit présente dans le reste de l'ensemble de contraintes. Puisque chaque variable n'apparaît qu'une seule fois dans une forme résolue par énumération cette règle ne peut pas être appliquée.

On prouve maintenant que toute forme non résolue par énumération est reductible par une règle de notre système. La différence entre la forme basique de départ et la forme résolue par énumération que l'on désire obtenir consiste en la présence des contraintes d'appartenance et des contraintes du problème original. On montrera donc qu'à chaque fois qu'apparaît une de ces contraintes, on peut appliquer une des règles proposées.

- La règle **ArcConsistency** est applicable uniquement lorsqu'elle peut réduire l'ensemble des valeurs prises par une variable.
- Si l'une des contraintes d'appartenance associe un domaine vide la règle **Falsity** détectera cette situation et réécrira le CSP à **F**.
- La règle **Instantiation** remplace une contrainte d'appartenance par une contrainte d'égalité seulement si l'ensemble de valeurs prises par une variable est réduit à un élément.
- Chaque fois qu'une variable instanciée apparaît dans l'ensemble de contraintes original, la règle **Elimination** la remplace par sa valeur. Ainsi, toutes les contraintes d'arité $e > 1$, où la variable apparaît, sont transformées en des contraintes d'arité $(e-1)$. Les contraintes d'arité $e = 1$, où la variable apparaît, sont évaluées. Les formules closes satisfaites trivialement sont éliminées et donc on réduit l'ensemble de contraintes original. Lorsqu'une formule close est fausse cette règle réécrit l'ensemble de contraintes à **F**.
- Si les quatre règles précédentes ne peuvent plus être appliquées, c'est-à-dire aucun ensemble des valeurs prises par une variable ne peut être modifié, aucun de ces ensembles ni consiste en un seul élément ni est vide et toutes les variables instanciées ont été éliminées, mais il y a encore des contraintes d'appartenance, alors on pourra toujours appliquer la règle **SplitDomain**. Cette règle est applicable jusqu'à ce que l'ensemble des valeurs prises par une variable soit réduit à un seul élément. Une fois que ces ensembles sont réduits à un seul élément, les variables peuvent être instanciées en utilisant la règle **Instantiation**. Ce processus continuera jusqu'à ce que toutes les contraintes d'appartenance soient éliminées. La règle **Elimination** se charge de remplacer les variables instanciées par leur valeur et de réduire ainsi l'ensemble de contraintes en vérifiant toujours la satisfaisabilité des formules closes.

□

Théorème 1 *Étant donné un CSP P en forme basique, en appliquant les règles de* { **ArcConsistency**, **Falsity**, **Instantiation**, **Elimination**, **SplitDomain** } *tant que possible, on obtient* **F** *si et seulement si P n'a pas de solution, sinon on obtient une forme résolue par énumération de P.*

Preuve : Elle est directe en utilisant les quatre lemmes précédents.

□

3.5.3 Stratégies de vérification de consistance

Dans cette section, nous montrons comment simuler les techniques de résolution de CSPs par l'application contrôlée de l'ensemble de règles de réécriture présenté dans la section précédente. Nous commençons par mettre en œuvre des stratégies pour la vérification de consistance locale et ensuite nous inclurons ces stratégies dans un schéma de recherche exhaustive ce qui nous permettra de construire des stratégies pour la vérification de consistance globale.

Stratégies pour la vérification de consistance locale

Le pouvoir d'expression des systèmes de calcul nous permet de simuler différents heuristiques de résolution de CSPs à travers la notion de stratégie. De cette façon, par exemple, l'application répétée de la règle **ArcConsistency** nous permet d'implanter l'algorithme AC-1 présenté dans la section 3.3. Après chaque application de la règle **ArcConsistency** nous pouvons obtenir deux résultats possibles : soit l'ensemble de valeurs prises par une variable a été réduit mais il contient encore des éléments, soit cet ensemble est vide. Afin de détecter ces deux situations il nous faut seulement vérifier l'insatisfaisabilité du problème considéré en essayant d'appliquer la règle **Falsity**. La figure 3.19 présente une stratégie qui implante cette idée.

AC-1ForElementaryConstraints =>
repeat* (*ArcConsistency*; **first one** (*Falsity* , **id**))
end

FIG. 3.19 – *Stratégie pour implanter l'algorithme AC-1*

Une analyse plus détaillée de la transformation réalisée par la règle **ArcConsistency** nous permet de voir que lors de son application un cas particulier peut se produire : l'ensemble de valeurs prises par une variable a été réduit à un seul élément. Dans ce cas nous pourrions appliquer la règle **Instantiation** et ensuite essayer d'appliquer la règle **Elimination**. De cette façon, si la variable instanciée apparaît dans l'ensemble de contraintes nous pouvons le simplifier. Une stratégie qui considère cette idée est présentée dans la figure 3.20.

```
LocalConsistencyForElementaryConstraints =>
repeat* (ArcConsistency;
          first one (Instantiation; first one (Elimination , id) ,
                Falsity ,
                id
                )
          )
end
```

FIG. 3.20 – *Stratégie AC-1 avec élimination de variables*

Pour la vérification de consistance locale, nous avons uniquement utilisé des règles de simplification et des règles de propagation. L'application des deux stratégies présentées ici retourne un CSP consistant sur les arcs. Des niveaux différents de consistance pourrait être atteints en modifiant la définition de la règle **ArcConsistency**.

Stratégies pour la vérification de consistance globale

Afin d'obtenir un CSP globalement consistant, nous pourrions soit appliquer un niveau plus élevé de vérification de consistance dans la règle **ArcConsistency**, soit énumérer toutes les combinaisons possibles de valeurs. Dans cette section, on présente des stratégies pour la vérification de consistance globale par énumération.

Pour implanter les heuristiques présentées dans la section 3.3, nous introduisons une spécialisation de la règle **SplitDomain** présentée dans la figure 3.18. Une instance particulière de cette règle correspond au cas où un des domaines D'_x ou D''_x contient seulement une valeur. Ceci est exprimé par la règle **EnumerateValue** de la figure 3.21.

[EnumerateValue]
$x \in^? D_x \ \wedge \ C$
$\Rightarrow$
$x =^? v \ \wedge \ C$ **or** $x \in^? D_x \backslash v \ \wedge \ C$
if $v \in D_x$

FIG. 3.21 – *Spécialisation de la règle pour la séparation de l'ensemble de valeurs prises par une variable*

La règle **EnumerateValue** exprime le simple fait de la séparation d'un domaine pour réaliser une recherche exhaustive. À partir du problème initial nous générons deux sous-problèmes. Dans le premier sous-problème, nous éliminons une contrainte d'appartenance et nous instancions la variable associée à cette contrainte par une valeur quelconque (parmi ses valeurs possibles). Dans l'autre sous-problème, nous éliminons de l'ensemble de valeurs prises pour cette variable, la valeur choisie dans le premier sous-problème.

La première technique de recherche exhaustive présentée dans la section 3.3, *générer puis tester*, peut être facilement implantée en itérant l'application de l'opérateur de stratégie non-déterministe **dk** sur la règle **EnumerateValue**, afin d'énumérer toutes les combinaisons possibles de valeurs. Ensuite, la règle **Elimination** devra être appliquée pour remplacer toutes les variables par leurs valeurs.

La figure 3.22 présente la stratégie ***GenerateTest*** qui implante cette idée.

GenerateTest =>
repeat* (**dk** (*EnumerateValue*));
repeat* (*Elimination*)
end

FIG. 3.22 – *Stratégie de* générer puis tester

La stratégie ***GenerateTest*** est exécutée de la manière suivante.

- D'abord, la règle **EnumerateValue** est essayée. Si c'est possible, elle est appliquée sur le problème initial et on obtient comme résultat deux nouveaux problèmes. Chacun de ces nouveaux problèmes devient le problème d'entrée pour la prochaine itération de la première boucle **repeat***. De cette façon, à chaque itération de la boucle **repeat*** on crée un point de choix. Ceci est itéré jusqu'à ce que la règle **EnumerateValue** ne puisse plus être appliquée. Le résultat de la dernière itération de la boucle **repeat*** sur chaque problème obtenu au cours de l'itération sera un problème d'entrée pour la deuxième boucle **repeat***.
- La deuxième boucle **repeat*** itère l'application de la règle **Elimination** sur chaque problème d'entrée jusqu'à ce qu'elle ne puisse plus être appliquée. L'union des résultats de la dernière application de la boucle sur chaque problème d'entrée sera le résultat final de l'application de la stratégie ***GenerateTest***.

Une simple modification dans l'ordre d'application des règles dans la stratégie ***GenerateTest*** nous permet de simuler la technique de recherche de *Backtracking*. Au lieu de remplacer les variables par leurs valeurs une fois que toutes les variables sont instanciées, nous pouvons remplacer une variable par sa valeur après chaque instanciation. Cette modification est réalisée dans la stratégie ***Backtracking*** présentée dans la figure 3.23.

Backtracking =>
repeat* (**dk** (*EnumerateValue*) ; **first one** (*Elimination* , **id**))
end

FIG. 3.23 – *Stratégie de* Backtracking

Dans la stratégie ***Backtracking***, le sens de **first one** (*Elimination* , **id**) est le suivant

- on essaie d'abord d'appliquer la règle **Elimination** et, si c'est possible, elle est appliquée et on obtient le résultat ;
- si on ne peut pas appliquer la règle **Elimination**, on essaie d'appliquer la deuxième règle, **id**. Dû fait que la règle **id** peut toujours être appliquée et qu'en plus elle retourne la contrainte d'entrée, elle est appliquée et on continue sans avoir modifié l'ensemble de contraintes.

Pour simuler la stratégie hybride de *Full Lookahead* il nous faut seulement inclure dans la stratégie ***Backtracking*** la vérification de consistance locale comme un pas de pré-traitement au tout début du processus de résolution et après chaque instanciation. La stratégie ***FullLookaheadForElementaryConstraints*** qui inclut ces simples modifications est présentée dans la figure 3.24.

FullLookaheadForElementaryConstraints =>
LocalConsistencyForElementaryConstraints ;
repeat* (**dk** (*EnumerateValue*) ;
first one (*Elimination* , **id**) ;
LocalConsistencyForElementaryConstraints
)
end

FIG. 3.24 – *Stratégie de* Full Lookahead

Si on applique les stratégies pour la vérification de consistance globale présentées dans cette section à un problème insatisfaisable alors on obtient **F**. Si le problème a au moins une solution, l'application de ces stratégies retourne de formes résolues qui consistent uniquement en une contrainte d'égalité par variable. En fait, chaque solution correspond à une affectation complète, i.e., toutes les variables sont instanciées. Si nous sommes intéressés à trouver toutes les solutions d'un problème nous obtiendrons autant de formes résolues que de solutions du problème. Évidemment, d'un point de vue pratique, cette situation doit être prise en compte lors de la résolution d'un problème qui a un grand nombre de solutions, ce qui en général est difficile à prévoir.

Ces trois stratégies nous permettent de justifier clairement une heuristique générale utilisée dans la résolution de contrainte : réaliser des calculs déterministes autant que possible avant de réaliser des calculs non-déterministes. Dans ce cas, le calcul non-déterministe est représenté par l'énumération des variables et de leurs valeurs. La différence principale entre ces stratégies se base sur l'élimination ou non des valeurs prises par les variables avant de réaliser des choix *a priori* de valeurs lors d'une énumération exhaustive.

Dans cette section, nous avons exprimé des techniques de résolution de CSPs par des règles de réécriture et des stratégies. Les règles de réécriture ont été définies le plus simplement possible afin d'isoler les transformations essentielles réalisées par ces techniques. Ceci a pour avantage que

les propriétés de correction et de complétude peuvent être prouvées très facilement : il suffit uniquement d'étudier chaque transformation individuellement. Nous avons montré que l'application de ces règles dans des ordres différents nous permet de simuler très facilement des techniques de résolution de CSPs basées sur un *backtracking* chronologique. Cette approche nous permet aussi de mieux comprendre la résolution de CSPs : nous avons clairement exprimé les heuristiques décrites dans la section 3.3 et nous avons réalisé quelles sont les différences fondamentales entre elles. Cependant, il est important de signaler que des heuristiques du style *Intelligent Backtracking* [15] et *Conflict-Directed Backjumping* [98], développées pour expliciter les raisons des échecs lors du processus de résolution, ne peuvent pas être simulées facilement. Ceci est dû au enchaînement des inférences dans le langage de stratégies utilisé qui est basé sur un mécanisme de retour en arrière. Celui-ci ne permet pas de sauts dans l'arbre de recherche.

3.6 Implantation : COLETTE

Afin de valider les idées présentées tout au long de ce travail, nous avons implanté en ELAN le système COLETTE (COnstraint Lixiviation, Estrategias y Técnicas para el Tratamiento de los Elementos[17]) [25].

Dans cette section nous décrivons brièvement l'implantation du système de calcul présenté dans la section 3.5. Nous introduisons également des règles qui nous permettent d'ajouter des critères de sélection des variables et des valeurs. Nous traitons ensuite le cas des CSPs décomposables. Nous montrons les possibilités de réaliser une mise en œuvre flexible des heuristiques en utilisant des stratégies paramétrées et nous signalons quelques considérations liées à la gestion des points de choix lors de l'énumération de valeurs. Finalement, on présente un exemple afin de décrire le fonctionnement du solveur ainsi construit.

3.6.1 Structure de données

La définition des symboles de fonctions présentée dans l'exemple 7 nous permet de travailler avec un ensemble de contraintes sur les entiers bornés. Pour considérer les contraintes d'appartenance, les contraintes d'égalité et l'opérateur de conjonction que nous utilisons dans notre formalisation, on définit les symboles de fonctions suivants

```
operators
global
        @ in? @      : (var             domain)        constraint;
        @ = @        : (var               term)        constraint;
        @ & @        : (constraint constraint)         constraint (AC) pri 20;
end
```

La sorte `domain` utilisée dans la définition de la sorte `constraint` est spécifiée de la manière suivante dans le cas des entiers bornés

17. Lixiviation de contraintes, stratégies et techniques pour le traitement des éléments.

```
sort    domain;
end

operators
global
        empty       :                   domain;
        [@,..,@]    : (int int)         domain;
        @ U @       : (domain domain)   domain assocRight;
end
```

Nous définissons l'objet représentant un CSP, et qui est réécrit par le processus de calcul, comme étant un 5-uplet

$$CSP[lmc, lec, EC, DC, store]$$

où

- *lmc* est une liste qui contient les contraintes d'appartenance ;
- *lec* est une liste qui contient les contraintes d'égalité ;
- *EC* est une liste qui contient les contraintes élémentaires à résoudre ;
- *DC* est une liste qui contient les contraintes disjonctives à résoudre [18] ;
- *store* est une liste qui contient les contraintes élémentaires déjà vérifiées.

Cette définition est exprimée en ELAN de la manière suivante

```
operators
global
 CSP@           : (set:tuple[list[constraint], list[constraint], list[constraint],
                             list[constraint], list[constraint]])      csp;
 Unsatisfiable  :  csp;
end
```

Le choix de cette structure de données se base sur les considérations suivantes.

- L'utilisation d'un connecteur logique de conjonction associatif-commutatif améliore la clarté de la formalisation des règles de transformation. Cependant, au niveau de l'implantation il est souhaitable de gérer séparément les différents types de contraintes (contraintes d'appartenance, contraintes d'égalité, contraintes élémentaires, contraintes disjonctives).
- Nous avons vu qu'en utilisant les règles de transformation, définies dans la section 3.5, nous sommes capables de mettre en œuvre des techniques de résolution de CSPs. Néanmoins, l'algorithme que l'on utilise pour la vérification de consistance locale, l'algorithme AC-1, est trop naïf. Sa performance est sensiblement inférieure à celle de l'algorithme AC-3. Afin d'implanter l'algorithme AC-3 nous introduisons la notion de *store* de contraintes, ce qui nous permettra de stocker les contraintes qui ont été vérifiées. Cette notion de *store* de contraintes est présente dans tous les systèmes de résolution de contraintes.
- L'utilisation des listes pour stocker les contraintes de chaque type nous permettra principalement de faire des opérations de tri nécessaires pour ajouter des critères de sélection de contraintes, sélection de variables et sélection de valeurs. En fait, notre langage de contraintes permet d'exprimer des conjonctions de contraintes en utilisant l'opérateur &,

18. Afin de ne pas fausser la vraie structure de données utilisée dans l'implantation de COLETTE, on inclut dans cette définition les contraintes disjonctives même si elles ne seront traitées que dans le chapitre 5.

mais au début du processus de calcul nous transformons la conjonction de contraintes élémentaires en une liste de contraintes élémentaires.

3.6.2 Règles de transformation

L'implantation de la règle **ArcConsistency** est orientée de façon à pouvoir implanter la vérification de consistance locale basée sur l'algorithme AC-3 [76].

```
rules for csp
        lv                                          : list[var];
        c                                           : constraint;
        lmc, lec, EC, DC, store                     : list[constraint];
        lmc1, lmc2, EC1, newstore, remainingstore : list[constraint];
global

[ArcConsistency]
CSP[lmc,lec,c.EC,DC,store]
=>
CSP[append(lmc1,lmc2),lec,append(EC1,EC),DC,newstore]
where          [lmc1,lmc2,lv] := ()ReviseDxWRTc(lmc,lec,c)
where    [EC1,remainingstore] := ()GetConstraintsOnVar(lv,store,nil,nil)
choose   try     if ArityOfConstraint(c) == 1
                 where newstore := ()remainingstore
         try     if ArityOfConstraint(c) >= 2
                 where newstore := ()c.remainingstore
end
end
```

L'application de la règle **ArcConsistency** a donc l'effet suivant.

- La première contrainte dans la liste de contraintes élémentaires est vérifiée en utilisant l'opérateur `ReviseDxWRTc` qui retourne la liste de contraintes d'appartenance dont l'ensemble de valeurs prises par la variable a pour cardinalité 0 ou 1 (`lmc1`)[19], le reste de la liste de contraintes d'appartenance (`lmc2`) et la liste de variables dont leur ensemble de valeurs possible a été modifiés (`lv`).
- L'opérateur `GetConstraintsOnVar` récupère du `store` toutes les contraintes qui contiennent des variables dont leur ensemble de valeurs possibles a été modifié (`EC`) et le reste du `store` (`remainingstore`).
- Si la contrainte analysée a une arité 1, le nouveau *store* (`newstore`) est construit seulement à partir de `remainingstore`. Si la contrainte analysée a une arité supérieur à 1, le nouveau *store* (`newstore`) est construit à partir de `remainingstore` et de la contrainte analysée.
- Finalement, le côté droite de la règle est construit par de simples unions de listes à partir des résultats intermédiaires.

L'implantation de la règle **Falsity** vérifie simplement si une des contraintes d'appartenance associe un domaine vide. Si c'est le cas, elle réécrit le problème à `Unsatisfiable`.

19. Si l'un des ensembles est vide il sera toujours en tête de la liste.

```
rules for csp
        x                          : var;
        lmc, lec, EC, DC, store : list[constraint];
global

[Falsity]
CSP[x in? empty.lmc,lec,EC,DC,store]
=>
Unsatisfiable
end
```

L'implantation de la règle **Instantiation** vérifie si une des contraintes d'appartenance associe un domaine dont la cardinalité est 1. Si c'est le cas, elle élimine la contrainte d'appartenance et crée une contrainte d'égalité qui instancie une variable.

```
rules for csp
        x                          : var;
        v                          : int;
        D                          : domain;
        lmc, lec, EC, DC, store : list[constraint];
global

[Instantiation]
CSP[x in? D.lmc,lec,EC,DC,store]
=>
CSP[lmc,x = v.lec,EC,DC,store]
if GetCard(D) == 1
where   v := ()GetFirstValueOfDomain(D)
end
```

La règle `Elimination` ci-desous réalise le remplacement d'une variable par sa valeur de la manière suivante.

- La première condition pour l'application de cette règle est l'occurrence de la variable dans l'ensemble de contraintes élémentaires ou dans l'ensemble de contraintes disjonctives.
- L'opérateur `GetRemainingConstraintsInEC` remplace la variable par sa valeur dans l'ensemble de contraintes élémentaires et retourne deux listes de contraintes : la première (`l1`) contient les contraintes unaires, la deuxième (`l2`) contient les autres contraintes élémentaires. Si lors du remplacement de la variables une formule close est évaluée à faux, alors la contrainte `false`, qui représente un CSP insatisfaisable, est placée en tête de la liste de contraintes unaires (dans ce cas, cette contrainte sera la seule composante de la liste).
- L'opérateur `GetRemainingConstraintsInDC` réalise le remplacement de la variable par sa valeur dans l'ensemble de contraintes disjonctives et retourne trois listes : la première (`l3`) contient les contraintes unaires, la deuxième (`l4`) contient les autres contraintes élémentaires et la troisième (`l5`) contient les contraintes disjonctives. Les contraintes élémentaires retournées sont celles qui ont été créées par la simplification des composantes des disjonctions. Le traitement des contraintes disjonctives sera bien expliqué dans le chapitre 5.
- Finalement, on réalise l'union des contraintes unaires (`l1` et `l3`) et l'union des autres contraintes élémentaires (`l2` et `l4`) pour créer une liste contenant toutes les contraintes élémentaires où les premiers éléments sont des contraintes unaires. L'opérateur `Shift-Constraint` envoie la contrainte d'égalité à la fin de la liste pour pouvoir traiter les contraintes d'égalité suivantes.

```
rules for csp
        x                              : var;
        v                              : int;
        P                              : csp;
        lmc, lec, EC, DC, store        : list[constraint];
        l1, l2, l3, l4, l5, lu, lb : list[constraint];
global

[Elimination]
CSP[lmc,x = v.lec,EC,DC,store]
=>
P
if      occurs x in EC or occurs x in DC
where   [l1,l2] := ()GetRemainingConstraintsInEC(x = v,EC,nil,nil)
choose  try if l1 == false.nil
            where P := ()Unsatisfiable
        try if l1 != false.nil
            where [l3,l4,l5]:=()GetRemainingConstraintsInDC(x = v,
                                DC,nil,nil,nil)
            choose try if l3 == false.nil
                       where P := ()Unsatisfiable
                   try if l3 != false.nil
                       where lu := ()append(l1,l3)
                       where lb := ()append(l2,l4)
                       where P:=()CSP[lmc,ShiftConstraint(lec,x = v),
                                      append(lu,lb),l5,store]
            end
end
```

L'implantation de la règle **EnumerateValue** sera détaillée dans la section 3.6.4.

3.6.3 Critères de sélection des variables

Maintenant que l'on peut trier les contraintes d'appartenance, car elles sont stockées dans une liste, on peut facilement inclure des critères de sélection des variables. Un critère largement connu consiste à choisir comme variable d'énumération celle qui a le plus petit ensemble de valeurs possibles[20]. Ce critère est une instance particulière d'un principe plus général appelé *First Fail Principle* [56]. La règle suivante, `GetVarWithMinimumDomain`, place en tête de la liste de contraintes d'appartenance celle qui satisfait ce critère.

20. Minimum Domain.

```
rules for csp
        x                                   : var;
        D                                   : domain;
        lmc, lec, EC, DC, store, newlmc : list[constraint];
global

[GetVarWithMinimumDomain]

CSP[x in? D.lmc,lec,EC,DC,store]
=>
CSP[newlmc,lec,EC,DC,store]
where   newlmc  := ()GetMinimumDomain(x in? D.lmc,nil,GetCard(D))
end

end
```

La définition de l'opérateur **GetMinimumDomain**, par des règles non-nommées, est la suivante.

```
operators
global
 GetMinimumDomain(@,@,@) : (list[constraint] list[constraint] int)  list[constraint];
end

rules for list[constraint]
        x       : var;
        n, m    : int;
        D       : domain;
        c       : constraint;
        l1, l2  : list[constraint];
global

[]      GetMinimumDomain(c.x in? D.l1,l2,n)
        =>
        where   m := ()GetCard(D)
        switch
        case    m < n   then
                GetMinimumDomain(x in? D.l1,c.l2,m)
        otherwise
                GetMinimumDomain(c.l1,x in? D.l2,n)
        end
end

[]      GetMinimumDomain(c.nil,l2,n)
        =>
        c.l2
end

end
```

Évidemment, le critère présenté ici n'est qu'une des possibilités de choix de variables. D'autres critères, tels que *Minimal Width Ordering* et *Maximum Cardinality Ordering* [107], peuvent être facilement implantés en changeant l'opérateur `GetMinimumDomain` par un autre plus adéquat.

3.6.4 Critères de sélection des valeurs

Pour gérer le non-déterminisme du coté droit de la règle **EnumerateValue**, présentée dans la section 3.5.3, nous le décomposons en deux règles : **InstantiateValue** et **EliminateValue**.

[InstantiateValue]
$x \in^? D_x \ \wedge \ C$
$\Rightarrow$
$x =^? v \ \wedge \ C$
if $v \in D_x$

FIG. 3.25 – *Règle pour l'instanciation de valeurs*

La règle **EliminateValue**, présentée dans la figure 3.26, correspond à la deuxième composante du côté droit de la règle **EnumerateValue**.

[EliminateValue]
$x \in^? D_x \ \wedge \ C$
$\Rightarrow$
$x \in^? D_x \backslash v \ \wedge \ C$
if $v \in D_x$

FIG. 3.26 – *Règle pour l'élimination de valeurs*

Avec ces deux règles nous sommes capables de gérer le non-déterminisme du côté droit de la règle **EnumerateValue**, mais nous n'avons pas encore spécifié clairement quelle est la valeur v à considérer pour l'application de la règle. Dans le cas des domaines ordonnés, on pourrait considérer, par exemple, une énumération des valeurs à partir de la plus petite ou de la plus grande valeur. Étant donné que l'on travaille sur les entiers bornés, on définit donc les règles `InstantiateFirstValueOfDomain` et `EliminateFirstValueOfDomain` qui implantent une version particulière des règles **InstantiateValue** et **EliminateValue**, respectivement.

```
rules for csp
        x                          : var;
        v                          : Type;
        D, RD                      : domain;
        lmc, lec, EC, DC, store : list[constraint];
global

[InstantiateFirstValueOfDomain]
CSP[x in? D.lmc,lec,EC,DC,store]
=>
CSP[lmc,x = v.lec,EC,DC,store]
if    D != empty
where v := ()GetFirstValueOfDomain(D)
end

[EliminateFirstValueOfDomain]
CSP[x in? D.lmc,lec,EC,DC,store]
=>
CSP[x in? RD.lmc,lec,EC,DC,store]
if    GetCard(D) > 1
where RD := ()DeleteFirstValueOfDomain(D)
end

end
```

3.6.5 Stratégies de vérification de consistance

Nous avons détaillé la description de toutes les règles nécessaires pour implanter les stratégies de vérification de consistance locale et globale considérant en plus des critères de sélection de variables et de valeurs.

La stratégie ***GenerateTest***, présentée dans la section 3.5, est implantée par la stratégie `GenerateTestFirstToLastAll` qui retourne toutes les solutions. Pour obtenir uniquement la première solution, nous définissons la stratégie `GenerateTestFirstToLastOne`. Ces deux stratégies énumèrent les valeurs à partir du plus petit élément.

```
stratop
global
        GenerateTestFirstToLastAll : < csp -> csp >      bs;
        GenerateTestFirstToLastOne : < csp -> csp >      bs;
end

strategies for csp
implicit

[] GenerateTestFirstToLastAll =>
repeat* (dk (InstantiateFirstValueOfDomain , EliminateFirstValueOfDomain)) ;
repeat* (Elimination) ;
GetSolutionCSP
end

[] GenerateTestFirstToLastOne =>
first one (GenerateTestFirstToLastAll)
end

end
```

La stratégie `BacktrackingFirstToLastAll` retourne toutes les solutions, implantant ainsi la stratégie ***Backtracking*** présentée dans la section 3.5, alors que la stratégie `BacktrackingFirstToLastOne` ne retourne que la première solution.

```
stratop
global
        BacktrackingFirstToLastAll : < csp -> csp >      bs;
        BacktrackingFirstToLastOne : < csp -> csp >      bs;
end

strategies for csp
implicit

[] BacktrackingFirstToLastAll =>
repeat* (dk (InstantiateFirstValueOfDomain ; first one (Elimination , id) ,
             EliminateFirstValueOfDomain)) ;
GetSolutionCSP
end

[] BacktrackingFirstToLastOne =>
first one (BacktrackingFirstToLastAll)
end

end
```

La stratégie `LocalConsistencyForEC` vérifie la consistance locale pour un ensemble de contraintes élémentaires, implantant ainsi la stratégie ***LocalConsistencyForElementaryConstraints*** présentée dans la section 3.5.

```
stratop
global
        LocalConsistencyForEC : < csp -> csp >      bs;
end

strategies for csp
implicit

[] LocalConsistencyForEC =>
repeat* (ArcConsistency ;
         repeat* (first one (Instantiation ;
                             first one (ExtractConstraintsOnEqualityVar , id) ;
                             first one (Elimination , id) ,
                             Falsity)
                  )
        )
end

end
```

La stratégie `FLAMinimumDomainFirstToLastAll` implante la stratégie ***FullLookaheadForElementaryConstraints*** présentée dans la section 3.5, laquelle retourne toutes les solutions. La stratégie `FLAMinimumDomainFirstToLastOne` retourne quant à elle uniquement la première solution. L'implantation de ces deux stratégies considère aussi un critère dynamique de sélection des variables : lors de chaque instanciation, on choisit la variable qui associe le plus petit ensemble de valeurs possibles.

```
stratop
global
        FLAMinimumDomainFirstToLastAll : < csp -> csp >      bs;
        FLAMinimumDomainFirstToLastOne : < csp -> csp >      bs;
end

strategies for csp
implicit

[] FLAMinimumDomainFirstToLastAll =>
LocalConsistencyForEC ;
repeat* (GetVarWithMinimumDomain ;
         dk (InstantiateFirstValueOfDomain ;
             first one (ExtractConstraintsOnEqualityVar , id) ;
             first one (Elimination , id) ;
             LocalConsistencyForEC ,
             EliminateFirstValueOfDomain)
        ) ;
GetSolutionCSP
end

[] FLAMinimumDomainFirstToLastOne =>
first one (FLAMinimumDomainFirstToLastAll)
end

end
```

Finalement, une simple modification à cette dernière stratégie nous permet d'implanter l'heuristique de *Forward Checking* ci-desous.

```
stratop
global
        FCMinimumDomainFirstToLastAll : < csp -> csp >      bs;
end

strategies for csp
implicit

[] FCMinimumDomainFirstToLastAll =>
LocalConsistencyForEC ;
repeat* (GetVarWithMinimumDomain ;
         dk (InstantiateFirstValueOfDomain ;
             first one (ExtractConstraintsOnEqualityVar , id) ;
             first one (Elimination , id) ;
             first one (LocalConsistencyInEC , id) ,
             EliminateFirstValueOfDomain)) ;
GetSolutionCSP
end

end
```

Nous avons seulement changé la stratégie `LocalConsistencyForEC` par la stratégie `LocalConsistencyInEC` : la première vérifie la consistance de tout l'ensemble de contraintes, la deuxième vérifie la consistance seulement pour les contraintes qui ont été modifiées par le remplacement d'une variable par sa valeur lors de l'application de la règle `Elimination`.

3.6.6 CSPs décomposables

La décomposition d'un CSP en un ensemble de sous-problèmes dont leurs ensembles de variables sont disjoints, lorsqu'elle est possible, permet évidement d'améliorer les performances car on résoud des problèmes plus simples et on a en plus la possibilité de considérer, par exemple, leur résolution en parallèle.

Dans cette section, on analyse le cas des CSPs qui peuvent être décomposés en plusieurs sous-problèmes. Même si c'est un cas plutot rare, il nous servira pour l'analyse postérieure des contraintes disjonctives et on l'utilisera aussi comme un exemple pour montrer différents schémas de coopération de solveurs.

La figure 3.27, montre l'espace de recherche d'un problème qui peut être décomposé en trois sous-problèmes.

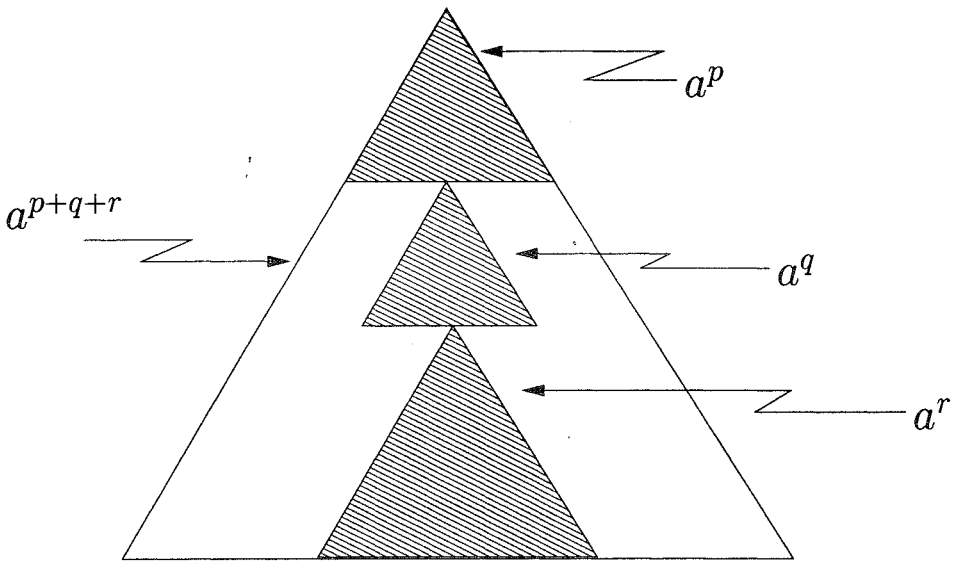

FIG. 3.27 – *Espace de recherche lors de la décomposition d'un CSP*

Si on considère que le problème initial peut être décomposé en trois sous-problèmes de p, q et r variables et que chaque variable peut prendre a valeurs, alors l'espace de recherche de chaque sous-problème est respectivement a^p, a^q et a^r. Évidemment, l'espace de recherche du problème initial a^{p+q+r} est plus grand que la somme des espaces de recherche des trois sous-problèmes.

La règle **Split-Disjunctions**, présentée dans la figure 3.28, est définie pour créer deux sous-problèmes dont les ensembles de variables sont disjoints. L'idée derrière cette règle, quand elle est itérée sur chacun de ses résultats, est d'obtenir le maximum de sous-problèmes indépendants.

[SplitConstraints]
$\bigwedge_{k \in K} c_k^?$
$\Rightarrow$
$\bigwedge_{k \in K'} c_k^?$ **and** $\bigwedge_{k \in K''} c_k^?$
if $K' \cup K'' = K$ **and** $\bigcup_{k \in K'} \mathcal{V}ar(c_k^?) \cap \bigcup_{k \in K''} \mathcal{V}ar(c_k^?) = \emptyset$

FIG. 3.28 – *Règle pour la création de deux sous-ensembles de contraintes avec ensembles de variables disjoints*

Puisqu'en appliquant cette règle on obtient un ensemble de CSPs chacun étant un ensemble de contraintes élémentaires, il nous suffit simplement d'appliquer à chaque CSP résultant une stratégie pour la résolution d'un ensemble de contraintes élémentaires, comme celles définies dans la section 3.6.5, pour obtenir la solution du problème initial. Ceci est exprimé par la stratégie ***SolveDisjointCSP*** présentée dans la figure 3.29.

SolveDisjointCSP =>
repeat* (*SplitConstraints*;
FullLookaheadForElementaryConstraints)
end

FIG. 3.29 – *Stratégie pour la résolution de CSPs décomposables*

En appliquant la stratégie ***SolveDisjointCSP*** la solution du problème initial est obtenue

par conjonction des solutions de chaque sous-problème.

Afin de faciliter la manipulation des sous-problèmes, on définit la structure de données suivante en ELAN.

```
operators
global
        CSPDis@     : (set:tuple[csp,list[csp],list[csp]])     cspdis;
end
```

L'idée est de stocker dans la première composante le problème initial, dans la deuxième une liste contenant les sous-problèmes à résoudre et dans la troisième une liste contenant les sous-problèmes déjà résolus.

Pour implanter la règle **SplitConstraints** nous définissons en ELAN la règle `DecomposeCSPDis` ci-dessous.

```
rules for cspdis
        lmc, lec, EC, DC, store        : list[constraint];
        luCsp, luCsp1, newluCsp, lsCsp : list[csp];
global

[DecomposeCSPDis]
        CSPDis[CSP[lmc,lec,EC,DC,store],luCsp,lsCsp]
        =>
        CSPDis[CSP[nil,nil,nil,nil,nil],newluCsp,lsCsp]
        where   luCsp1  := ()CreateListOfCSP(lmc)
        where   newluCsp:= ()DecomposeCSP(luCsp1,EC)
end
end
```

L'explication de cette dernière règle est la suivante.

- Le CSP initial contient une contrainte d'appartenance pour chaque variable dans l'ensemble de contraintes. Elles sont toutes stockées dans la liste `lmc`. Au début la liste de contraintes d'égalité `lec` est vide.
- Toutes les contraintes élémentaires du problème sont stockées dans la liste `EC`. Pour l'instant nous ne traitons pas de contraintes disjonctives, donc la liste `DC` est vide. La liste `store` est également vide.
- L'opérateur `CreateListOfCSP` est défini récursivement de la manière suivante :

  ```
  CreateListOfCSP(x in? D.lmc) =>
  CSP[x in? D.nil, nil, nil, nil, nil].CreateListOfCSP(lmc)
  end

  CreateListOfCSP(nil) =>
  nil
  end
  ```

  Ainsi, on crée une liste `luCsp1` de CSPs, chacun contenant une seule contrainte d'appartenance extraite de la liste `lmc`. Cette liste `luCsp1` correspond au nombre maximum de sous-problèmes que l'on peut créer à partir de l'ensemble de contraintes initial.
- À partir de chaque contrainte élémentaire apparaissante dans la liste `EC`, l'opérateur `DecomposeCSP` extrait ses variables, cherche dans la liste `luCsp1` les composantes contenant ces variables et réalise leur union. De cette manière on regroupe les composantes de la liste

`luCsp1` lorsqu'ils partagent de variables communes dans l'ensemble de contraintes élémentaires. On procède dans l'ordre inverse par rapport à la règle **SplitConstraints**, qui essaie de décomposer un ensemble de contraintes en deux sous-ensembles. Nous avons choisi cette méthode car elle a une complexité en temps linéaire dans le nombre de contraintes élémentaires du problème.

- Il faut finalement signaler que la règle `DecomposeCSPDis` élimine le CSP initial.

Pour résoudre chaque sous-problème nous définissons la règle `SolveSubCSPDis` présentée ci-dessous.

```
rules for cspdis
        P, Q, R      : csp;
        luCsp, lsCsp : list[csp];
global

[SolveSubCSPDis]
        CSPDis[P,Q.luCsp,lsCsp]
        =>
        CSPDis[P,luCsp,R.lsCsp]
        where   R := (FLAMinimumDomainFirstToLastOne)Q
end
end
```

L'explication de cette règle est très simple : nous appliquons une stratégie de résolution de CSPs au premier élément de la liste contenant les sous-problèmes à résoudre, nous ajoutons le résultat à la liste de sous-problèmes déjà résolus et nous enlevons le sous-problème de la liste des sous-problèmes à résoudre.

Finalement, pour construire la solution du problème initial, nous la composons à partir des solutions des sous-problèmes comme il est exprimé par la règle `ComposeCSPDis`.

```
rules for cspdis
        P, Q, R      : csp;
        luCsp, lsCsp : list[csp];
global

[ComposeCSPDis]
        CSPDis[P,luCsp,Q.lsCsp]
        =>
        CSPDis[R,luCsp,lsCsp]
        where   R := ()ComposeCSP(P,Q)
end
end
```

Cette règle utilise l'opérateur `ComposeCSP` qui réalise l'union des listes contenant le même type de contraintes.

À partir de cet ensemble de règles on définit la stratégie `SolveCSPDis` de la manière suivante.

```
stratop
global
        SolveCSPDis : < cspdis -> cspdis >    bs;
end

strategies for cspdis
implicit

[] SolveCSPDis =>
first one (DecomposeCSPDis);
repeat* (first one (SolveSubCSPDis));
repeat* (first one (ComposeCSPDis))
end
end
```

Le comportement de cette stratégie est très simple : on décompose au maximum le CSP initial, on résout chacun des sous-problèmes et, à partir des solutions des sous-problèmes, on compose la solution au problème initial.

D'une manière générale, il faut signaler que, même si la formalisation, présentée sous la forme d'une règle de décomposition d'un CSP et d'une stratégie, est facile à comprendre et exprimée d'une façon très abstraite, pour des raisons d'implantation nous avons été obligés de la modifier. La principale raison est évidemment la façon de traiter le décomposition de l'ensemble de contraintes.

3.6.7 Flexibilité de prototypage en utilisant des stratégies paramétrées

L'art de la résolution de contraintes consiste en un processus d'itérations entre vérification de consistance locale et séparation des ensembles de valeurs prises par les variables, de façon telle qu'une petite partie de l'espace de recherche soit explorée avant de trouver une solution. Pour cette raison on ajoute toujours des critères différents, par exemple, pour la sélection de variables ou de valeurs. Évidemment, définir des stratégies spécifiques pour chaque combinaison possible de heuristiques et critères semble impraticable. Dans ce contexte, les besoins d'un langage flexible pour la mise en œuvre des heuristiques sont clairs. En ELAN, nous avons la possibilité d'utiliser des stratégies paramétrées ce qui nous permet de prototyper une heuristique considérant plusieurs critères sans avoir à définir explicitement chaque heuristique.

On peut considérer comme exemple la définition de la stratégie `FLAMinimumDomainFirstToLastAll`, présentée dans la section 3.6.5. Cette stratégie utilise comme critère de sélection des variables le principe du plus petit ensemble de valeurs possibles. Comme nous l'avons déjà signalé, on peut imaginer plusieurs critères de ce style, comme par exemple, la variable qui apparaît le plus dans l'ensemble de contraintes.

La définition suivante nous permet d'utiliser l'heuristique de *Full Lookahead* avec comme paramètre le critère de sélection des variables :

```
stratop
global
      FullLookAheadFirstToLastAll(@) : < csp -> csp >   < csp -> csp >;
end

strategies for csp
implicit

     RuleChoiceVar : < csp -> csp >;

[] FullLookAheadFirstToLastAll(RuleChoiceVar) =>
LocalConsistencyForEC ;
repeat* (RuleChoiceVar) ;
         dk (InstantiateFirstValueOfDomain ;
             first one (ExtractConstraintsOnEqualityVar , id) ;
             first one (Elimination , id) ;
             LocalConsistencyForEC ,
             EliminateFirstValueOfDomain)) ;
GetSolutionCSP
end
```

Ainsi, si l'on applique la stratégie

```
FullLookAheadFirstToLastAll(GetVarWithMinimumDomain)
```

on obtient une stratégie correspondant à `FLAMinimumDomainFirstToLastAll`.

3.6.8 Traitement de l'énumération

Lorsque l'on applique la stratégie `FLAMinimumDomainFirstToLastAll` à un CSP

$$x \in^? [1, \ldots, 10] \wedge C$$

l'énumération des valeurs est réalisée de la manière suivante.

- La première itération de la boucle **repeat**$*$ crée un point de choix. Une branche contient le CSP $x = 1 \wedge C$ et l'autre branche contient le CSP $x \in^? [2, \ldots, 10] \wedge C$.
- Lorsque l'on continue à énumérer les valeurs de la variable x dans le deuxième sous-problème, on crée à nouveau un point de choix : une branche contient le CSP $x = 2 \wedge C$ et l'autre branche contient le CSP $x \in^? [3, \ldots, 10] \wedge C$.

On voit que le nombre de points de choix créés est proportionnel au nombre de valeurs potentielles de la variable à énumérer. Ce comportement peut être amélioré par la définition de la stratégie suivante.

```
stratop
global
        FLAMinimumDomainFirstToLastAll : < csp -> csp >      bs;
end

strategies for csp
implicit

[] FLAMinimumDomainFirstToLastAll =>
LocalConsistencyForEC ;
repeat* (GetVarWithMinimumDomain ;
        dk (iterate* (EliminateFirstValueOfDomain)) ;
        first one (InstantiateFirstValueOfDomain) ;
        first one (ExtractConstraintsOnEqualityVar , id) ;
        first one (Elimination , id) ;
        LocalConsistencyForEC ) ;
GetSolutionCSP
end

end
```

En appliquant cette stratégie, l'utilisation adéquate des opérateurs **dk** et **iterate*** nous permet de créer un seul point de choix lors de l'énumération d'une variable.

3.6.9 Exemple

Afin de décrire le fonctionnement du solveur construit, dans cette section nous illustrons la résolution d'un problème très simple de calcul de calendriers.

Nous considérons ci-dessous une conjonction de contraintes élémentaires d'arité quelconque utilisé pour le calcul du calendrier babylonien [37] :

```
Nb =? 360(*)Yb (+) 30(*)Mb (+) Db &
Db <? 30 &
Mb <? 12 &
Nb =? 1000
```

Les trois premières contraintes expriment la relation entre une date absolue (Nb) et l'année (Yb), le mois (Mb) et le jour (Db) dans le calendrier babylonien. Pour exprimer, par exemple, la date absolue 1000 dans le calendrier babylonien, on ajoute la quatrième contrainte. La trace de l'application de la stratégie `FLAMinimumDomainFirstToLastAll` est la suivante[21].

```
[] start with term:
CreateCSP(Nb=?360(*)Yb(+)30(*)Mb(+)Db & Db<?30 & Mb<?12 & Nb=?1000)

'ArcConsistency on constraint Nb=?1000':
CSP[Nb in?[1000,..,1000].Db in?[1,..,100000].Mb in?[1,..,100000].Yb in?[1,..,100000].nil,
nil, 1(*)Mb<?12.1(*)Db<?30.1(*)Nb=?360(*)Yb(+)30(*)Mb(+)1(*)Db.nil, nil, nil]

'Instantiation of variable Nb':
CSP[Db in?[1,..,100000].Mb in?[1,..,100000].Yb in?[1,..,100000].nil,
Nb=1000.nil, 1(*)Mb<?12.1(*)Db<?30.1(*)Nb=?360(*)Yb(+)30(*)Mb(+)1(*)Db.nil, nil, nil]
```

21. Nous initialisons à [1,..,100000] l'ensemble de valeurs possibles pour chaque variable.

```
'Elimination of variable Nb':
CSP[Db in?[1,..,100000].Mb in?[1,..,100000].Yb in?[1,..,100000].nil,
Nb=1000.nil, 1(*)Db<?30.1(*)Mb<?12.1000=?360(*)Yb(+)30(*)Mb(+)1(*)Db.nil, nil, nil]

'ArcConsistency on constraint 1(*)Db<?30':
CSP[Db in?[1,..,29].Mb in?[1,..,100000].Yb in?[1,..,100000].nil,
Nb=1000.nil, 1(*)Mb<?12.1000=?360(*)Yb(+)30(*)Mb(+)1(*)Db.nil, nil, nil]

'ArcConsistency on constraint 1(*)Mb<?12':
CSP[Mb in?[1,..,11].Yb in?[1,..,100000].Db in?[1,..,29].nil,
Nb=1000.nil, 1000=?360(*)Yb(+)30(*)Mb(+)1(*)Db.nil, nil, nil]

'ArcConsistency on constraint 1000=?360(*)Yb(+)30(*)Mb(+)1(*)Db':
CSP[Yb in?[2,..,2].Mb in?[9,..,11].Db in?[1,..,29].nil,
Nb=1000.nil, nil, nil, 1000=?360(*)Yb(+)30(*)Mb(+)1(*)Db(+)0.nil]

'Instantiation of variable Yb':
CSP[Mb in?[9,..,11].Db in?[1,..,29].nil,
Yb=2.Nb=1000.nil, nil, nil, 1000=?360(*)Yb(+)30(*)Mb(+)1(*)Db.nil]

'ExtractConstraintsOnEqualityVar Yb':
CSP[Mb in?[9,..,11].Db in?[1,..,29].nil,
Yb=2.Nb=1000.nil, 1000=?360(*)Yb(+)30(*)Mb(+)1(*)Db.nil, nil, nil]

'Elimination of variable Yb':
CSP[Mb in?[9,..,11].Db in?[1,..,29].nil,
Nb=1000.Yb=2.nil, 1000=?30(*)Mb(+)1(*)Db(+)720.nil, nil, nil]

'ArcConsistency on constraint 1000=?30(*)Mb(+)1(*)Db(+)720':
CSP[Db in?[10,..,10].Mb in?[9,..,9].nil,
Nb=1000.Yb=2.nil, nil, nil, 1000=?30(*)Mb(+)1(*)Db(+)720.nil]

'Instantiation of variable Db':
CSP[Mb in?[9,..,9].nil,
Db=10.Nb=1000.Yb=2.nil, nil, nil, 1000=?30(*)Mb(+)1(*)Db(+)720.nil]

'ExtractConstraintsOnEqualityVar Db':
CSP[Mb in?[9,..,9].nil,
Db=10.Nb=1000.Yb=2.nil, 1000=?30(*)Mb(+)1(*)Db(+)720.nil, nil, nil]

'Elimination of variable Db':
CSP[Mb in?[9,..,9].nil,
Nb=1000.Yb=2.Db=10.nil, 1000=?30(*)Mb(+)730.nil, nil, nil]

'Instantiation of variable Mb':
CSP[nil, Mb=9.Nb=1000.Yb=2.Db=10.nil, 1000=?30(*)Mb(+)730.nil, nil, nil]

'Elimination of variable Mb':
CSP[nil, Nb=1000.Yb=2.Db=10.Mb=9.nil, nil, nil, nil]

'GetSolutionCSP':
CSP[nil, Yb=2.Mb=9.Db=10.Nb=1000.nil, nil, nil, nil]
```

```
[] result term:
CSP[nil, Yb=2.Mb=9.Db=10.Nb=1000.nil, nil, nil, nil]
```

Dans l'annexe A.1, nous montrons en détail des résultats expérimentaux de l'application de notre système pour la résolution de quatre problèmes crypto-arithmétiques, du problème des N reines et du problème de coloriage de graphes.

La figure 3.30 montre le comportement de douze stratégies pour la résolution du puzzle *SEND+MORE=MONEY*. D'une part, on peut remarquer que l'heuristique de *Forward Checking* est plus performante que l'heuristique de *Full Lookahead* lors que l'on considère les mêmes critères de choix de variables et de choix de valeurs. D'autre part, on observe que lors de la recherche d'une seule solution, l'ordre d'énumération *first to last* est plus performant que l'ordre d'énumération *last to first* uniquement lorsqu'il est utilisé sans critère de sélection des variables. L'ordre d'énumération n'est absolument pas significatif.

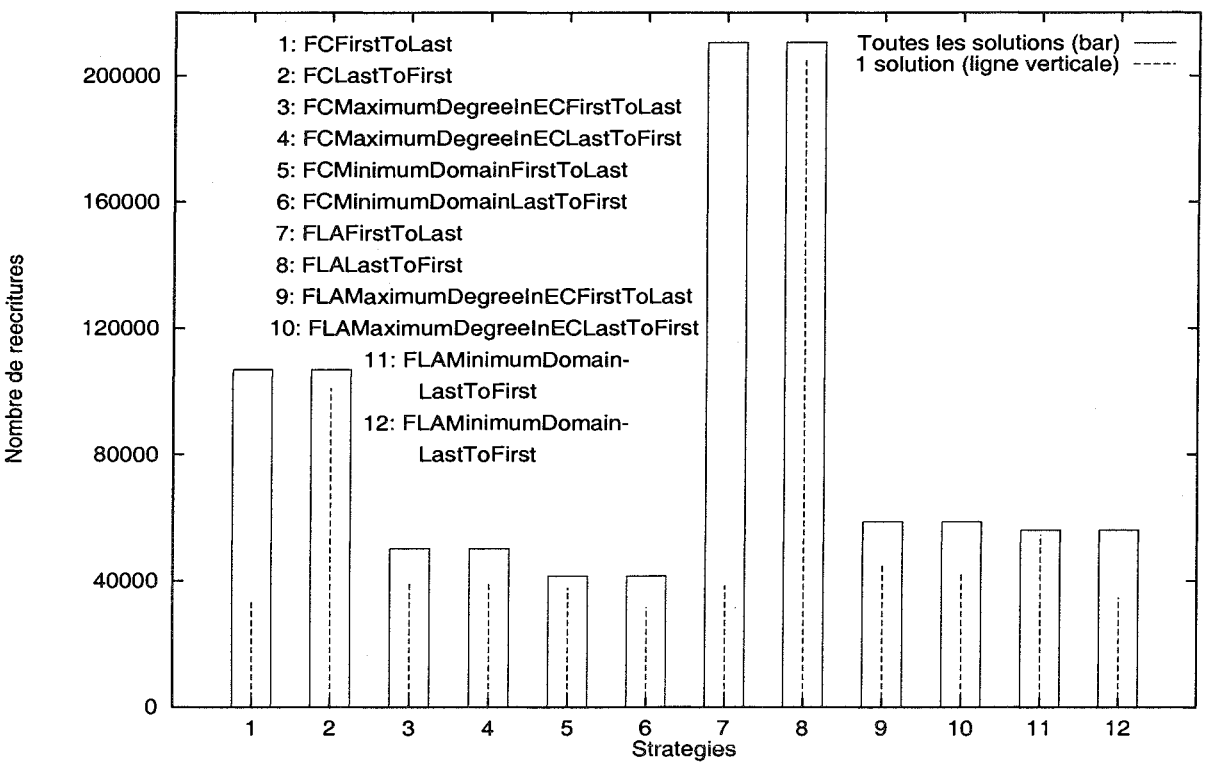

FIG. 3.30 – *Résolution du puzzle* $SEND + MORE = MONEY$

La figure 3.31 présente le comportement de la meilleure stratégie que nous avons utilisée pour la résolution du problème des N reines : `FCMinimumDomainFirstToLast`. Nous pouvons observer que lorsque le problème est insatisfaisable (le cas des 3 reines) les stratégies de recherche d'une solution et de toutes les solutions ont un comportement similaire car évidemment dans les deux cas il nous faut parcourir tout le espace de recherche.

La figure 3.32 montre les performances du système pour la résolution du problème des N reines en utilisant la même stratégie `FCMinimumDomainFirstToLastAll` pour trouver toutes les solutions. On peut rémarquer que la performance du système se détériore dans la mesure qu'il fait plus de calculs. Ce comportement est difficile à expliquer car on compte seulement le nombre de règles appliquées. Les règles qui ne sont qu'essayées ne sont pas prises en compte.

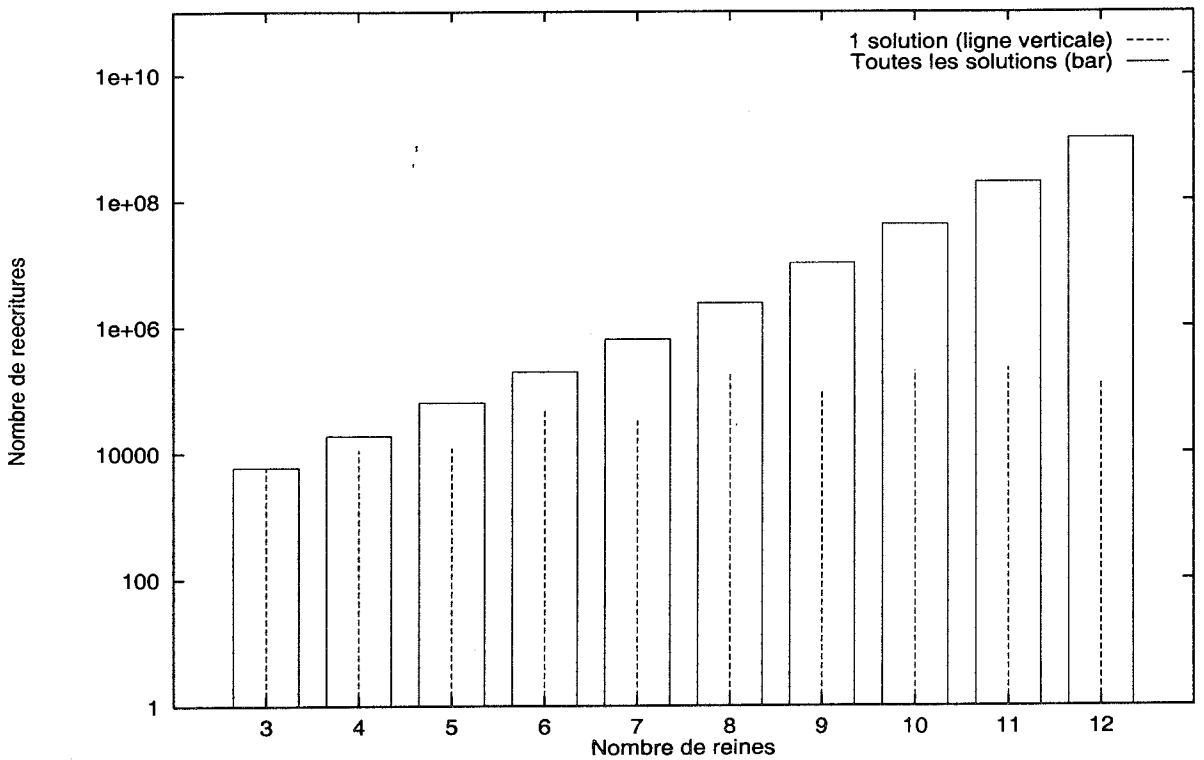

FIG. 3.31 – *Résolution du problème des N Reines*

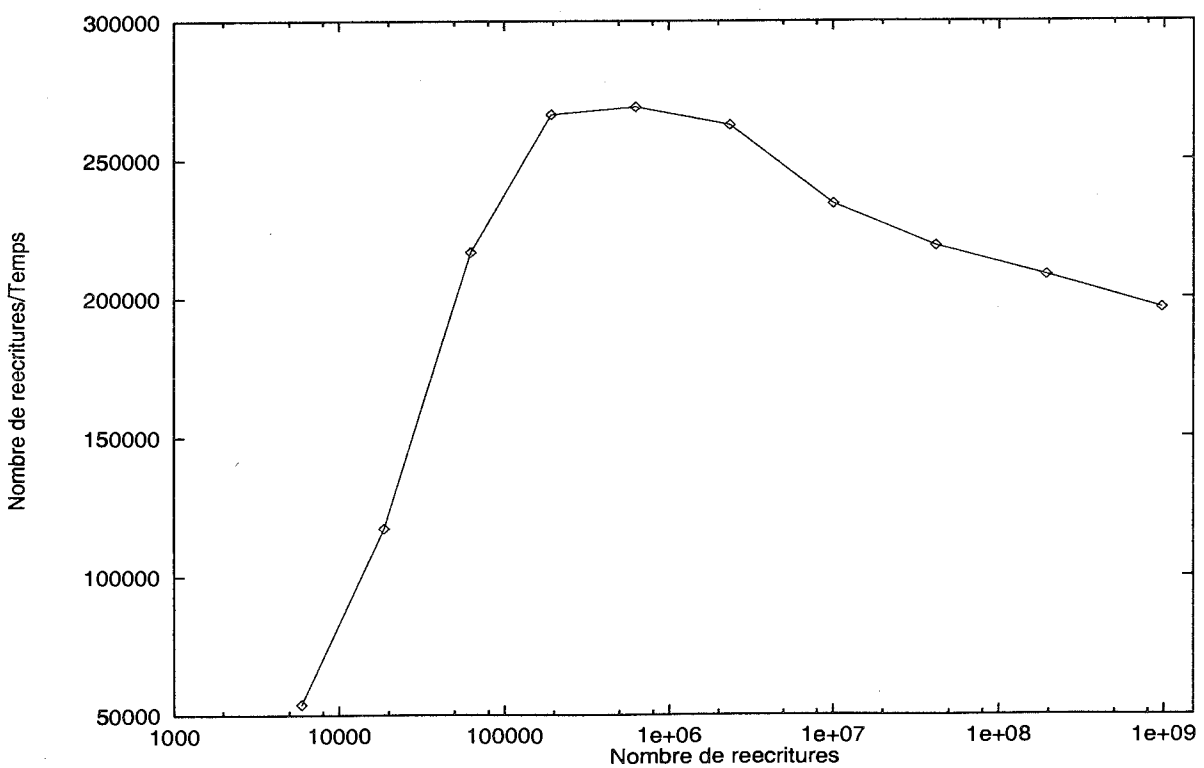

FIG. 3.32 – *Performances lors de la résolution du problème des N Reines*

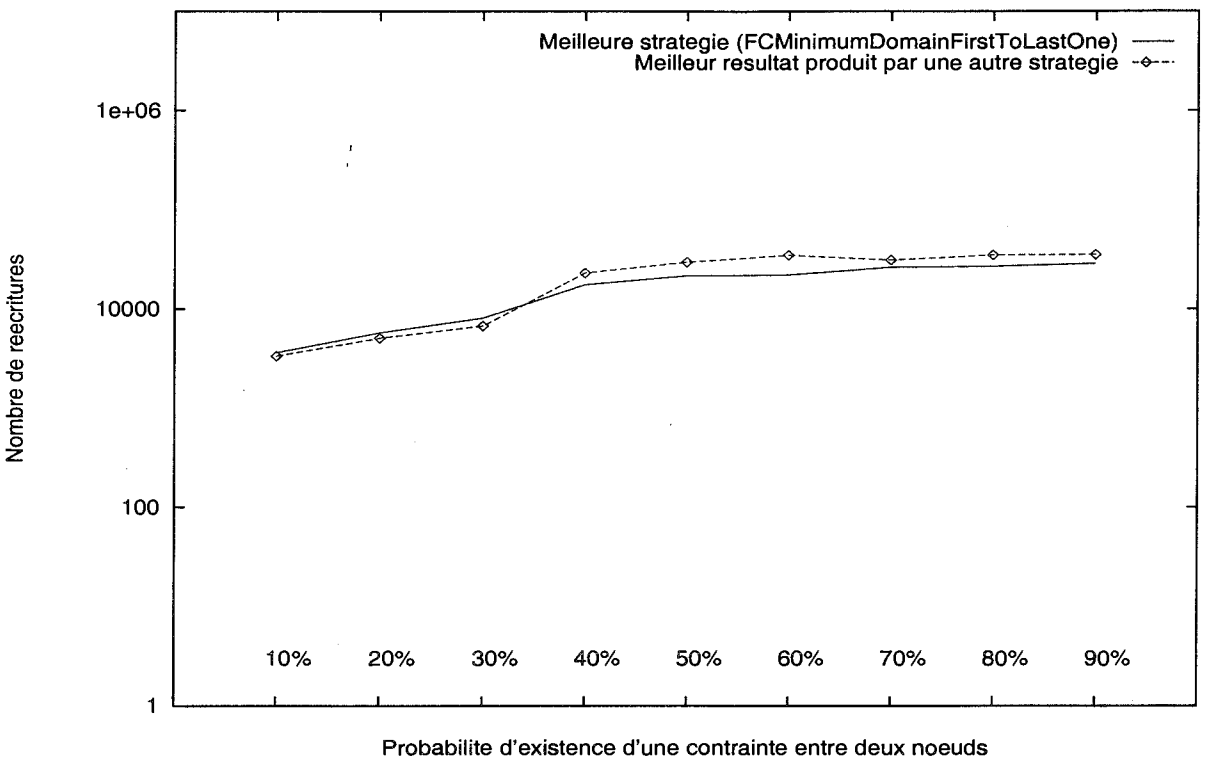

FIG. 3.33 – *Résolution du problème de coloriage de graphes : satisfaisabilité considérant 10 nœuds*

Finalement, les figures 3.33 et 3.34 présentent, respectivement, le comportement de la stratégie la plus performante que nous avons utilisée pour la résolution du problème de coloriage de graphes considérant 10 et 20 nœuds : `FCMinimumDomainFirstToLastOne`.

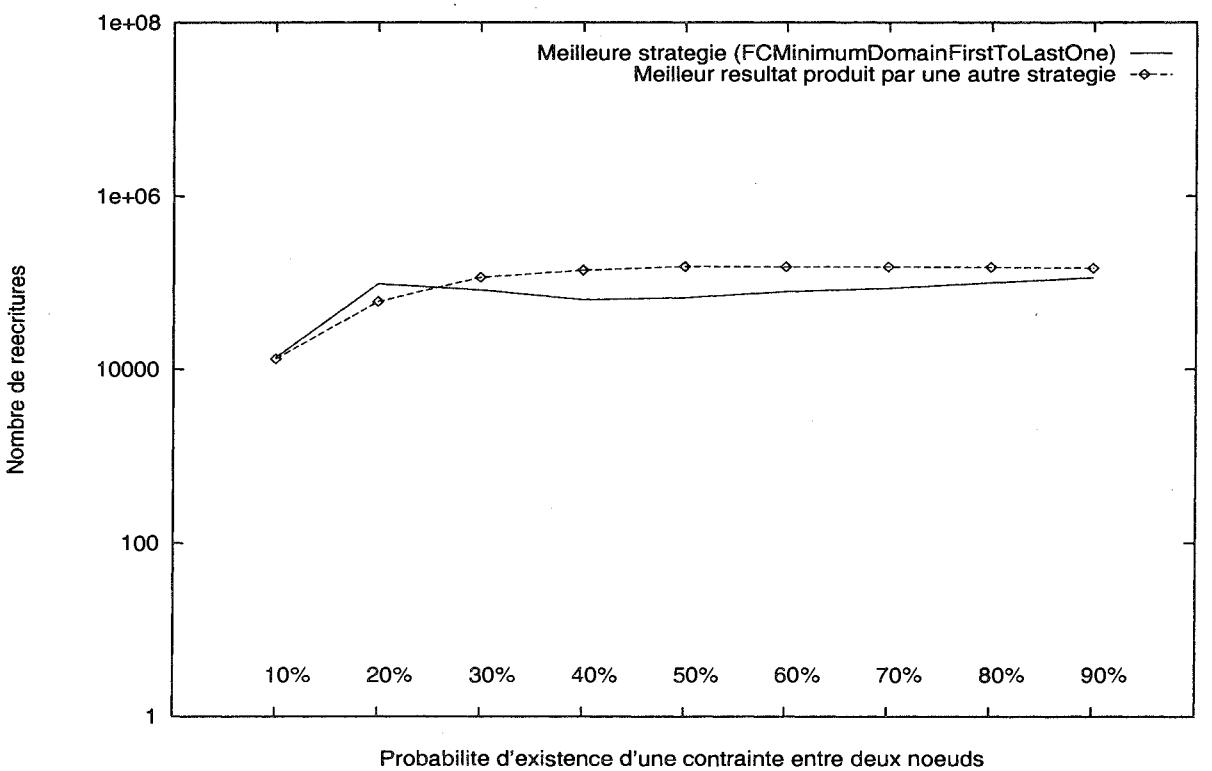

FIG. 3.34 – *Résolution du problème de coloriage de graphes : satisfaisabilité considérant 20 nœuds*

Chapitre 4

Problème d'optimisation

Sommaire

Lors de la modélisation mathématique d'un problème, on s'intéresse souvent non seulement à satisfaire un ensemble de relations entre des variables, mais aussi à trouver une valeur optimale pour une certaine fonction qui exprime la qualité des solutions. Cette extension des CSPs est appelée *problème d'optimisation de satisfaction de contraintes* (CSOP[22]). Ce problème est largement étudié par la communauté RO. Une des méthodes, peut-être la plus connue en optimisation, est la méthode du simplexe, qui permet de résoudre ce type de problèmes lorsque les variables portent sur des valeurs rationelles. La même méthode est intégrée dans des schémas de relaxation pour résoudre des problèmes sur les entiers, des problèmes qui sont étudiés sous le nom de programmation linéaire entière. D'autre part, la communauté de résolution de contraintes a attaqué ce problème en se basant sur une technique très utilisée par les communautés IA et RO : séparation-évaluation[23].

Dans ce chapitre, nous rappelons les approches de programmation linéaire entière et de contraintes pour la résolution du problème d'optimisation sur de nombres entiers. Basé principalement sur l'approche contraintes, nous modélisons ensuite la résolution d'un CSOP sous la forme d'un système de calcul. Finalement, nous présentons quelques considérations liées à l'implantation de ce système.

22. Constraint Satisfaction Optimisation Problem.
23. On reprend ici la terminologie de Fages pour le terme anglais Branch and Bound [41].

4.1 Définitions

Basé sur la définition d'un CSP, on définit un CSOP de la manière suivante.

Définition 35 (CSOP) *Un CSOP est défini par un couple $\langle P, f\rangle$, où P est un CSP et f est une fonction d'optimisation qui associe à chaque solution de P une valeur de D.*

Définition 36 (Solution d'un CSOP) *Étant donné un CSOP $\langle P, f\rangle$, une solution de $\langle P, f\rangle$ est une application α de $\mathcal{X}$ vers D qui associe à chaque variable $x \in \mathcal{X}$ un élément de D tel que $\alpha(c_i^?)$ est vraie dans $\mathcal{D}$ pour toute contrainte $c_i^? \in P$. L'ensemble de toutes les solutions de P est défini par*

$$Sol_{\mathcal{D}}(\langle P, f\rangle) = \{\alpha \in \alpha_{\mathcal{D}}^{\mathcal{X}} | \alpha(c_i^?) = \mathbf{V} \quad \forall c_i^? \in P\}$$

La définition d'une solution d'un CSOP correspond exactement à celle d'une solution d'un CSP. La différence entre ces deux problèmes se base seulement sur la considération de la fonction d'optimisation qui définit la qualité des solutions dans le cas des CSOPs. La notion de qualité d'une solution est définie à partir d'une relation de pré-ordre $\preceq$ entre les valeurs de la fonction d'optimisation f et un critère d'optimisation : minimisation ou maximisation. Ainsi, on définit le concept de solution optimale de la manière suivante.

Définition 37 (Solution optimale d'un CSOP) *Étant donné un CSOP $\langle P, f\rangle$, une solution optimale α de $\langle P, f\rangle$ est une solution telle que*

$$\forall \beta \in Sol_{\mathcal{D}}(P) : \beta(f) \preceq \alpha(f)$$

où $\preceq$ est une relation de pré-ordre définie pour le critère d'optimisation considéré. L'ensemble de toutes les solutions optimales est désigné par $Sol_{\mathcal{D}}^*$.

Le problème d'optimisation consiste à déterminer l'ensemble de solutions optimales de $\langle P, f\rangle$. En pratique, le problème d'optimisation est normalement réduit à trouver une seule solution optimale et éventuellement montrer qu'il existe de solutions optimales alternatives.

4.2 Approche de programmation linéaire : séparation-évaluation

Dans les années quarante, Dantzig propose une technique, connue comme la méthode du simplexe, pour la résolution du problème d'optimisation où la fonction et les contraintes sont linéaires et portent sur des variables rationelles [34]. L'étude de ce type de problème d'optimisation est aujourd'hui connue sous le nom de programmation linéaire. Malheureusement, quand on impose la contrainte additionnelle que les variables peuvent prendre seulement des valeurs entières on change complètement la nature du problème et la méthode du simplexe ne peut pas être appliquée. Cependant, il existe plusieurs techniques spécifiques pour attaquer ce type de problèmes. Parmi elles, on peut citer les techniques appelées d'énumération implicite [54] ou les techniques, très utilisées, appelées plans de coupure [102]. Dans [9], Bockmayr et Kasper présentent une formalisation de cette dernière technique et de l'approche contraintes dans un cadre uniforme. Cependant, la technique peut-être la plus utilisée par la communauté ILP est appelée séparation-évaluation, laquelle intégre la méthode du simplexe dans des schémas de relaxation qui résolvent une séquence de problèmes sans considérer les contraintes qui imposent des valeurs entières pour les variables jusqu'à ce que l'on obtient une solution entière. Dans cette section, on présente une version de la technique de séparation-évaluation.

La méthode de séparation-évaluation fut proposée par Land et Doig au début des années soixante [74]. Au cours du temps, plusieurs variantes ont été introduites. Nous présentons, à travers un exemple, une version couramment utilisée.

On considère un problème d'investissement : il y a quatre projets, dont leur revenus sont estimés à 8, 11, 6 et 4, respectivement, exprimés dans une certaine unité. L'investissement nécessaire pour leur développement est de 5, 7, 4 et 3, respectivement, et on dispose d'un capital de 14 unités. Le problème consiste à choisir la combinaison de projets qui maximise les bénéfices. Ce problème est un cas particulier du problème de recouvrement, très étudié par la communauté mathématique. Sa modélisation de base consiste à utiliser des variables binaires (variables 0/1) qui sont associées à la décision d'entreprendre (valeur 1) ou non (valeur 0) un projet. Le modèle sous la forme d'un CSOP est présenté ci-dessous

- l'ensemble de variables $\mathcal{X} = \{x_1, x_2, x_3, x_4\}$;
- les valeurs prises par les variables $\forall i \in [1, \ldots, 4] : \; x_i \in \{0, 1\}$;
- l'ensemble de contraintes $C = \{5x_1 + 7x_2 + 4x_3 + 3x_4 \leq^? 14\}$;
- la fonction d'optimisation $f = 8x_1 + 11x_2 + 6x_3 + 4x_4$.

Afin de mieux visualiser le problème de programmation linéaire entière, on peut transformer chaque contrainte $x_i \in \{0, 1\}$ comme suit

$$\forall\, x_i : \; x_i \geq^? 0 \;\wedge\; x_i \leq^? 1 \;\wedge\; x_i \in \mathbb{N}$$

On voit maintenant que si on enlève les contraintes $x_i \in \mathbb{N}$ on obtient un problème de programmation linéaire qui peut être résolu par la méthode du simplexe. Cette simplification est appelée relaxation, une technique très utilisée par la communauté RO pour attaquer les problèmes de très haute complexité.

La figure 4.1 présente graphiquement l'évolution de la procédure de séparation-évaluation. La signification de cet arbre est la suivante.

- En éliminant les contraintes qui imposent des valeurs entières pour les variables on obtient une relaxation du problème et en appliquant la méthode du simplexe, on obtient une valeur pour $f = 22$. On sait donc que la solution optimale du problème initial ne peut pas être supérieure à 22, on connaît une borne supérieure pour la fonction f. Malheureusement, la variable x_3 est la seule à ne pas avoir une valeur entière dans la solution obtenue, donc on crée deux sous-problèmes en ajoutant les contraints $x_3 = 0$ et $x_3 = 1$, respectivement.
- En appliquant la méthode du simplexe à chaque sous-problème on obtient deux solutions, aucune d'entre elles donnant des valeurs entières pour toutes les variables. Néanmoins, on sait maintenant que la fonction f ne peut pas prendre une valeur supérieure à 21, car tous les coefficients des variables dans la fonction f sont entiers. Vu que la branche de droite a une meilleure valeur pour la fonction f on décide de créer deux nouveaux sous-problèmes en y ajoutant les contraintes $x_2 = 0$ et $x_2 = 1$, respectivement.
- En appliquant toujours la méthode du simplexe à la relaxation de chaque sous-problème on obtient deux solutions, une d'entre elles maintenant donnant des valeurs entières pour toutes les variables. Maintenant, on sait donc que la valeur 18 est une borne inférieure pour la fonction f. En plus, on sait que tous les sous-problèmes générés à partir de cette branche nous donneront des valeurs inférieures pour la fonction f, vu qu'en ajoutant des contraintes on contraint encore plus l'espace de solutions, et donc il ne faut plus les considérer. Néanmoins, il y a encore des branches qui pourraient nous donner des meilleures valeurs. On choisit le nœud qui a une valeur pour $f = 21,8$ et on ajoute les contraintes $x_1 = 0$ et $x_1 = 1$ pour créer encore deux sous-problèmes.

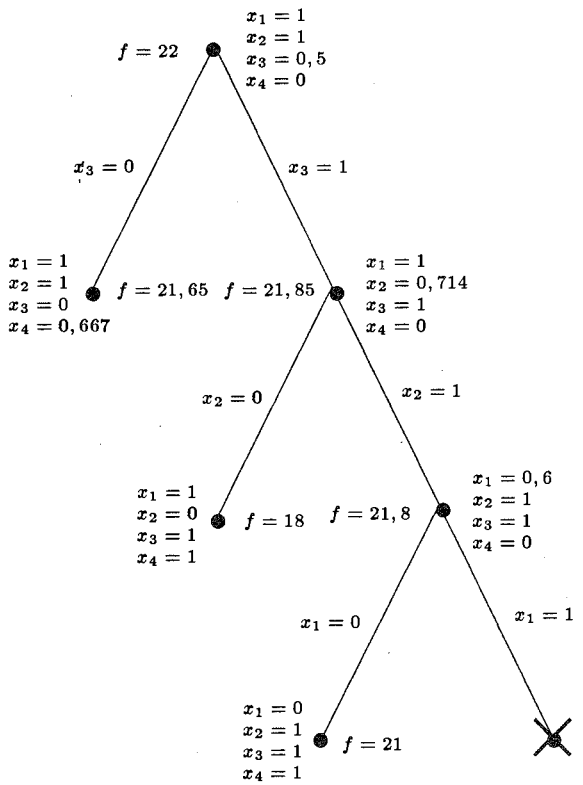

FIG. 4.1 – *Résolution de CSOPs par la technique de séparation-évaluation*

- La méthode du simplexe vérifie qu'un des sous-problèmes n'a pas de solution et l'autre donne des valeurs entières pour toutes les variables et la valeur de la fonction f, 21, est supérieure à sa valeur actuelle. On met à jour donc la borne inférieure. Le sous-problème qui nous reste à explorer pourrait nous donner dans le meilleur cas une valeur pour f égale à celle déjà obtenue, donc la solution $x_1 = 0$, $x_2 = 1$, $x_3 = 1$ et $x_4 = 1$ est la solution optimale.

Il est intéressant de noter qu'à partir de la solution du problème initial relaxé, $x_1 = 1$, $x_2 = 1$, $x_3 = 0,5$ et $x_4 = 0$, on pourrait penser à arrondir la valeur de la variable x_3 afin d'obtenir une solution avec toutes les variables entières. Cependant, si on arrondit $x_3 = 0$ on obtient une solution dont la valeur de f est égale à 19, c'est-à-dire inférieure à la solution optimale trouvée par la technique de séparation-évaluation. Plus grave encore, si on arrondit $x_3 = 1$, on obtient une affectation de variables qui viole la contrainte $5x_1 + 7x_2 + 4x_3 + 3x_4 \leq^? 14$ est donc cette affectation n'est pas une solution.

Nous avons présenté l'essentiel de la méthode. Il existe plusieurs critères pour choisir la variable à fixer. Quand on traite des problèmes où les variables sont restreintes à prendre des valeurs entières mais non seulement 0 ou 1, on change un peu les contraintes qui sont ajoutées : si dans une solution on obtient, par exemple, une valeur 4, 7 pour une variable x qui est restreinte à prendre des valeurs entières seulement, on crée deux sous-problèmes en ajoutant les contraintes $x \leq^? 4$ et $x \geq^? 5$. Du fait que la taille de l'arbre de recherche construit dépend du nombre de variables entières et non du nombre de contraintes, la méthode de séparation-évaluation a des bonnes performances quand il s'agit de problèmes à un petit nombre de variables, dans le cas contraire la méthode devient vite impraticable.

4.3 Approche contraintes

Même si le problème d'optimisation a été étudié depuis quelques années par les communautés CSP et CLP, il existe très peu de travaux qui décrivent formellement les techniques de résolution utilisées. Le travail de Bockmayr et Kasper, où est présenté un cadre unifié pour le traitement des CSOPs du point de vue de l'ILP, d'une part, et des contraintes, d'autre part, nous semble être la meilleure référence disponible sur le sujet [9]. Tsang [107] et Fages [41] décrivent aussi l'approche contraintes.

Dans cette section, on présente deux approches proposées à partir de la résolution systématique d'un ensemble de contraintes. La première, proposée par la communauté CSP, se base sur une extension de l'algorithme de *Backtracking*: avant d'étendre une affectation partielle lors de l'énumération des variables, une estimation de la fonction d'optimisation est réalisée et comparée avec une variable globale qui stocke la meilleure solution courante afin d'éliminer des branches de l'arbre d'énumération qui ne peuvent pas améliorer la solution courante. La deuxième, proposée par la communauté CLP, est basée sur la résolution de CSPs dynamiques : à partir d'une solution possible initiale, on essaie de l'améliorer par l'ajout d'une nouvelle contrainte qui impose une meilleure borne pour la fonction d'optimisation.

4.3.1 Résolution de CSOPs basée sur le *Backtracking*

Afin d'expliquer cette approche, on considère l'exemple ci-dessous lequel s'inspire de [107]. Soient

- l'ensemble de variables $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5\}$;
- les valeurs prises par les variables $D_{x_1} = \{4, 5\}$, $D_{x_2} = \{3, 5\}$, $D_{x_3} = \{3, 5\}$, $D_{x_4} = \{2, 3\}$ et $D_{x_5} = \{1, 3\}$;
- l'ensemble de contraintes $C = \{x_1 \neq^? x_2, x_2 =^? x_3, x_3 >^? x_4, x_4 \neq^? x_5\}$;
- la fonction d'optimisation $f = x_1 + x_2 + x_3 + x_4 + x_5$.

Le problème consiste à déterminer l'affectation des variables qui maximise la fonction d'optimisation f.

La description informelle de cet algorithme est la suivante.

- Une variable globale, appelée *borne inférieure*, est initialisée à $-\infty$. Cette variable stocke la valeur de la fonction d'optimisation évaluée à partir de la meilleure solution possible courante.
- On définit une fonction de majoration qui associe à chaque affectation partielle des variables une valeur en D. Cette fonction est définie à partir de la fonction d'optimisation
  - toute variable instanciée par l'affectation partielle est remplacée dans la fonction d'optimisation par sa valeur ;
  - toute variable non-instanciée par l'affectation partielle est remplacée dans la fonction d'optimisation par sa meilleure valeur possible vis-à-vis du critère d'optimisation.
- Un algorithme du style *Backtracking* est appliqué pour trouver une solution possible de C, sauf qu'avant d'étendre une affectation partielle par l'instanciation d'une nouvelle variable, la fonction de majoration est calculée : si cette évaluation de la fonction de majoration est plus petite que la valeur courante de la variable globale *borne inférieure*, alors le sous-arbre considéré est éliminé.

- Chaque fois qu'une nouvelle solution possible est trouvée, la fonction f est évaluée: si sa valeur est supérieure à celle de la *borne inférieure* courante cette solution devient la meilleure solution courante et l'évaluation de f devient la *borne inférieure* courante.
- Une fois que tout l'arbre d'énumération a été exploré, la solution possible courante correspond à la solution optimale du problème.

L'application de cette procédure pour la résolution de l'exemple précédent est présentée graphiquement dans la figure 4.2.

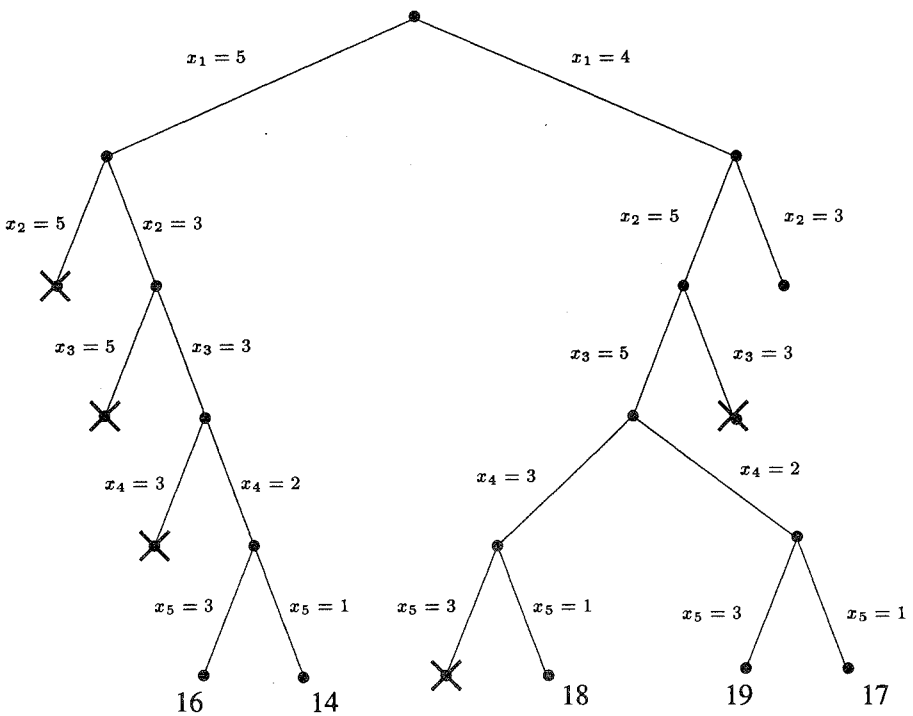

FIG. 4.2 – *Résolution de CSOPs en utilisant le* Backtracking

L'explication de cette figure est la suivante.

- La variable globale *borne inférieure* est initialisée à $-\infty$.
- L'algorithme cherche une solution possible en énumérant les variables dans l'ordre x_1, x_2, x_3, x_4, x_5 et en les instanciant à partir de la plus grande valeur qu'elles peuvent prendre.
- Les nœuds cochés dénotent une instanciation des variables qui viole une contrainte et donc le sous-arbre est éliminé (par exemple, l'instanciation partielle $x_1 \mapsto 5$, $x_2 \mapsto 5$).
- Quand la première solution possible $x_1 = 5$, $x_2 = 3$, $x_3 = 3$, $x_4 = 2$ et $x_5 = 3$ est trouvée, la fonction d'optimisation est évaluée à 16 et cette valeur devient la nouvelle valeur de la variable *borne inférieure*.
- Chaque fois que l'on trouve une meilleure solution possible, par rapport à la fonction d'optimisation, la variable *borne inférieure* est mise à jour. Ainsi, quand la solution possible $x_1 = 4$, $x_2 = 5$, $x_3 = 5$, $x_4 = 2$ et $x_5 = 3$ est trouvée, la fonction d'optimisation est évaluée à 19 et la variable *borne inférieure* est mise à jour.
- Quand le nœud correspondant à l'affectation partielle $x_1 = 4$ et $x_2 = 3$ est atteint, la fonction de majoration est évaluée à 18 et le sous-arbre est éliminé car à partir de cette affectation partielle on ne peut pas trouver une meilleur solution que la solution courante.

- Après l'élimination du sous-arbre du nœud $x_1 = 4$ et $x_2 = 3$, tout l'arbre d'énumération a été exploré, donc la solution optimale est $x_1 = 4$, $x_2 = 5$, $x_3 = 5$, $x_4 = 2$ et $x_5 = 3$.

Évidemment, toutes les considérations concernant l'algorithme de *Backtracking*, utilisé pour la résolution de CSPs, sont aussi valables dans le cas des CSOPs. Il faut cependant ajouter que l'estimation de la *borne inférieure* initiale aura une grande influence sur la forme de l'arbre d'énumération exploré car elle nous permettra d'éliminer différentes branches selon sa valeur.

4.3.2 Résolution de CSOPs basée sur des CSPs dynamiques

Un type particulier de CSPs, appelé CSP dynamique, consiste à résoudre des problèmes où les contraintes sont ajoutées ou retirées dynamiquement durant le processus de résolution [35, 112]. Ces problèmes se produisent souvent pour des applications pratiques où, une fois qu'une solution a été trouvée, les changements dans l'environnement entraînent des modifications de l'ensemble de contraintes. Normalement, on a intérêt à ne pas résoudre à nouveau le problème mais on cherche plutôt à dériver une nouvelle solution à partir de la solution courante. L'approche que l'on présente dans cette section se situe dans ce cadre.

La méthode utilise une borne inférieure pour trouver une solution optimale. On résout une séquence de CSPs qui donnent successivement des meilleures solutions possibles par rapport à la fonction d'optimisation. Plus précisément, et en considérant toujours un problème de maximisation, on calcule une solution $\alpha \in Sol_{\mathcal{D}}(P)$ et on ajoute la contrainte $f >^? \alpha(f)$ qui contraint l'ensemble de solutions possibles à celles qui donnent des meilleures valeurs pour la fonction d'optimisation[24]. Si après l'ajout d'une contrainte de ce type le problème devient insatisfaisable, alors la dernière solution possible trouvée est optimale.

Lors de la recherche d'une nouvelle solution, le calcul peut s'effectuer par des retours arrières en conservant la structure de l'arbre d'énumération courant ou par itération en développant un nouvel arbre. En présence d'heuristiques, la deuxième alternative a l'avantage de développer un nouvel arbre d'énumération en suivant une stratégie adaptée au but augmenté de la contrainte $f >^? \alpha(f)$ [41].

La règle ci-dessous exprime le pas réalisé pour trouver une nouvelle solution du CSOP $\langle P, f \rangle$

$$\langle P, f \rangle \rightarrow \langle P \wedge f >^? \alpha(f), f \rangle \textbf{ if } \alpha \in Sol(P)$$

Si on suppose que le CSP initial P est satisfaisable, l'itération de cette règle nous donnera une séquence de solutions possibles du CSOP. Dès que l'on obtient un problème insatisfaisable la dernière solution possible calculée correspond à la solution optimale.

Cette approche est un cas particulier d'un CSP dynamique où l'on résout une séquence de problèmes satisfaisables jusqu'à trouver un problème insatisfaisable. On peut penser aussi à réaliser une recherche dichotomique sur les valeurs prises par la fonction d'optimisation. Pour cela il nous faut des bornes inférieure et supérieure pour les valeurs prises par la fonction d'optimisation et à chaque étape on essaie de réduire l'intervalle des valeurs possibles par une analyse dichotomique.

De même que les techniques hybrides sont davantage utilisées que l'algorithme de *Backtracking* pour résoudre des CSPs, cette deuxième approche basée sur des CSPs dynamiques est largement préférée par rapport à la première basée sur le *Backtracking* pour résoudre des CSOPs : l'utilisation des contraintes est beaucoup plus active en appliquant l'approche basée sur des CSPs dynamiques.

24. Bockmayr et Kasper travaillent avec des fonctions f qui prennent toujours des valeurs entières et donc ils ajoutent la contrainte $f \geq^? \alpha(f) + 1$.

4.4 Système de calcul pour la résolution de CSOPs

Dans cette section, nous présentons une formalisation de la résolution de CSOPs basée principalement sur la résolution de CSPs dynamiques et en ajoutant une gestion explicite des bornes comme on le fait dans les techniques de séparation-évaluation en ILP.

Nous commençons par définir la forme basique qui représente le résultat qui nous désirons obtenir et ensuite, à partir des techniques existantes pour la résolution de CSOPs, on propose des règles de réécriture et des stratégies qui nous permettent de décrire le comportement de ces techniques. Sans perte de généralité, on considère le problème de minimisation de la fonction d'optimisation.

4.4.1 Forme basique d'un CSOP

La forme basique d'un CSOP est définie de façon telle de nous permettre d'éliminer de solutions mais en préservant toujours l'ensemble de solutions optimales.

Définition 38 (Forme basique d'un CSOP) *$\langle P', f\rangle$ est une* forme basique *pour un CSOP $\langle P, f\rangle$ si P' est une conjonction de formules de la forme*

$$\bigwedge_{i\in I}(x_i \in^? D_{xi}) \wedge \bigwedge_{j\in J}(x_j =^? v_j) \wedge \bigwedge_{m\in M} c_m^?$$

où $v_j \in D$ est une valeur quelconque dans le domaine d'interprétation consideré et $c_m^?$ est une contrainte élémentaire, telle que

- $\forall i, j \in I \cup J : x_i \neq x_j$
- $\forall i \in I : D_{xi} \neq \emptyset$
- $\forall m \in M : \mathcal{V}ar(c_m^?) \subseteq \{x_i | i \in I \cup J\}$
- $Sol_{\mathcal{D}}(P') \subseteq Sol_{\mathcal{D}}(P)$
- $Sol^*_{\mathcal{D}}(P') = Sol^*_{\mathcal{D}}(P)$

De la même façon comme nous le faisons pour les CSPs, il est possible d'associer à tout CSOP en forme basique une *affectation basique* obtenue en affectant chaque variable x_j dans les contraintes d'égalité par la valeur associée v_j et chaque variable x_i dans les contraintes d'appartenance par une valeur quelconque dans l'ensemble des valeurs D_{x_i}.

Une solution optimale d'un CSOP $\langle P, f\rangle$ est définie comme étant une solution du CSP P satisfaisant la condition d'optimalité.

Définition 39 (Solution optimale d'un CSOP) *Une* solution optimale *d'un CSOP $\langle P, f\rangle$ est une affectation basique α satisfaisant toutes les contraintes dans P et telle que*

$$Sol_{\mathcal{D}}(P \wedge f <^? \alpha(f)) = \emptyset$$

Les techniques de résolution de CSOPs que nous allons présenter se basent sur l'ajout de contraintes afin de réduire l'ensemble de solutions d'un CSOP jusqu'à ce qu'il est exactement égal à l'ensemble de solutions optimales

$$Sol_{\mathcal{D}}(\langle P, f\rangle) = Sol^*_{\mathcal{D}}(\langle P, f\rangle)$$

4.4.2 Règles de transformation

Afin de gérer explicitement le concept de borne, on étend la définition d'un CSOP de la manière suivante.

Définition 40 (CSOP avec bornes) *Un CSOP avec bornes est défini par un 4-uplet $\langle P, f, lb, ub\rangle$, où P est un CSP, f est une fonction d'optimisation et lb et ub représentent la borne inférieure et la borne supérieure des valeurs prises par f, respectivement.*

Cette extension est purement syntaxique. L'idée derrière cette extension est de stocker dans la borne inférieure la valeur la plus petite que la fonction d'optimisation peut prendre sans considérer les contraintes (normalement on obtient cette valeur par une minoration de la fonction). La borne supérieure stocke la valeur de la fonction d'optimisation évaluée dans la meilleure solution courante. L'existence des bornes inférieure et supérieure pour la fonction d'optimisation est une condition nécessaire et suffisante pour la terminaison des règles présentées dans cette section.

La transformation fondamentale réalisée par l'approche contraintes, basée sur la résolution de CSPs dynamiques, peut être exprimée par la règle **DecreaseUpperBound** présentée dans la figure 4.3.

[DecreaseUpperBound]
$\langle P, f, lb, ub\rangle$
$\Rightarrow$
$\langle P \wedge f \leq^? \alpha(f), f, lb, \alpha(f)\rangle$
if $\alpha \in Sol_{\mathcal{D}}(P \wedge f <^? ub)$

FIG. 4.3 – *Règle pour la mise à jour de la borne supérieure*

Cette règle vérifie d'abord s'il est possible de trouver une solution α du CSP $P \wedge f <^? ub$, et dans ce cas elle évalue la fonction d'optimisation à partir de la solution α et met à jour la borne supérieure. La borne supérieure représente donc la meilleure valeur obtenue jusqu'à présent. La borne inférieure n'est jamais prise en compte par cette approche. La seule différence entre cette règle et celle utilisée normalement dans l'approche contraintes est que nous gardons toujours un CSP satisfaisable lors de son application. Cette caractéristique vient du fait que lors que l'on fait un pas de réécriture on maintient toujours le CSP en forme résolue, ce qui ne serait pas le cas si on ajoute une contrainte qui rend le problème insatisfaisable.

Lemme 5 *La règle* **DecreaseUpperBound** *est correcte et complète (i.e. elle préserve $Sol^*_{\mathcal{D}}$).*

Preuve: Cette règle considère en entrée un CSP P dont son ensemble de solutions est $Sol_{\mathcal{D}}(P)$ et son ensemble de solutions optimales est $Sol^*_{\mathcal{D}}(P)$. Comme résultat on obtient un ensemble de contraintes $P \wedge f \leq^? \alpha(f)$ où α correspond à une solution de $P \wedge f <^? ub$. Évidemment, $Sol_{\mathcal{D}}(P \wedge f \leq^? \alpha(f)) \subset Sol_{\mathcal{D}}(P)$ et $Sol^*_{\mathcal{D}}(P \wedge f \leq^? \alpha(f)) \subset Sol^*_{\mathcal{D}}(P)$. C'est-à-dire cette règle est correcte car en ajoutant une contrainte à P on n'ajoute pas de solutions et on n'ajoute donc pas de solutions optimales. Cette règle est appliquée si elle vérifie la condition $Sol_{\mathcal{D}}(P \wedge f <^? ub) \neq \emptyset$. Dans ce cas, forcement $Sol_{\mathcal{D}}(P \wedge f \leq^? \alpha(f)) \neq \emptyset$, c'est-à-dire le CSP P a toujours au moins une solution. Donc la règle **DecreaseUpperBound** est complète car même si l'ensemble de solutions est réduit, l'ensemble de solutions optimales n'est pas modifié.

□

Lemme 6 *L'application répétée de la règle* **DecreaseUpperBound** *termine.*

Preuve : Nous considérons la cardinalité de l'ensemble de solutions comme notre mesure de complexité. À chaque application de la règle **DecreaseUpperBound**, on ajoute la contrainte $f \leq^? \alpha(f)$ à P, où α correspond à une solution de $P \wedge f <^? ub$. La valeur $\alpha(f)$ devient aussi la nouvelle valeur de la borne supérieure ub. La prochaine application de cette règle essayera de trouver une solution à l'ensemble de contraintes $P \wedge f <^? ub$, considérant la nouvelle valeur de la borne supérieure ub. Cet ensemble de contraintes est équivalent à $P \wedge f \leq^? ub - 1$, c'est-à-dire on cherche une solution qui decroît la borne supérieure au moins d'une unité. De cette façon à chaque application de cette règle on réduit l'ensemble de solutions car on élimine au moins une solution : celle qui a été obtenue lors de la dernière application de cette règle. Donc, si on considère qu'il existe une borne inférieure pour la fonction d'optimisation l'application répétée de cette règle termine forcement.

□

Lemme 7 *Étant donné un CSOP $\langle P, f, lb, ub\rangle$ en forme basique, en appliquant la règle* **DecreaseUpperBound** *on obtient toujours un CSOP en forme basique.*

Preuve : Il est évident que cette règle laisse toujours le CSOP en forme basique car elle ajoute seulement la contrainte $f \leq^? \alpha(f)$ à P, où f est la fonction d'optimisation (on n'ajoute pas de variables) et α est une solution à $P \wedge f <^? ub$.

□

Théorème 2 *Étant donné un CSOP en forme basique, en utilisant la règle* **DecreaseUpperBound** *jusqu'à ce qu'elle ne puisse plus être appliquée on obtient une forme basique $\langle P, f, lb, ub\rangle$ telle que*

$$\forall \alpha \in Sol_{\mathcal{D}}(P) : Sol_{\mathcal{D}}(P \wedge f <^? \alpha(f)) = \emptyset$$

Preuve : Elle est directe en utilisant les trois lemmes précédents.

□

Ainsi, pour obtenir chaque solution optimale du CSOP $\langle P, f, lb, ub\rangle$ résultant de l'application répétée de la règle **DecreaseUpperBound**, il nous suffit de résoudre le CSP P : toute solution à P est alors une solution optimale.

Par ailleurs, si on veut réaliser une recherche dichotomique sur les valeurs prises par la fonction d'optimisation, on peut concevoir les deux règles de transformation suivantes.

La règle **DecreaseUpperBoundBySplitting**, présentée dans la figure 4.4, cherche une nouvelle valeur pour la borne supérieure ub qui réduit, au moins de moitié, l'intervalle des valeurs prises par la fonction d'optimisation f chaque fois qu'une nouvelle solution est trouvée.

[DecreaseUpperBoundBySplitting]
$\langle P, f, lb, ub\rangle$
$\Rightarrow$
$\langle P \wedge f \leq^? \alpha(f), f, lb, \alpha(f)\rangle$
if $\alpha \in Sol_{\mathcal{D}}(P \wedge f <^? \frac{lb+ub}{2})$

FIG. 4.4 – *Règle pour la mise à jour de la borne supérieure par* splitting

À différence de la règle **DecreaseUpperBound**, la non-application de la règle **DecreaseUpperBoundBySplitting** n'implique pas que l'on a trouvé la solution optimale. Il peut y

avoir encore des solutions qui donnent des valeurs pour la fonction d'optimisation entre sa borne supérieure courante et la valeur correspondant à la moitié de l'intervalle. La mise à jour de la borne inférieure est donc faite par l'application de la règle ci-dessous.

[IncreaseLowerBoundBySplitting]
$\langle P, f, lb, ub\rangle$
$\Rightarrow$
$\langle P \wedge f \geq^? \lceil \frac{lb+ub}{2} \rceil, f, \lceil \frac{lb+ub}{2} \rceil, ub\rangle$
if $lb \neq ub$ **and** $Sol_{\mathcal{D}}(P \wedge f <^? \frac{lb+ub}{2}) = \emptyset$

FIG. 4.5 – *Règle pour la mise à jour de la borne inférieure par* splitting

La règle **IncreaseLowerBoundBySplitting** vérifie donc qu'il n'y a pas de solution qui donne une valeur pour la fonction d'optimisation plus petite que la valeur correspondant à la moitié de l'intervalle.

Évidemment, la taille de cette réduction peut être, en général, une proportion quelconque de la taille de l'intervalle des valeurs prises par la fonction d'optimisation. En fait, la première approche peut être vue comme un cas particulier de cette dernière. Voyons en détail cette situation. Pour simplifier l'analyse, on suppose que la fonction f prend uniquement des valeurs entières. Ainsi, la contrainte $f <^? ub$, que l'on ajoute dans la règle **DecreaseUpperBound**, peut aussi être exprimée comme $f \leq^? ub - 1$, et on voit bien ici que chaque application de cette règle réduit l'intervalle en un taux de $\frac{1}{ub-lb}$. Donc, il est clair que la règle **DecreaseUpperBound** est un cas particulier de la règle **DecreaseUpperBoundBySplitting**, où le taux de valeurs que l'on essaie d'éliminer est variable.

Lemme 8 *Les règles* **DecreaseUpperBoundBySplitting** *et* **IncreaseLowerBoundBySplitting** *sont correctes et complètes.*

Preuve: La règle **DecreaseUpperBoundBySplitting** considère en entrée un CSP P dont son ensemble de solutions est $Sol_{\mathcal{D}}(P)$ et son ensemble de solutions optimales est $Sol^*_{\mathcal{D}}(P)$. Comme résultat on obtient un ensemble de contraintes $P \wedge f \leq^? \alpha(f)$ où α correspond à une solution de $P \wedge f <^? \frac{lb+ub}{2}$. Évidemment, $Sol_{\mathcal{D}}(P \wedge f \leq^? \alpha(f)) \subset Sol_{\mathcal{D}}(P)$ et $Sol^*_{\mathcal{D}}(P \wedge f \leq^? \alpha(f)) \subset Sol^*_{\mathcal{D}}(P)$. C'est-à-dire cette règle est correcte car en ajoutant une contrainte à P on n'ajoute pas de solutions et on n'ajoute donc pas de solutions optimales. Cette règle est appliquée si elle vérifie la condition $Sol_{\mathcal{D}}(P \wedge f <^? \frac{lb+ub}{2}) \neq \emptyset$. Dans ce cas, on a forcement $Sol_{\mathcal{D}}(P \wedge f \leq^? \alpha(f)) \neq \emptyset$, c'est-à-dire le CSP P a toujours au moins une solution. Donc la règle **DecreaseUpperBoundBySplitting** est complète car même si l'ensemble de solutions est réduit, l'ensemble de solutions optimales n'est pas modifié.

La règle **IncreaseLowerBoundBySplitting** ajoute la contrainte $f \geq^? \lceil lb + (ub - lb)/2) \rceil$ à P et elle est donc correcte car en ajoutant une contrainte à P on n'ajoute pas de solutions et on n'ajoute donc pas de solutions optimales. Une des conditions d'application de cette règle est que $Sol_{\mathcal{D}}(P \wedge f <^? \frac{lb+ub}{2}) = \emptyset$, donc en ajoutant la contrainte $f \geq^? \lceil lb + (ub - lb)/2) \rceil$ à P on n'élimine aucune solution, alors la règle **IncreaseLowerBoundBySplitting** est complète.

□

Lemme 9 *L'application répétée des règles* **DecreaseUpperBoundBySplitting** *et* **IncreaseLowerBoundBySplitting** *termine.*

Preuve: Nous considérons l'intervalle de valeurs $[lb, \ldots, ub]$ prises par la fonction d'optimisation comme notre mesure de complexité. À chaque application de la règle **DecreaseUpperBoundBySplitting** on calcule une nouvelle solution α, on remplace la borne supérieure par $\alpha(f)$ et la borne inférieure reste inchangée. Étant donné que α est une solution à $P \wedge f <^? \frac{lb+ub}{2}$ on sait que forcement $\alpha(f) <^? \frac{lb+ub}{2}$, donc on réduit au moins de moitié l'intervalle de valeurs prises par la fonction d'optimisation. À chaque application de la règle **IncreaseLowerBoundBySplitting** on remplace la borne inférieure par $\lceil \frac{lb+ub}{2} \rceil$ et la borne supérieure reste inchangée. Cette mise à jour de la borne inférieure réduit donc de moitié l'intervalle de valeurs prises par la fonction d'optimisation. Puisque la fonction d'optimisation ne peut prendre qu'un nombre fini de valeurs, on peut en conclure que l'application répétée de ces deux règles termine.

□

Lemme 10 *Étant donné un CSOP $\langle P, f, lb, ub \rangle$ en forme basique, en appliquant les règles* **DecreaseUpperBoundBySplitting** *et* **IncreaseLowerBoundBySplitting** *on obtient toujours un CSOP en forme basique.*

Preuve: Il est évident que la règle **DecreaseUpperBoundBySplitting** laisse le CSOP en forme basique car elle ajoute seulement la contrainte $f \leq^? \alpha(f)$ à P, où f est la fonction d'optimisation (on n'ajoute pas de variables) et α est une solution à $P \wedge f <^? \frac{lb+ub}{2}$. Également, l'ajout de la contrainte $f \geq^? \lceil \frac{lb+ub}{2} \rceil$ par l'application de la règle **IncreaseLowerBoundBySplitting** laisse toujours le CSOP en forme basique.

□

Théorème 3 *Étant donné un CSOP en forme basique, en utilisant les règles* **DecreaseUpperBoundBySplitting** *et* **IncreaseLowerBoundBySplitting** *jusqu'à ce qu'elles ne puissent plus être appliquées on obtient une forme basique $\langle P, f, lb, ub \rangle$ telle que*

$$\forall \alpha \in Sol_{\mathcal{D}}(P) : Sol_{\mathcal{D}}(P \wedge f <^? \alpha(f)) = \emptyset$$

Preuve: Elle est directe en utilisant les trois lemmes précédents.

□

Donc, si on veut obtenir chaque solution optimale du CSOP $\langle P, f, lb, ub \rangle$ résultant de l'application répétée des règles **DecreaseUpperBoundBySplitting** et **IncreaseLowerBoundBySplitting** il nous suffit ensuite de résoudre le CSP P.

4.4.3 Stratégies d'optimisation

La première stratégie que l'on peut envisager est la simple itération de l'application de la règle **DecreaseUpperBound**, comme cela est réalisé par la stratégie ***MinSatToUnsat*** présentée dans la figure 4.6.

MinSatToUnsat =>
repeat* (*DecreaseUpperBound*)
end

FIG. 4.6 – *Stratégie d'optimisation par une approche* satisfaisabilité à insatisfaisabilité

À chaque application de la règle **DecreaseUpperBound**, la stratégie ***MinSatToUnsat*** décroît la borne supérieure, *ub*, jusqu'à ce qu'elle ne puisse plus être appliquée, c'est-à-dire un problème insatisfaisable est atteint. Pour cette raison nous l'appelons *satisfaisabilité à insatisfaisabilité.*

Pour réaliser une recherche dichotomique sur les valeurs prises par la fonction d'optimisation nous désignons la stratégie ***MinSplitting*** présentée dans la figure 4.7.

MinSplitting =>
repeat*(**first one** (*DecreaseUpperBoundBySplitting*,
IncreaseLowerBoundBySplitting)
end

FIG. 4.7 – *Stratégie d'optimisation par une approche* splitting

En utilisant cette stratégie, l'intervalle de valeurs prises par la fonction d'optimisation est réduit en ses deux extrêmes : quand la borne inférieure est égale à la borne supérieure la solution courante correspond à la solution optimale. Nous appelons cette approche *splitting.*

Il est important de noter une différence importante entre ces deux stratégies.

- La stratégie ***MinSatToUnsat***, en appliquant la règle **DecreaseUpperBound**, essaie d'éliminer, dans le cas général, un nombre constant des valeurs prises par la fonction f, toujours au moins une valeur. C'est-à-dire, le nombre de valeurs éliminées est constant mais la proportion du nombre de valeurs éliminées par rapport au nombre total de valeurs prises par la fonction d'optimisation est variable.
- La stratégie ***MinSplitting***, en appliquant les règles **DecreaseUpperBoundBySplitting** ou **IncreaseLowerBoundBySplitting**, essaie de réduire l'intervalle de valeurs prises par la fonction f à un taux constant, par exemple, 50%. C'est-à-dire, la proportion du nombre de valeurs éliminées par rapport au nombre total est constante mais le nombre de valeurs éliminées est variable.

Cette remarque aura une influence dans l'explication des comportements de ces deux heuristiques.

4.5 Implantation

Dans cette section, nous décrivons l'implantation du système de calcul conçu pour traiter des CSOPs. Nous décomposons les règles introduites dans la section 4.4 afin d'avoir des éléments encore plus atomiques, ce qui nous permettra de réutiliser quelques règles dans l'implantation des différents solveurs. On présente finalement un résumé des jeux d'essai réalisés afin de montrer les performances des solveurs développés.

4.5.1 Structure de données

Nous définissons l'objet représentant un CSOP comme étant un 4-uplet

$$CSOP[P, f, lb, ub]$$

Cette structure, définie dans la perspective de l'implantation, correspond donc exactement à celle présentée dans la formalisation des CSOPs.

Cette définition est exprimée en ELAN de la manière suivante

```
operators
global

CSOP@          : (set:tuple[csp,term,int,int])        csop;

end
```

4.5.2 Règles de transformation

Si on analyse en détail les trois règles et les deux stratégies introduites dans la section 4.4, on peut tirer quelques conclusions.

- Il y a des opérations générales qui se répètent : mise à jour des bornes, ajout de contraintes et résolution de CSPs.
- À chaque mise à jour d'une borne, les deux approches modifient l'ensemble de contraintes initial seulement par l'ajout d'une contrainte. Cela implique qu'à chaque itération le processus de calcul est relancé à partir du problème initial plus les contraintes qui ont été ajoutées. Tout le travail fait pour trouver une solution est perdu quant aux modifications des contraintes. Une alternative est de vérifier la consistance locale de l'ensemble de contraintes lors de la mise à jour des bornes.
- À chaque mise à jour d'une borne, les deux approches ajoutent une contrainte qui est, en général, n-aire. Afin d'éviter cette augmentation de la taille de l'ensemble de contraintes on pourrait penser à ajouter à l'ensemble de contraintes initial une contrainte du style

  $$x =^? f$$

  où x est une nouvelle variable du problème et ainsi, à chaque mise à jour des bornes, on devrait ajouter une contrainte unaire du style

  $$x <^? \alpha(f)$$

  L'avantage de gérer de cette façon l'ajout des contraintes vient du fait que les contraintes unaires sont éliminées une fois qu'elles sont traitées par les algorithmes de vérification de consistance et donc on n'augmente pas la taille de l'ensemble de contraintes.
- L'analyse dichotomique, qui est réalisée par l'approche *splitting*, réduit uniquement, à chaque application de la règle **DecreaseUpperBoundBySplitting**, la borne supérieure de l'intervalle des valeurs prises par la fonction d'optimisation. On pourrait penser à minorer la fonction d'optimisation à partir des ensembles réduits de valeurs prises par les variables et de cette façon mettre à jour également la borne inférieure.
- Les deux règles utilisées par la stratégie *MinSplitting*, **DecreaseUpperBoundBySplitting** et **IncreaseLowerBoundBySplitting**, sont appliquées dans des situations tout à fait opposées. Il faut donc seulement vérifier que, par exemple, la première ne peut pas être appliquée pour appliquer la deuxième sans vérifier la condition (qui est coûteuse).

Tenant en compte ces considérations, on propose l'ensemble de règles suivant.

- Un premier type de règles est défini afin de gérer la mise à jour des bornes inférieure et supérieure. On définit la règle `SetUpperBoundToMiddle` ci-dessous pour affecter provisoirement la borne supérieure par la valeur correspondant à la moitié de l'intervalle et de la même façon la règle `SetLowerBoundToMiddle` est définie pour affecter provisoirement la borne inférieure par la valeur correspondant à la moitié de l'intervalle. On définit également la règle `SetLowerBoundToUpperBound` qui affecte la borne inférieure par la valeur courante de la borne supérieure. À titre d'exemple, nous présentons la spécification de la règle `SetUpperBoundToMiddle`.

```
rules for csop
        x              : var;
        ub, lb, middle : int;
        P              : csp;
global

[SetUpperBoundToMiddle]
        CSOP[P,x,lb,ub]
        =>
        CSOP[P,x,lb,middle]
        if      lb != ub
        choose
        try
                if      (lb+lb)%2 == 0
                where   middle := ()(lb+ub)/2
        try
                where   middle := ()(lb+ub)/2 + 1
        end
end
```

- Les règles suivantes sont définies pour poster les contraintes qui sont ajoutées dynamiquement au cours du processus d'isolation des solutions optimales. La règle `PostLTUpperBound` ajoute la contrainte `1(*)x(+)0<?ub` à l'ensemble de contraintes en entrée[25].

```
rules for csop
        x                       : var;
        ub, lb                  : int;
        lmc, lec, EC, DC, store : list[constraint];
global

[PostLTUpperBound]
        CSOP[CSP[lmc,lec,EC,DC,store],x,lb,ub]
        =>
        CSOP[CSP[lmc,lec,1(*)x(+)0<?ub.EC,DC,store],x,lb,ub]
        if      lb < ub
        if      not occurs x in lec
end
```

De la même façon on définit la règle `PostLTEUpperBound` qui ajoute une contrainte du style `<=?` au lieu d'ajouter une contrainte `<?`.

25. COLETTE maintient toujours les deux termes d'une contrainte en forme linéaire, c'est-à-dire $x_1 \times a_1 + \ldots + x_e \times a_e + a_0$, où x_i est une variable telle que $\forall i, j \in [1, \ldots, e] : i \neq j \Leftrightarrow x_i \neq x_j$ et a_i est une constante non-négative $\forall i \in [0, \ldots, e]$. Cela explique la forme du terme gauche `1(*)x(+)0<?ub`.

- Afin de tirer des conclusions relatives à la borne inférieure, après le calcul d'une nouvelle borne supérieure, on définit la règle **EstimateLowerBound**. Chaque fois qu'une nouvelle solution est trouvée on minore la borne inférieure pour calculer la plus petite valeur que peut atteindre la fonction d'optimisation.

```
rules for csop
        x                       : var;
        ub, lb, newlb           : int;
        lmc, lec, EC, DC, store : list[constraint];
global

[EstimateLowerBound]
        CSOP[CSP[lmc,lec,EC,DC,store],x,lb,ub]
        =>
        CSOP[CSP[lmc,lec,EC,DC,store],x,newlb,ub]
        where   newlb   := ()MinOfTerm(1(*)x,lmc,lec,0)
end
```

- Finalement, pour chercher une solution d'un CSP, on définit la règle **GetNewUpperBound** qui applique une stratégie de résolution de CSPs. Dans l'exemple ci-dessous, on considère la stratégie **FCFirstToLastOne** définie dans la section 3.6.5, mais évidemment on pourrait utiliser une stratégie de résolution de CSPs quelconque.

```
rules for csop
        x, y            : var;
        ub, lb, newub   : int;
        P, Q            : csp;
        l               : list[constraint];
global

[GetNewUpperBound]
        CSOP[P,x,lb,ub]
        =>
        CSOP[P,x,lb,newub]
        where   Q :=(FCFirstToLastOne) P
        if      Q != Unsatisfiable
        where   (list[constraint]) y = newub.l := ()GetVarInHead(x,GetLEC(Q),nil)
end
```

Cette règle est appliquée seulement si le CSP en entrée est satisfaisable et dans ce cas elle récupère de la liste de contraintes d'égalité la contrainte contenant la variable **x** qui représente la fonction d'optimisation. La valeur de cette variable devient donc la nouvelle borne supérieure.

4.5.3 Stratégies d'optimisation

Maintenant que nous avons des règles plus atomiques et considérant des aspects relatifs à l'implantation, nous pouvons redéfinir les stratégies d'optimisation présentées dans la section 4.4.

La stratégie **MinSatToUnsat** implante l'approche que nous avons appelée *satisfaisabilité à insatisfaisabilité.*

```
stratop
global
        MinSatToUnsat : < csop -> csop >      bs;
end

[] MinSatToUnsat =>
PostLTEUpperBound ;
LocalConsistencyForEC ;
GetNewUpperBound ;
repeat* (first one (PostLTUpperBound ;
                    LocalConsistencyForEC ;
                    GetNewUpperBound
                    ,
                    SetLowerBoundToUpperBound)
        ) ;
FCFirstToLastOne ;
GetSolutionCSOP
end
```

- La règle `PostLTEUpperBound` ajoute une contrainte du style `1(*)x(+)0<=?ub`.
- La sous-stratégie `LocalConsistencyForEC`, définie dans la section 3.6.5, vérifie la consistance locale de tout l'ensemble de contraintes. Ce pas modifie donc les contraintes d'appartenance et éventuellement élimine des formules closes.
- La règle `GetNewUpperBound` cherche une solution de l'ensemble de contraintes courant. S'il y en a, à partir de la première solution trouvée, elle met à jour la borne supérieure, mais puisque cette solution est choisie arbitrairement, elle ne modifie pas l'ensemble de contraintes. S'il n'y a pas de solution, alors cette règle ne peut pas être appliquée et donc la stratégie `MinSatToUnsat` échoue.
- La première sous-stratégie essayée par l'opérateur **first one** d'abord poste la contrainte `1(*)x(+)0<?ub`, vérifie la consistance locale (l'ensemble de contraintes d'appartenance est donc éventuellement réduit) et cherche une nouvelle borne supérieure. Si cette sous-stratégie est appliquée, on obtient donc une nouvelle borne supérieure et un ensemble de contraintes modifié seulement par la vérification de consistance locale. Si cette sous-stratégie ne peut pas être appliquée parce que le problème est devenu insatisfaisable, la règle `SetLowerBoundToUpperBound` peut s'appliquer si les bornes inférieure et supérieure sont différentes. Dans le cas contraire, si les bornes sont égales, alors cette règle ne peut pas être appliquée et la boucle **repeat*** termine.
- Finalement, après la boucle **repeat***, on calcule la solution optimale. La règle `GetSolutionCSOP` fait uniquement des transformations pour la présentation des résultats.

Même si l'approche *satisfaisabilité à insatisfaisabilité* n'utilise pas la borne inférieure, nous avons ajouté ici la règle `SetLowerBoundToUpperBound` pour maintenir une cohérence par rapport à l'approche *splitting* et ainsi pouvoir réutiliser les mêmes règles atomiques dans la spécification des deux stratégies.

La stratégie `MinSplitting` implante l'approche que nous avons appelée *splitting*.

```
stratop
global
      MinSplitting     : < csop -> csop >     bs;
end

[] MinSplitting =>
PostLTEUpperBound ;
LocalConsistencyForEC ;
GetNewUpperBound ;
EstimateLowerBound ;
repeat* (first one (SetUpperBoundToMiddle ;
                    PostLTUpperBound ;
                    LocalConsistencyForEC ;
                    GetNewUpperBound ;
                    EstimateLowerBound
                    ,
                    SetLowerBoundToMiddle)
        ) ;
FCFirstToLastOne ;
GetSolutionCSOP
end
```

- Les calculs avant la boucle **repeat*** sont similaires a ceux réalisés par la stratégie `MinSatToUnsat`, sauf que l'on ajoute un pas de minoration de la fonction d'optimisation pour mettre à jour la borne inférieure lors du calcul d'une nouvelle solution.
- La première sous-stratégie de l'opérateur **first one** est similaire à celle de la stratégie `MinSatToUnsat`, sauf que
  - avant de poster la contrainte `1(*)x(+)0<?ub`, on affecte provisoirement la borne supérieure par la valeur correspondant à la moitié de l'intervalle de valeurs prises par la fonction d'optimisation ;
  - chaque fois qu'une nouvelle solution est trouvée, la borne inférieure est affectée par le résultat de la minoration de la fonction d'optimisation.
- Si la première sous-stratégie de l'opérateur **first one** ne peut pas être appliquée, alors on met à jour la borne inférieure sans avoir à vérifier qu'il n'y a pas de solution qui donne une valeur pour la fonction d'optimisation plus petite que la valeur correspondant à la moitié de l'intervalle. Tout ce travail a été déjà fait par la non-application de la première sous-stratégie.

Il est important de signaler deux faits concernant le comportement de ces deux stratégies.

- Le comportement dans le pire cas de la stratégie `MinSatToUnsat` se produit lorsqu'on fait face à un problème dont sa solution optimale se trouve très loin de la borne supérieure initiale. La stratégie fera beaucoup de calcul avant de la détecter car, comme nous l'avons signalé dans la section 4.4.3, en utilisant cette stratégie la proportion de valeurs éliminées par rapport au nombre total de valeurs prises par la fonction d'optimisation est très basse.
- En appliquant la stratégie `MinSplitting`, la même situation se produit quand la solution optimale se trouve près de la borne supérieure initiale.

Puisqu'il n'est pas évident de connaître *a priori* la localisation de la solution optimale, un choix entre ces deux stratégies est impossible dans le cas général.

Dans l'annexe A.2, nous montrons des résultats expérimentaux de l'application de notre système pour la résolution de problèmes d'ordonnancement d'activités et de coloriage de graphes.

La figure 4.8 présente le comportement de la meilleure stratégie que nous avons utilisée, la stratégie `MinSplittingFCMinimumDomainFirstToLast`, pour la résolution du problème de coloriage de graphes considérant 10 nœuds.

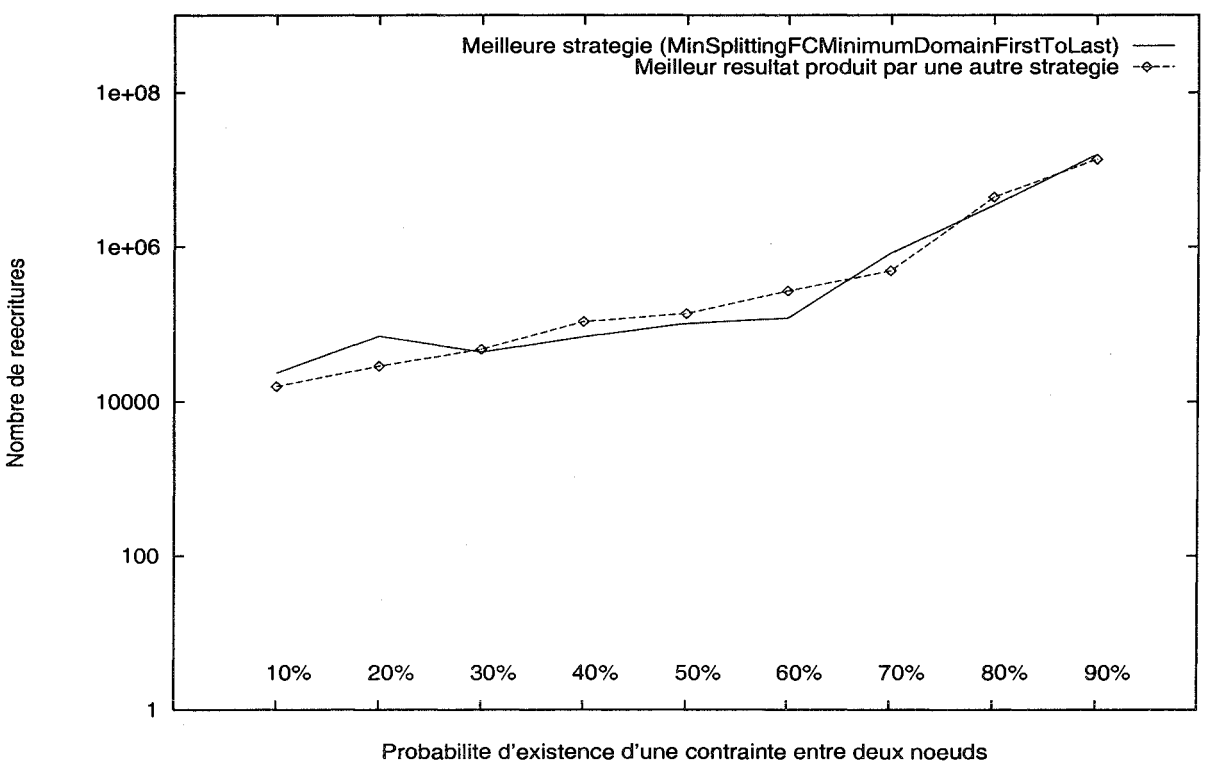

FIG. 4.8 – *Résolution du problème de coloriage de graphes : minimisation considérant 10 nœuds*

Finalement, la figure 4.9 montre le comportement de la stratégie la plus performante que nous avons utilisée pour la résolution du problème d'ordonnancement d'activités considérant 10 activités : `MinSplittingFCMaximumDegreeInECFirstToLast`.

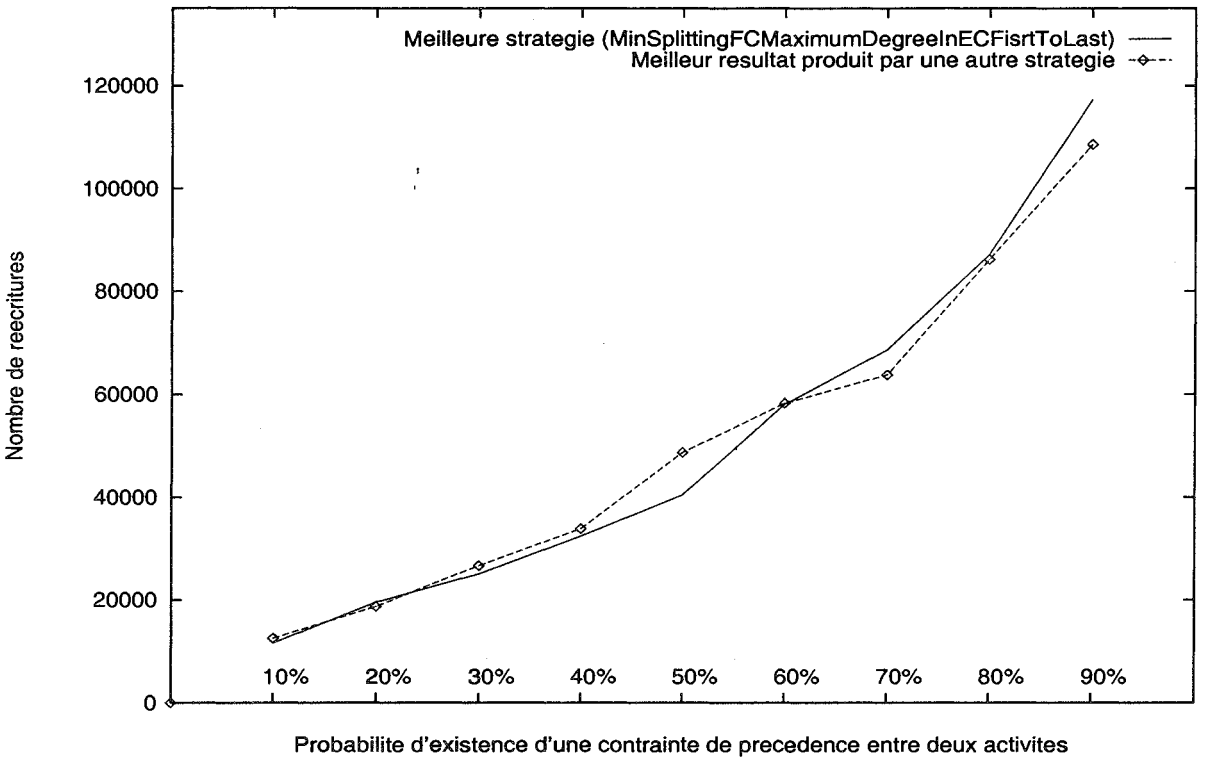

FIG. 4.9 – *Résolution du problème d'ordonnancement :* 10 *activités*

Nous pouvons observer que dans les deux problèmes considérés on ne peut pas vérifier la présence du phénomène de transition de phase car l'effort nécessaire pour résoudre un problème est d'autant plus grand que l'ensemble de contraintes est restrictif. Nous pensons que l'absence du phénomène de transition de phase s'explique par le fait que notre système effectue des calculs de pré-traitement du problème qui dépendent de la taille du problème. Vu le temps négligeable de résolution de chaque problème, ce phénomène n'est pas assez important pour se manifester.

Chapitre 5

Contraintes disjonctives

Sommaire

Nous avons présenté jusqu'ici un langage de contraintes très restreint quant à ses possibilités de combinaison des contraintes élémentaires. Nous nous intéressons maintenant à l'extension du langage afin de considérer la combinaison des contraintes élémentaires avec les connecteurs logiques de disjonction, d'implication et d'équivalence. Pour cela nous ramènerons ce problème à la résolution de problèmes en forme normale conjonctive (CNF[26]). Dans ce cadre nous trouvons un problème additionnel, lequel consiste à gérer la disjonction entre contraintes élémentaires ce qui rend le problème encore plus dur. L'utilisation de contraintes disjonctives se trouve à la base de nombreuses applications. Le traitement de la disjonction a été étudié par les communautés ILP et CLP depuis longtemps. Dans le cadre CLP, l'approche dite de la disjonction constructive semble être la plus riche d'un point de vue conceptuel dû à l'utilisation très active qu'elle fait des composantes des disjonctions. C'est à partir de ce concept de disjonction constructive que nous proposons dans ce chapitre une nouvelle approche pour le traitement des contraintes disjonctives.

Nous commençons par présenter les approches existantes pour traiter la disjonction : l'utilisation de variables binaires, proposée par la communauté ILP, la gestion de points de choix et la disjonction constructive, toutes les deux utilisées par la communauté CLP. À partir de ces

26. Conjunctive Normal Form.

techniques, nous définissons ensuite un système de calcul qui fait une utilisation passive des disjonctions et un autre implantant l'approche des points de choix. Basés sur la notion de disjonction constructive, nous proposons une nouvelle approche qui peut être considérée comme une extension basée principalement sur des considérations d'implantation de la disjonction constructive. Nous détaillons ensuite l'implantation de ces systèmes et présentons aussi quelques considérations relatives à la gestion des points de choix. Finalement, nous présentons le comportement de notre système pour la résolution d'un problème très simple de calcul de calendriers et de l'un des problèmes les plus durs considérés par la communauté CSP : le problème d'ordonnancement disjonctif.

5.1 Préliminaires

Un problème très difficile auquel on fait face lors de la résolution de contraintes est le traitement des contraintes disjonctives. Une contrainte disjonctive est définie comme étant une combinaison de contraintes élémentaires avec des opérateurs de disjonction $\vee$.

Définition 41 (CSP avec disjonctions) *Étant donnés une signature $\Sigma = (\mathcal{F}, \mathcal{P})$, un ensemble de symboles de variables $\mathcal{X}$ et une structure $\mathcal{D} = (D, I)$, un $\langle \Sigma, \mathcal{X}, \mathcal{D} \rangle$-CSP avec disjonctions est un ensemble formé par des contraintes élémentaires et par leur combinaison avec les connecteurs de conjonction $\wedge$ et de disjonction $\vee$.*

Afin de bien expliquer la nécessité de traiter la disjonction, nous présentons quelques exemples où les contraintes disjonctives sont utilisées pour modéliser un problème.

- La fonction valeur absolue inclut implicitement une disjonction. Soient x, y deux variables et I une constante. La valeur absolue

$$| x_1 - x_2 | \ \geq^? \ I$$

peut être représentée par la contrainte disjonctive ci-dessous

$$x_1 - x_2 \ \geq^? \ I \ \vee \ x_2 - x_1 \ \geq^? \ I \tag{5.1}$$

- Lors de la modélisation du problème de coloriage de graphes, on utilise fondamentalement des contraintes du style $x_i \neq x_j$ pour interdire l'assignation de la même couleur à deux nœuds adjacents. Ces contraintes peuvent aussi être exprimées par des disjonctions de la forme

$$x_i - x_j \ \geq^? \ 1 \ \vee \ x_j - x_i \ \geq^? \ 1 \tag{5.2}$$

- Le problème d'ordonnancement disjonctif représente un des problèmes les plus difficiles traités par la communauté CSP. Si deux tâches A et B, de durées respectives d_A et d_B, ne peuvent pas être réalisées en même temps, la contrainte disjonctive ci-dessous exprime cette situation

$$t_A + d_A \leq t_B \ \vee \ t_B + d_B \leq t_A$$

où t_A et t_B représentent le temps de début de la tâche A et de la tâche B, respectivement.

Du point de vue de l'expressivité d'un langage de contraintes, la disjonction joue aussi un rôle important. Comme les connecteurs logiques d'implication et d'équivalence peuvent être exprimés par des combinaisons de conjonctions et des disjonctions, il nous suffit de définir les règles de transformation adéquates pour étendre notre langage de contraintes.

Nous pouvons donc étendre la définition d'un CSP et considérer que les contraintes élémentaires peuvent aussi être combinées avec les opérateurs d'implication $\Rightarrow$ et d'équivalence $\Leftrightarrow$. En appliquant les règles de transformation suivantes à un tel CSP, on obtient un CSP avec des disjonctions.

$$\begin{aligned}
c_1 \Leftrightarrow c_2 &\rightarrow (c_1 \Rightarrow c_2) \wedge (c_2 \Rightarrow c_1) \\
c_1 \Rightarrow c_2 &\rightarrow \neg c_1 \vee c_2 \\
\neg\neg c_1 &\rightarrow c_1 \\
\neg(c_1 \vee c_2) &\rightarrow \neg c_1 \wedge \neg c_2 \\
\neg(c_1 \wedge c_2) &\rightarrow \neg c_1 \vee \neg c_2 \\
c_1 \vee (c_2 \wedge c_3) &\rightarrow (c_1 \vee c_2) \wedge (c_1 \vee c_3) \\
(c_2 \wedge c_3) \vee c_1 &\rightarrow (c_2 \vee c_1) \wedge (c_3 \vee c_1)
\end{aligned}$$

Le traitement de la négation des contraintes élémentaires peut être réalisé par des règles de transformation des symboles de prédicats de la signature (par exemple, $\neg(x =^? y)$ peut être transformée en $x \neq^? y$).

5.2 Approches existantes pour le traitement de la disjonction

En raison de son intérêt théorique et pratique, la manipulation de la disjonction est un sujet très étudié [111, 60, 122]. La différence entre les techniques existantes se base fondamentalement sur l'utilisation des composantes d'une disjonction. La première approche proposée dans le cadre CLP utilise le mécanisme de retour arrière des langages à la Prolog : les composantes d'une disjonction sont postées en suivant une philosophie *générer puis tester* [109]. Le principe *Andorra* utilise les disjonctions uniquement pour vérifier la faisabilité des affectations des variables mais non pour réduire l'espace de recherche [118]. L'opérateur de cardinalité utilise les disjonctions d'une manière plus active. Par exemple, quand une des composantes d'une disjonction est impliquée par le store de contraintes on élimine la contrainte disjonctive concernée ; quand une des composantes est prouvée être inconsistante par rapport au store de contraintes, alors on l'élimine et de cette façon on simplifie la contrainte disjonctive concernée [110]. Ces approches considèrent les disjonctions au niveau des prédicats et non au niveau des variables. L'approche basée sur l'utilisation de variables binaires, proposée par la communauté RO, entre aussi dans ce schéma [120]. L'utilisation la plus active des disjonctions est représentée actuellement par le concept de disjonction constructive [111]. Dans cette approche des informations communes à toutes les composantes d'une disjonction sont extraites sans faire des choix *a priori* entre les composantes. Dans cette section, nous présentons un résumé de ces approches. La fonction valeur absolue, présentée dans la section 5.1, est un bon exemple pour expliquer les différentes approches.

5.2.1 Variables binaires

Une technique largement utilisée par la communauté RO se base sur l'introduction de variables binaires, qui sont aussi appelées variables 0/1 [120]. L'idée consiste à associer une variable

binaire à chaque contrainte disjonctive de façon telle que chaque valeur prise par la variable active une composante de la disjonction et désactive l'autre[27]. Une contrainte est dite activée si elle est satisfaite par toutes les combinaisons de valeurs de toutes les variables apparaissant dans la contrainte.

Considérant la contrainte disjonctive (5.1), on peut introduire la variable binaire $z \in \{0,1\}$ de la manière suivante

$$\begin{aligned} (1-z) * M + x_1 - x_2 &\geq^? I \\ z * M + x_2 - x_1 &\geq^? I \end{aligned}$$

où M est un nombre suffisamment grand[28]. Nous avons ainsi transformé une contrainte disjonctive en une conjonction de deux contraintes élémentaires. Si $z = 1$, alors la deuxième contrainte élémentaire est activée (la deuxième composante de la disjonction), c'est-à-dire satisfaite, et la première contrainte élémentaire (la première composante de la disjonction) contraint les valeurs prises par les variables.

Cette formalisation donne l'effet de poster les contraintes comme un point de choix, mais dans ce cas la procédure d'énumération peut choisir la variable d'énumération au meilleur endroit dans l'espace de recherche. Dès qu'une valeur est affectée à la variable binaire pendant le processus de résolution, une composante de la disjonction sera impliquée par l'ensemble de contraintes et l'autre sera utilisée pour réduire l'ensemble de valeurs prises par les variables. Dans le pire des cas, une procédure d'énumération essayera toutes les combinaisons de contraintes élémentaires possibles, c'est-à-dire si on suppose qu'il y a K contraintes disjonctives chacune contenant deux composantes, on doit analyser 2^K ensembles de contraintes élémentaires différents.

Cette simple idée peut être étendue pour considérer en général la sélection de sous-ensembles des composantes d'une contrainte disjonctive. Étant donnée la disjonction suivante composée de m contraintes élémentaires

$$f_1(x_1,\ldots,x_n) \geq^? 0 \vee \ldots \vee f_m(x_1,\ldots,x_n) \geq^? 0$$

- la sélection d'au moins k contraintes élémentaires est exprimée par

$$\begin{aligned} f_1(x_1,\ldots,x_n) - z_1 * L_1 &\geq^? 0 \\ &\vdots \\ f_m(x_1,\ldots,x_n) - z_m * L_m &\geq^? 0 \\ \sum_{i=1}^{m} z_i &\leq^? m-k \end{aligned}$$

- la sélection d'exactement k contraintes élémentaires est exprimée par

$$\begin{aligned} f_1(x_1,\ldots,x_n) - z_1 * L_1 &\geq^? 0 \\ &\vdots \\ f_m(x_1,\ldots,x_n) - z_m * L_m &\geq^? 0 \end{aligned}$$

27. On considère ici seulement deux composantes par disjonction.
28. En fait il nous suffit $Max\{Max(I-(x_1-x_2)), Max(I-(x_2-x_1))\}$.

$$\sum_{i=1}^{m} z_i \quad =^? \quad m-k$$

- la sélection d'au plus k contraintes élémentaires est exprimée par

$$\begin{aligned} f_1(x_1,\ldots,x_n) - z_1 * L_1 &\geq^? 0 \\ &\vdots \\ f_m(x_1,\ldots,x_n) - z_m * L_m &\geq^? 0 \\ \sum_{i=1}^{m} z_i &\geq^? m-k \end{aligned}$$

où L_i est une borne inférieure pour la fonction $f_i(x_1,\ldots,x_n)$ et les z_i sont des variables binaires.

5.2.2 Points de choix

La première approche utilisée par la communauté CLP pour traiter les contraintes disjonctives se base sur des choix *a priori* entre leurs composantes pendant le processus de résolution. Une composante est choisie et postée comme une contrainte élémentaire : si l'ensemble de contraintes élémentaires ainsi obtenue est inconsistant alors une autre composante est choisie et postée [109].

L'approche est basée sur le mécanisme de retour arrière des programmes logiques. La contrainte (5.1) est gérée par les prédicats Prolog suivants

```
p(X1,X2,I) :-
   X1 - X2 >= I.
p(X1,X2,I) :-
   X2 - X1 >= I.
```

Cette approche peut être vue comme un processus d'énumération des contraintes élémentaires du problème : les composantes de chaque contrainte disjonctive sont postées en créant ainsi toutes les combinaisons possibles de contraintes élémentaires. La figure 5.1 présente l'arbre d'énumération pour un ensemble de contraintes $\{C, c_{11} \vee c_{12} \vee c_{13}, c_{21} \vee c_{22}\}$, où toutes les contraintes dans C sont élémentaires.

L'espace de recherche n'est pas réduit activement, sauf lorsqu'une contrainte est postée de façon non-déterministe ce qui peut impliquer évidemment une explosion combinatoire. Dans le pire des cas, 2^K ensembles de contraintes différents doivent être analysés, où K désigne le nombre de disjonctions et on suppose à nouveau que chacune contient deux composantes. Ainsi, pour beaucoup de problèmes une telle approche introduit trop de points de choix ce qui entraîne des performances peu satisfaisantes.

5.2.3 Disjonction constructive

Une approche plus récente, développée également au sein de la communauté CLP, est appelée disjonction constructive. Son idée fondamentale est d'extraire des informations communes à toutes les composantes d'une disjonction [111]. Nous allons utiliser l'instance suivante de la contrainte (5.1) pour expliquer cette approche

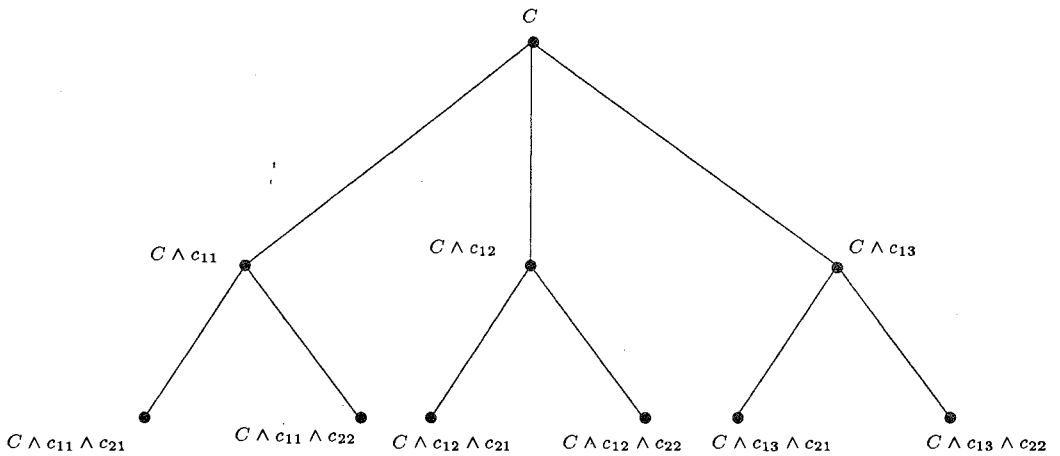

FIG. 5.1 – *Arbre d'énumération des disjonctions*

$$\begin{array}{lll} x_1 - x_2 \geq^? 8 & \vee & x_2 - x_1 \geq^? 8 \\ x_1 & \in^? & \{1,\ldots,10\} \\ x_2 & \in^? & \{1,\ldots,10\} \end{array}$$

- la première composante de la disjonction, $x_1 - x_2 \geq^? 8$, contraint d'une part $x_1 \in^? \{9,10\}$ et d'autre part $x_2 \in^? \{1,2\}$;
- la deuxième composante, $x_2 - x_1 \geq^? 8$, contraint $x_1 \in^? \{1,2\}$ et $x_2 \in^? \{9,10\}$.

Indépendamment de la composante choisie, nous savons que ni x_1 ni x_2 ne peuvent prendre les valeurs $\{3,\ldots,8\}$. L'information commune que nous pouvons déduire à partir de ces deux alternatives est $x_1 \in^? \{1,2,9,10\}$ et $x_2 \in^? \{1,2,9,10\}$, c'est-à-dire l'union des ensembles de valeurs restantes une fois que l'on poste chaque composante indépendamment.

Ceci est l'essence de la disjonction constructive, on déduit des informations communes à partir de toutes les composantes d'une disjonction afin de permettre que d'autres parties du problème profitent de cette information additionnelle. Les contraintes disjonctives sont utilisées activement sans choix *a priori* entre ses différentes alternatives.

Il est important de remarquer que pour utiliser cette approche il faut être capable de calculer l'information concernant chaque composante. Une possibilité est d'utiliser justement l'approche de points de choix : on crée plusieurs sous-problèmes en postant chaque composante d'une disjonction, ainsi la solution de chaque sous-problème nous donne les valeurs possibles pour chaque variable et donc l'union de ces valeurs correspond aux valeurs restantes pour chaque variable.

Dans le pire des cas, si on ne peut pas déduire d'information additionnelle à partir des composantes, un processus d'énumération des contraintes doit être réalisé et il analysera 2^K ensembles de contraintes élémentaires différents.

5.3 Systèmes de calcul pour la résolution de CSPs avec des disjonctions

Les langages de programmation implantant le non-déterminisme par des techniques de retour arrière permettent une manipulation naturelle des disjonctions en se basant sur l'approche

des points de choix. On verra dans cette section comment, en définissant une seule règle de transformation, nous pouvons simuler cette approche. Cependant, nous nous sommes interessés à l'étude de techniques plus performantes, en particulier, la disjonction constructive. Ainsi, nous avons proposé une nouvelle approche pour le traitement de la disjonction basée sur le concept de disjonction constructive. Brièvement, notre idée consiste à décomposer l'ensemble de contraintes disjonctives en plusieurs sous-ensembles qui ne partagent pas de variables. De cette façon nous appliquons l'approche de la disjonction constructive en considérant toutes les combinaisons des composantes de chaque sous-ensemble et on obtient ainsi plus d'informations qu'en utilisant l'approche standard. Notre principale motivation a été de concevoir une technique générale pour la manipulation des disjonctions qui soit indépendante du domaine d'application. Des techniques spécifiques et très efficaces ont été proposées, par exemple, pour le traitement du problème d'ordonnancement, mais elles sont basées sur des caractéristiques spécifiques de ce problème [17].

Dans cette section, nous définissons d'abord la forme résolue représentant le résultat que nous désirons obtenir. Basés sur l'analyse des techniques pour le traitement de la disjonction, présentées dans la section 5.2, nous proposons ensuite divers systèmes de calcul : un système de calcul qui fait une utilisation tout à fait passive des contraintes disjonctives, un système de calcul basé sur l'approche des points de choix et notre proposition basée sur la disjonction constructive.

5.3.1 Forme résolue

La définition d'une forme basique d'un CSP avec des disjonctions est une simple extension de la forme basique définie dans la section 3.5.1. Cette extension considère un ensemble de contraintes en CNF.

Définition 42 (Forme basique d'un CSP avec des disjonctions) *Une* forme basique *pour un CSP P avec des disjonctions est une conjonction de formules*

$$C = \bigwedge_{i \in I}(x_i \in^? D_{x_i}) \wedge \bigwedge_{j \in J}(x_j =^? v_j) \wedge \bigwedge_{m \in M}(c^?_m) \wedge \bigwedge_{k \in K} \bigvee_{l \in L_k}(c^?_{kl})$$

équivalente à P, où $v_j \in D$ est une valeur quelconque dans le domaine d'interprétation considéré et $c^?_m$ et $c^?_{kl}$ sont des contraintes élémentaires, telle que

- $\forall i, j \in I \cup J : x_i \neq x_j$
- $\forall i \in I : D_{xi} \neq \emptyset$
- $\forall m \in M : \mathcal{V}ar(c^?_m) \subseteq \{x_i | i \in I \cup J\}$
- $\forall k \in K \; \forall l \in L_k : \mathcal{V}ar(c^?_{kl}) \subseteq \{x_i | i \in I \cup J\}$

À partir de cette définition de forme basique, on étend aussi les définitions d'affectation basique, de forme résolue et de solution.

5.3.2 Approche passive

L'approche la plus simple que l'on peut envisager pour traiter les disjonctions consiste à les considérer uniquement quand toutes leurs variables ont été instanciées. Cela pourrait être implanté donc dans les stratégies de vérification de consistance globale présentées dans la section 3.5.3. Si on analyse ces stratégies on peut réaliser que la seule règle qu'il faut modifier c'est la règle **Elimination**. Lors de l'application de cette règle, il nous faut évaluer la formule close correspondant à chaque composante et prendre en compte les propriétés logiques suivantes

$$C \vee \mathbf{V} \rightarrow \mathbf{V}$$

$$C \vee \mathbf{F} \rightarrow C$$

De cette façon la règle **Elimination** modifiée est capable de gérer les contraintes disjonctives et on peut donc obtenir des versions étendues des stratégies ***GenerateTest***, ***Backtracking*** et ***FullLookaheadForElementaryConstraints*** pour traiter des conjonctions de contraintes disjonctives. Évidemment, cette approche fait une utilisation tout à fait passive des disjonctions.

5.3.3 Approche des points de choix

Dans cette section, nous présentons une règle de transformation et des stratégies de vérification de consistance locale et globale en se basant sur l'approche des points de choix. Cette formalisation fut réalisée d'abord pour le cas des contraintes binaires [21, 22]. Elle fut ensuite étendu pour le cas des contraintes d'arité quelconque [24].

Règle de transformation

Afin de simuler l'approche des points de choix, nous définissons la règle **CreateChoicePoint**, laquelle est présentée dans la figure 5.2. Cette règle décompose l'ensemble de composantes d'une contrainte disjonctive en deux sous-ensembles et les envoie à l'ensemble de contraintes C créant ainsi deux sous-problèmes.

[CreateChoicePoint]
$\bigvee_{l \in L_k}(c^?_{kl}) \ \wedge \ C$
$\Rightarrow$
$\bigvee_{l \in L'_k}(c^?_{kl}) \ \wedge \ C$ **or** $\bigvee_{l \in L''_k}(c^?_{kl}) \ \wedge \ C$
if $L'_k \cup L''_k = L_k$ **and** $L'_k \cap L''_k = \emptyset$

FIG. 5.2 – *Règle pour la séparation des composantes d'une contrainte disjonctive*

Le coté droit de la règle **CreateChoicePoint** est équivalent à

$$(\bigvee_{l \in L'_k}(c^?_{kl}) \quad \textbf{or} \quad \bigvee_{l \in L''_k}(c^?_{kl})) \ \wedge \ C$$

Cette expression est équivalente à

$$(\bigvee_{l \in L'_k}(c^?_{kl}) \quad \vee \quad \bigvee_{l \in L''_k}(c^?_{kl})) \ \wedge \ C$$

ce qui correspond au coté gauche de la règle, car $L'_k \cup L''_k = L_k$. La règle **CreateChoicePoint** est donc correcte et complète.

Lemme 11 *La règle* **CreateChoicePoint** *est correcte et complète.*

Preuve: Elle est directe à partir de l'examen de la règle.

□

Lemme 12 *L'application répétée de la règle* **CreateChoicePoint** *termine.*

Preuve : Comme il existe un nombre fini de composantes dans chaque disjonction et il existe un nombre fini de disjonctions, l'application répétée de cette règle terminera forcément.

□

Lemme 13 *Étant donné un CSP P avec des disjonctions en forme basique, en appliquant la règle* **CreateChoicePoint** *on obtient toujours un CSP avec des disjonctions en forme basique.*

Preuve : Comme la règle **CreateChoicePoint** sépare en deux ensembles les composantes d'une disjonction pour créer deux sous-problèmes, il est évident que les sous-problèmes ainsi obtenus sont en forme basique.

□

Théorème 4 *Étant donné un CSP P avec des disjonctions en forme basique, en utilisant la règle* **CreateChoicePoint** *jusqu'à ce qu'elle ne puisse plus être appliquée, on obtient un ensemble de CSPs en forme basique formés uniquement de conjonctions de contraintes élémentaires.*

Preuve : Elle est directe en utilisant les trois lemmes précédents.

□

Donc, si on veut obtenir chaque solution du CSP P initial il nous suffit de créer l'ensemble de CSPs consistant seulement des conjonctions de contraintes élémentaires en appliquant la règle **CreateChoicePoint** et appliquer ensuite une stratégie pour la résolution de conjonctions de contraintes élémentaires : la solution du problème initial se trouve dans l'union des solutions à chaque sous-problème.

Stratégies de vérification de consistance

La vérification de consistance locale en utilisant l'approche des points de choix implique l'énumération de toutes les combinaisons possibles des contraintes élémentaires qui peuvent être créées lors de l'envoi des composantes des contraintes disjonctives.

À partir de la stratégie de vérification de consistance locale pour un ensemble de contraintes élémentaires ***LocalConsistencyForElementaryConstraints***, définie dans la section 3.5.3, et de la règle **CreateChoicePoint**, nous définissons la stratégie ***LocalConsistencyForAllConstraints***, laquelle est présentée dans la figure 5.3

```
LocalConsistencyForAllConstraints =>
LocalConsistencyForElementaryConstraints;
repeat* (dk (CreateChoicePoint);
          LocalConsistencyForElementaryConstraints)
end
```

FIG. 5.3 – *Vérification de consistance locale pour des CSPs avec des disjonctions*

Cette stratégie commence par réaliser un pas de vérification de consistance locale pour l'ensemble initial de contraintes élémentaires. Ensuite, à chaque itération de la boucle **repeat***, elle engendre deux sous-problèmes par la création d'un point de choix et elle vérifie la consistance locale pour chaque sous-problème. L'itération de la boucle **repeat*** termine quand il n'y a plus de contraintes disjonctives à décomposer. On obtient donc comme résultat un ensemble de conjonctions de contraintes élémentaires localement consistantes.

La stratégie ***FullLookaheadForElementaryConstraints***, définie dans la section 3.5.3, résout des conjonctions de contraintes élémentaires. D'ailleurs, l'application de la stratégie de vérification de consistance locale ***LocalConsistencyForAllConstraints*** retourne un ensemble de sous-problèmes localement consistants, chacun étant une conjonction de contraintes élémentaires. À partir de ces deux stratégies nous pouvons définir la stratégie de vérification de consistance globale ***FullLookaheadForAllConstraints*** qui est présentée dans la figure 5.4.

```
FullLookaheadForAllConstraints =>
LocalConsistencyForAllConstraints;
repeat* (dk (EnumerateValue);
         first one (Elimination , id);
         LocalConsistencyForElementaryConstraints)
end
```

FIG. 5.4 – *Vérification de consistance globale pour des CSPs avec des disjonctions*

En appliquant cette stratégie, on vérifie d'abord la consistance locale et on obtient ainsi plusieurs résultats chacun étant une conjonction de contraintes élémentaires. À partir de chaque résultat obtenu, on réalise une énumération des variables afin de trouver les solutions du problème.

Cette stratégie exprime un principe fondamental de la résolution de contraintes présenté lors de la définition des stratégies de vérification de consistance globale pour des conjonctions de contraintes élémentaires dans la section 3.5.3 : nous réalisons autant de calculs déterministes que possible avant de réaliser de calculs non-déterministes. Dans ce cas, le calcul non-déterministe est représenté par les choix *a priori* qu'il faut faire entre les composantes des disjonctions et entre les variables et leurs valeurs. Évidemment, si on réalise d'abord le calcul non-déterministe concernant l'énumération des variables et de leurs valeurs, on se situe dans le cas d'une utilisation tout à fait passive des disjonctions.

5.3.4 Proposition d'une nouvelle approche basée sur la disjonction constructive

À notre avis, les implantations existantes de la disjonction constructive sont très restreint quand à l'interaction des contraintes : l'information obtenue se base uniquement sur l'analyse de chaque composante d'une disjonction par rapport au store de contraintes [60, 122].

Notre motivation principale est de développer une approche générale, indépendant du domaine d'application, pour traiter les disjonctions. Plusieurs techniques spécifiques ont été proposées, en utilisant principalement des contraintes redondantes, pour attaquer les contraintes disjonctives dans des problèmes d'ordonnancement [17]. Ces techniques sont très efficaces mais se basent sur la structure particulière du problème, c'est-à-dire elles ne peuvent pas être appliquées à tout type de problème contenant de contraintes disjonctives. La principale contribution de notre approche est qu'elle se base sur la structure mathématique du problème.

Notre idée générale est de décomposer d'abord un problème en deux sous-problèmes. Le premier contenant toutes les contraintes élémentaires et le deuxième contenant toutes les contraintes disjonctives. La consistance locale peut être vérifiée efficacement pour le premier sous-problème. Le deuxième sous-problème est attaqué en utilisant la notion de disjonction constructive afin de tirer des informations sur les valeurs impossibles et communes à toutes les combinaisons possibles des composantes des disjonctions. Comme les deux sous-problèmes partagent des variables, l'information concernant les valeurs prises par les variables est communiquée à travers les contraintes d'appartenance. Néanmoins, vu que la vérification de consistance pour un ensemble

de contraintes disjonctives est un problème difficile, nous proposons de décomposer l'ensemble de contraintes disjonctives au maximum. Pour cela, il faut détecter le nombre maximum de sous-problèmes ayant des ensembles de variables disjoints. De cette façon, on travaille avec un ensemble de problèmes plus simples. En utilisant les contraintes d'appartenance éventuellement modifiées, on vérifie à nouveau la consistance locale pour l'ensemble de contraintes élémentaires et ainsi de suite. L'algorithme termine quand il n'y a plus de modifications dans l'ensemble de contraintes d'appartenance. L'idée générale derrière cette approche peut être appliquée pour vérifier la consistance locale et elle peut aussi être intégrée dans des techniques de recherche exhaustive. Notre approche fut développé d'abord pour le cas binaire [21]. Dans [23, 24], nous l'avons étendu pour considérer des contraintes d'arité quelconque.

Dans cette section, on présente notre approche sous la forme d'un système de calcul. On présente ensuite un algorithme pour la vérification de consistance locale et on prouve que si on peut décomposer un ensemble de contraintes disjonctives en au moins deux sous-ensembles notre approche peut être plus performante que l'approche des points de choix. Finalement, afin de mieux expliquer cet algorithme, on l'applique pour résoudre un problème d'ordonnancement.

Règles de transformation

Si une contrainte disjonctive est composée seulement d'une composante, alors on peut poster cette composante comme une contrainte élémentaire et éliminer ainsi la disjonction. Cette idée est exprimée par la règle **PostConstraint** présentée dans la figure 5.5.

[PostConstraint]
$C \wedge \bigwedge_{k \in K} \bigvee_{l \in L_k} c^?_{kl}$
$\Rightarrow$
$C \wedge c^?_{k'l} \wedge \bigwedge_{k \in K; k \neq k'} \bigvee_{l \in L_k} c^?_{kl}$
if $|L_{k'}| = 1$

FIG. 5.5 – *Règle pour l'envoi d'une contrainte disjonctive*

Si une des composantes d'une disjonction est inconsistante avec les contraintes d'appartenance associées aux variables apparaissant dans la contrainte, alors cette composante peut être éliminée. Ceci est représenté par la règle **EliminateDisjunct** présentée dans la figure 5.6.

[EliminateDisjunct]
$\bigwedge_{i \in I}(x_i \in^? D_{x_i}) \wedge C \wedge \bigwedge_{k \in K} \bigvee_{l \in L_k} c^?_{kl}$
$\Rightarrow$
$\bigwedge_{i \in I}(x_i \in^? D_{x_i}) \wedge C \wedge \bigwedge_{k \in K; k \neq k'} \bigvee_{l \in L_k} c^?_{kl} \wedge \bigvee_{l \in L_{k'}; l \neq l'} c^?_{k'l}$
if $Sol_{\mathcal{D}}(\bigwedge_{i \in I}(x_i \in^? D_{x_i}) \wedge c^?_{k'l'}) = \emptyset$

FIG. 5.6 – *Règle pour l'élimination d'une composante inconsistante*

Par contre, si une des composantes est une formule close et sa valeur de vérité est vraie, alors on peut éliminer la disjonction. La règle **EliminateDisjunction**, présentée dans la figure 5.7, réalise cette élimination d'une disjonction.

[EliminateDisjunction]
$C \wedge \bigwedge_{k \in K} \bigvee_{l \in L_k} c^?_{kl}$
$\Rightarrow$
$C \wedge \bigwedge_{k \in K; k \neq k'} \bigvee_{l \in L_k} c^?_{kl}$
if $\exists\, l' \in L_{k'} : c^?_{k'l'} = \mathbf{V}$

FIG. 5.7 – *Règle pour l'élimination d'une disjonction*

La règle **SplitDisjunctions**, présentée dans la figure 5.8, divise l'ensemble de contraintes disjonctives en deux sous-ensembles dont les ensembles de variables sont disjoints. L'idée derrière cette règle, quand elle est itérée sur l'ensemble de contraintes disjonctives, est d'obtenir le maximum de sous-problèmes ayant des ensembles de variables disjoints.

[SplitDisjunctions]
$\bigwedge_{i \in I}(x_i \in^? D_{x_i}) \wedge C \wedge \bigwedge_{k \in K} \bigvee_{l \in L_k} c^?_{kl}$
$\Rightarrow$
$C \wedge (\bigwedge_{x_i \in \bigcup_{k \in K'; l \in L_k} \mathcal{V}ar(c^?_{kl})}(x_i \in^? D_{x_i}) \wedge \bigwedge_{k \in K'} \bigvee_{l \in L_k} c^?_{kl}$
or $\bigwedge_{x_i \in \bigcup_{k \in K''; l \in L_k} \mathcal{V}ar(c^?_{kl})}(x_i \in^? D_{x_i}) \wedge \bigwedge_{k \in K''} \bigvee_{l \in L_k} c^?_{kl})$
if $K' \cup K'' = K$ **and** $\bigcup_{k \in K'; l \in L_k} \mathcal{V}ar(c^?_{kl}) \cap \bigcup_{k \in K''; l \in L_k} \mathcal{V}ar(c^?_{kl}) = \emptyset$

FIG. 5.8 – *Règle pour la séparation de l'ensemble de disjonctions*

La règle **ComposeDisjunctions**, présentée dans la figure 5.9, compose en un seul ensemble deux sous problèmes. L'idée derrière cette règle est de reconstruire un CSP à partir des sous-problèmes qui sont créés lors de l'application de la règle **SplitDisjunctions**.

[ComposeDisjunctions]
$C \wedge (\bigwedge_{x_i \in \bigcup_{k \in K'; l \in L_k} \mathcal{V}ar(c^?_{kl})}(x_i \in^? D_{x_i}) \wedge \bigwedge_{k \in K'} \bigvee_{l \in L_k} c^?_{kl}$
or $\bigwedge_{x_i \in \bigcup_{k \in K''; l \in L_k} \mathcal{V}ar(c^?_{kl})}(x_i \in^? D_{x_i}) \wedge \bigwedge_{k \in K''} \bigvee_{l \in L_k} c^?_{kl})$
$\Rightarrow$
$C \wedge \bigwedge_{x_i \in \bigcup_{k \in K; l \in L_k} \mathcal{V}ar(c^?_{kl})}(x_i \in^? D_{x_i}) \wedge \bigwedge_{k \in K} \bigvee_{l \in L_k} c^?_{kl}$
if $K' \cup K'' = K$ **and** $\bigcup_{k \in K'; l \in L_k} \mathcal{V}ar(c^?_{kl}) \cap \bigcup_{k \in K''; l \in L_k} \mathcal{V}ar(c^?_{kl}) = \emptyset$

FIG. 5.9 – *Règle pour la composition de deux sous-problèmes*

Finalement, la règle **DomainReduction**, présentée dans la figure 5.10, vérifie la consistance locale en utilisant une approche constructive : les combinaisons impossibles de valeurs qui sont communes à toutes les combinaisons possibles des composantes d'un ensemble de contraintes disjonctives sont éliminées.

[DomainReduction]
$x_i \in^? D_{x_i} \wedge C \wedge \bigwedge_{k \in K'} \bigvee_{l \in L_k} c^?_{kl} \wedge \bigwedge_{k \in K''} \bigvee_{l \in L_k} c^?_{kl}$
$\Rightarrow$
$x_i \in^? D'_{x_i} \wedge C \wedge \bigwedge_{k \in K'} \bigvee_{l \in L_k} c^?_{kl} \wedge \bigwedge_{k \in K''} \bigvee_{l \in L_k} c^?_{kl}$
if $\forall k \in K' : x_i \in \bigcup_{l \in L_k} \mathcal{V}ar(c^?_{kl})$ **and** $D_{x_i} \neq D'_{x_i}$
and $\forall k \in K'' : x_i \notin \bigcup_{l \in L_k} \mathcal{V}ar(c^?_{kl})$

FIG. 5.10 – *Règle pour l'élimination de valeurs impossibles*

où le domaine réduit par l'application d'une approche constructive, D'_{x_i}, est défini comme suit

$$D'_{x_i} = D_{x_i} \setminus \bigcap \{v_i \in D_{x_i} \mid (\forall k \in k'; l \in L_k) \;\; c^?_{kl} \mid_{x_i=v_i} = \mathbf{F}\}$$

Stratégies de vérification de consistance

Afin de définir une stratégie de vérification de consistance locale pour un ensemble de contraintes élémentaires et de contraintes disjonctives, on commence par définir la stratégies ***LocalConsistencyForElementaryConstraintsWithConstructiveDisjunction***, présentée dans la figure 5.11

LocalConsistencyForElementaryConstraintsWithConstructiveDisjunction =>
LocalConsistencyForElementaryConstraints;
repeat* (**repeat*** (*SplitDisjunctions*);
repeat* (*DomainReduction*);
repeat* (*ComposeDisjunctions*);
repeat* (**first one** (*PostConstraint* , *EliminateDisjunct* , *EliminateDisjunction*));
LocalConsistencyForElementaryConstraints)
end

FIG. 5.11 – *Stratégie de vérification de consistance locale basée sur la disjonction constructive*

Cette stratégie combine les règles présentées et une stratégie de vérification de consistance locale pour un ensemble de contraintes élémentaires : la notion de disjonction constructive est utilisée pour tirer des informations à partir des disjonctions afin de réduire les ensembles de valeurs prises par les variables. Une fois que nous avons appliqué cette stratégie, il y aura toujours des disjonctions dans l'ensemble de contraintes. Pour obtenir des ensembles de contraintes élémentaires localement consistants, on définit la stratégie ***LocalConsistencyForAllConstraintsWithConstructiveDisjunction*** comme ci-dessous.

LocalConsistencyForAllConstraintsWithConstructiveDisjunction =>
LocalConsistencyForElementaryConstraintsWithConstructiveDisjunction;
repeat* (**dk** (*CreateChoicePoint*);
LocalConsistencyForElementaryConstraintsWithConstructiveDisjunction)
end

FIG. 5.12 – *Stratégie de vérification de consistance locale baseé sur la disjonction constructive*

On utilise la stratégie ***LocalConsistencyForElementaryConstraintsWithConstructiveDisjunction*** et on poste les composantes des disjonctions en utilisant la règle **CreateChoicePoint** laquelle fut introduite pour implanter l'approche des points de choix.

La stratégie ***LocalConsistencyForAllConstraintsWithConstructiveDisjunction*** peut être intégrée dans un schéma de recherche exhaustive du style *Full Lookahead* afin d'obtenir une stratégie de vérification de consistance globale. Ceci est exprimé par la stratégie ***FullLookaheadWithConstructiveDisjunction*** présentée dans la figure 5.13.

FullLookaheadWithConstructiveDisjunction =>
LocalConsistencyForAllConstraintsWithConstructiveDisjunction;
repeat* (**dk** (*EnumerateValue*);
first one (*Elimination* , **id**);
LocalConsistencyForElementaryConstraints)
end

FIG. 5.13 – *Stratégie de* Full Lookahead *utilisant la notion de disjonction constructive*

Cette stratégie essaie d'éliminer des valeurs inconsistantes avant de réaliser des choix *a priori* entre les composantes des disjonctions.

Un algorithme pour la vérification de consistance locale

Afin d'avoir une estimation de la complexité de notre approche, dans cette section nous présentons un algorithme pour la vérification de la consistance locale d'un ensemble de contraintes élémentaires et de contraintes disjonctives. Nous décrivons l'algorithme en utilisant une approche procédurale car nous ne disposons pas d'outils pour mesurer la complexité de l'application d'un ensemble de règles de réécriture.

Pour des raisons de clarté, nous exprimons l'ensemble de contraintes élémentaires et l'ensemble de contraintes disjonctives de la manière suivante.

$$EC = \bigwedge_{m \in M} (c_m^?)$$

$$DC = \bigwedge_{k \in K} \bigvee_{l \in L_k} (c_{kl}^?)$$

En appliquant un algorithme de décomposition sur l'ensemble de disjonctions DC, les contraintes peuvent être exprimées par

$$\bigwedge_{i \in L} \bigwedge_{k \in K_i} \bigvee_{l \in L_k} (c_{kl}^?)$$

tel que

$$\forall i_1, i_2 \in L : i_1 \neq i_2 \Rightarrow \mathcal{V}ar(\bigwedge_{k \in K_{i_1}} \bigvee_{l \in L_k} (c_{kl}^?)) \cap \mathcal{V}ar(\bigwedge_{k \in K_{i_2}} \bigvee_{l \in L_k} (c_{kl}^?)) = \emptyset$$

De cette manière

$$DC = \bigwedge_{i \in L} DC_i$$

où

$$DC_i = \bigwedge_{k \in K_i} \bigvee_{l \in L_k} (c_{kl}^?)$$

L'algorithme que nous présentons doit être vu comme un pas de pré-traitement : on essaie d'éliminer des valeurs impossible sans prendre de décisions *a priori*. Après l'application de cet algorithme, une approche comme celle des points de choix doit être appliquée pour énumérer les composantes des contraintes disjonctives. Évidemment, une itération entre une technique

d'énumération et notre algorithme est possible. Cependant, pour raisons de simplicité on présente ici l'idée essentielle.

```
1  debut
2  repeter
3  Obtenir EC et DC à partir de l'ensemble de contraintes C
4    Vérifier la consistance locale pour EC
5    Décomposer DC en ⋀_{i∈L} DC_i
6    pour tout DC_i faire
7      Vérifier la consistance locale en utilisant une approche constructive
8      pour tout disjonction dans DC_i faire
9        Essayer Eliminate-Disjunction
10       Essayer Eliminate-Disjunct
11       Essayer Post-Constraint
12     fin_faire
13   fin_faire
14 jusqu'a l'ensemble de contraintes d'appartenance n'est pas modifié
15 fin
```

Lemme 14 *L'algorithme termine et est correct.*

Preuve: La terminaison des algorithmes de vérification de consistance d'arc est bien connue, il nous faut donc seulement prouver la terminaison de la boucle **Repeter**. Dans le pire cas, après la vérification de consistance locale pour chaque DC_i, les contraintes d'appartenance seront différentes car un élément seul a été éliminé. Comme il y a au plus e variables dans l'ensemble de contraintes disjonctives et chaque variable peut prendre a valeurs, le nombre maximal d'itérations est (ae), c'est-à-dire au plus après (ae) itérations l'algorithme termine. La correction est évidente car on élimine uniquement des valeurs lorsqu'elles sont localement inconsistantes et donc on n'élimine aucune solution.

□

Théorème 5 *Si chaque disjonction contient seulement deux composantes et si nous pouvons décomposer l'ensemble de contraintes disjonctives en L sous-ensembles ayant des ensembles de variables disjoints, la complexité dans le pire cas de l'algorithme est bornée par $L2^{K-L+1}$, où K désigne le nombre total de contraintes disjonctives.*

Preuve: Comme K_i (pour $i = 1, \ldots, L$), désigne le nombre total de contraintes disjonctives dans le sous-ensemble i, pour vérifier la consistance locale pour chaque sous-ensemble i, en utilisant une approche constructive, nous devons analyser, dans le pire cas, 2^{K_i} ensembles de contraintes elementaires (on suppose que l'on utilise une approche des points de choix pour implanter la disjonction constructive). Donc, globalement dans chaque itération on doit vérifier la consistance locale pour $\sum_{i=1}^{L} 2^{K_i}$ ensembles de contraintes élémentaires. Comme $K = \sum_{i=1}^{L} K_i$, on a $\sum_{i=1}^{L} 2^{K_i} \leq L2^{K-L+1}$. Dans le pire cas $L = 1$, donc la complexité en terme de temps dans le pire cas est $O(2^K)$. On obtient ainsi le même résultat que la complexité de l'approche des points de choix.

□

Évidemment, la performance de notre algorithme dépend des caractéristiques spécifiques du problème. D'abord, plus on peut décomposer l'ensemble de contraintes disjonctives, plus efficace sera l'utilisation de la disjonction constructive car on doit traiter des sous-problèmes plus simples (ceci est le sens de l'expression $L2^{K-L+1}$). D'autre part, la performance de notre algorithme dépend de l'information que l'on peut tirer à partir des disjonctions. Plus restrictives sont les contraintes, plus d'informations on peut en tirer, i.e., plus de valeurs impossibles peuvent

être éliminées. Tenant compte de ces considérations, notre approche doit être vue comme un pas de pré-traitement, après lequel une autre technique doit être utilisée, comme par exemple, l'approche des points de choix. Finalement, il est important de signaler que notre approche peut être implantée facilement en utilisant plusieurs solveurs qui s'executent en parallèle chacun vérifiant la consistance locale d'un sous-ensemble de contraintes disjonctives.

Exemple

Nous appliquons notre algorithme à la résolution du problème d'ordonnancement 2×3, présenté dans le tableau 5.1

Job	Machine			Temps maximal
	1	2	3	
1	94	66	10	220
2	53	26	15	220

TAB. 5.1 – *Un problème d'ordonnancement* 2×3

$$Tâche_{13} + 10 \leq^? 220 \tag{5.3}$$
$$Tâche_{23} + 15 \leq^? 220 \tag{5.4}$$
$$Tâche_{11} + 94 \leq^? Tâche_{12} \tag{5.5}$$
$$Tâche_{12} + 66 \leq^? Tâche_{13} \tag{5.6}$$
$$Tâche_{21} + 53 \leq^? Tâche_{22} \tag{5.7}$$
$$Tâche_{22} + 26 \leq^? Tâche_{23} \tag{5.8}$$
$$Tâche_{11} \geq^? Tâche_{21} + 53 \quad \vee \quad Tâche_{21} \geq^? Tâche_{11} + 94 \tag{5.9}$$
$$Tâche_{12} \geq^? Tâche_{22} + 26 \quad \vee \quad Tâche_{22} \geq^? Tâche_{12} + 66 \tag{5.10}$$
$$Tâche_{13} \geq^? Tâche_{23} + 15 \quad \vee \quad Tâche_{23} \geq^? Tâche_{13} + 10 \tag{5.11}$$

Première itération

- On vérifie la consistance locale pour l'ensemble de contraintes élémentaires (5.3 - 5.8) et on obtient les contraintes d'appartenance suivantes.

$$\begin{aligned} Tâche_{11} &\in^? [0, .., 50] \\ Tâche_{12} &\in^? [94, .., 144] \\ Tâche_{13} &\in^? [160, .., 210] \\ Tâche_{21} &\in^? [0, .., 126] \\ Tâche_{22} &\in^? [53, .., 179] \\ Tâche_{23} &\in^? [79, .., 205] \end{aligned} \tag{5.12}$$

- En appliquant un algorithme de décomposition sur l'ensemble de contraintes disjonctives (5.9 - 5.11), on obtient trois sous-ensembles.

- On vérifie donc la consistance locale pour chaque sous-problème en utilisant une approche constructive.
  - En vérifiant la consistance locale pour la première contrainte disjonctive (contrainte 5.9) et les contraintes d'appartenance associées, présentées ci-dessous

$$\begin{array}{c} Tâche_{11} \geq^? Tâche_{21} + 53 \quad \vee \quad Tâche_{21} \geq^? Tâche_{11} + 94 \\ Tâche_{11} \in^? [0,..,50] \\ Tâche_{21} \in^? [0,..,126] \end{array}$$

on obtient

Solution	Contraintes d'appartenance
1	Insatisfaisable
2	$Tâche_{11} \in^? [0,..,32]$, $Tâche_{21} \in^? [94,..,126]$

TAB. 5.2 – *Première vérification de consistance locale pour le premier sous-problème*

Les contraintes d'appartenance sont

$$\begin{array}{l} Tâche_{11} \in^? [0,..,32] \\ Tâche_{21} \in^? [94,..,126] \end{array}$$

Comme information additionnelle nous savons que seule une composante est possible. Nous pouvons donc ajouter cette composante à l'ensemble de contraintes élémentaires et éliminer la contrainte disjonctive.

  - En vérifiant la consistance locale pour la deuxième contrainte disjonctive (contrainte 5.10) et les contraintes d'appartenance associées, présentées ci-dessous

$$\begin{array}{c} Tâche_{12} \geq^? Tâche_{22} + 26 \quad \vee \quad Tâche_{22} \geq^? Tâche_{12} + 66 \\ Tâche_{12} \in^? [94,..,144] \\ Tâche_{22} \in^? [53,..,179] \end{array}$$

on obtient

Solution	Contraintes d'appartenance
1	$Tâche_{12} \in^? [94,..,144]$, $Tâche_{22} \in^? [53,..,118]$
2	$Tâche_{12} \in^? [94,..,113]$, $Tâche_{22} \in^? [160,..,179]$

TAB. 5.3 – *Première vérification de consistance locale pour le deuxième sous-problème*

Dans ce cas, les contraintes d'appartenance obtenues sont

$$\begin{array}{l} Tâche_{12} \in^? [94,..,144] \\ Tâche_{22} \in^? [53,..,179] \end{array}$$

i.e., elles n'ont pas été modifiées.

- En vérifiant la consistance locale pour la troisième contrainte disjonctive (contrainte 5.11) et les contraintes d'appartenance associées, présentées ci-dessous

$$\begin{array}{rcl} Tâche_{13} \geq^? Tâche_{23} + 15 & \vee & Tâche_{23} \geq^? Tâche_{13} + 10 \\ Tâche_{13} & \in^? & [160, .., 210] \\ Tâche_{23} & \in^? & [79, .., 205] \end{array}$$

on obtient

Solution	Contraintes d'appartenance
1	$Tâche_{13} \in^? [160, .., 210]$, $Tâche_{23} \in^? [79, .., 195]$
2	$Tâche_{13} \in^? [160, .., 195]$, $Tâche_{23} \in^? [170, .., 205]$

TAB. 5.4 – *Première vérification de consistance locale pour le troisième sous-problème*

Les contraintes d'appartenance résultant sont

$$\begin{array}{rcl} Tâche_{13} & \in^? & [160, .., 210] \\ Tâche_{23} & \in^? & [79, .., 205] \end{array}$$

i.e., elles n'ont pas été modifiées.

Vu que les contraintes d'appartenance ont été modifiées on réalise une deuxième itération de l'algorithme.

Deuxième itération

- Nous avons ajouté la deuxième composante de la première disjonction (contrainte 5.9), la seule composante possible dans cette disjonction. On vérifie donc à nouveau la consistance locale pour l'ensemble de contraintes élémentaires (5.3 - 5.8, 5.9) et on obtient les contraintes d'appartenance suivantes.

$$\begin{array}{rcl} Tâche_{11} & \in^? & [0, .., 32] \\ Tâche_{12} & \in^? & [94, .., 144] \\ Tâche_{13} & \in^? & [160, .., 210] \\ Tâche_{21} & \in^? & [94, .., 126] \\ Tâche_{22} & \in^? & [147, .., 179] \\ Tâche_{23} & \in^? & [173, .., 205] \end{array} \quad (5.13)$$

- En appliquant un algorithme de décomposition sur l'ensemble de contraintes disjonctives, on obtient deux sous-ensembles.
- On vérifie donc la consistance locale pour chaque sous-problème en utilisant une approche constructive.
  - En vérifiant la consistance locale pour la deuxième contrainte disjonctive (contrainte 5.10) et les contraintes d'appartenance associées, on obtient

Solution	Contraintes d'appartenance
1	Insatisfaisable
2	$Tâche_{12} \in^? [94,..,113]$, $Tâche_{22} \in^? [160,..,179]$

TAB. 5.5 – *Deuxième vérification de consistance locale pour le deuxième sous-problème*

Les contraintes d'appartenance sont

$$Tâche_{12} \in^? [94,..,113] \tag{5.14}$$
$$Tâche_{22} \in^? [160,..,179] \tag{5.15}$$

Comme information additionnelle nous savons que seulement une composante est possible. Nous pouvons donc ajouter cette composante à l'ensemble de contraintes élémentaires et éliminer la contrainte disjonctive.

- En vérifiant la consistance locale pour la troisième contrainte disjonctive (contrainte 5.11) et les contraintes d'appartenance associées, on obtient

Solution	Contraintes d'appartenance
1	$Tâche_{13} \in^? [188,..,210]$ & $Tâche_{23} \in^? [173,..,195]$
2	$Tâche_{13} \in^? [160,..,195]$ & $Tâche_{23} \in^? [173,..,205]$

TAB. 5.6 – *Deuxième vérification de consistance locale pour le troisième sous-problème*

Les contraintes d'appartenance obtenues sont

$$Tâche_{13} \in^? [160,..,210] \tag{5.16}$$
$$Tâche_{23} \in^? [173,..,205] \tag{5.17}$$

i.e., elles n'ont pas été modifiées.

Puisque les contraintes d'appartenance ont été modifiées, on réalise une troisième itération de l'algorithme.

Troisième itération Nous avons ajouté la deuxième composante de la deuxième disjonction (contrainte 5.10), la seule composante possible dans cette disjonction.

- On vérifie donc à nouveau la consistance locale pour l'ensemble de contraintes élémentaires (5.3 - 5.8, 5.9, 5.10) et on obtient les contraintes d'appartenance suivantes.

$$\begin{aligned} Tâche_{11} &\in^? [0,..,19] \\ Tâche_{12} &\in^? [94,..,113] \\ Tâche_{13} &\in^? [160,..,210] \\ Tâche_{21} &\in^? [94,..,126] \\ Tâche_{22} &\in^? [160,..,179] \\ Tâche_{23} &\in^? [186,..,205] \end{aligned} \tag{5.18}$$

- L'ensemble de contraintes disjonctives a été réduit à une seule disjonction, on vérifie donc la consistance locale pour la troisième contrainte disjonctive (contrainte 5.11) et on obtient

Solution	Contraintes d'appartenance
1	$Tâche_{13} \in^? [201,..,210]$ & $Tâche_{23} \in^? [186,..,195]$
2	$Tâche_{13} \in^? [160,..,195]$ & $Tâche_{23} \in^? [186,..,205]$

TAB. 5.7 – *Troisième vérification de consistance locale pour le troisième sous-problème*

Les contraintes d'appartenance résultant sont

$$\begin{aligned} Tâche_{13} &\in^? [160,..,210] \\ Tâche_{23} &\in^? [186,..,205] \end{aligned}$$

i.e., elles n'ont pas été modifiées.

Comme nous avons extrait toute l'information à partir des disjonctions, nous pouvons continuer en utilisant par exemple l'approche des points de choix, c'est-à-dire qu'on résout tout l'ensemble modifié de contraintes présenté ci-dessous.

$$\begin{aligned} Tâche_{13} + 10 &\leq^? 220 \\ Tâche_{23} + 15 &\leq^? 220 \\ Tâche_{11} + 94 &\leq^? Tâche_{12} \\ Tâche_{12} + 66 &\leq^? Tâche_{13} \\ Tâche_{21} + 53 &\leq^? Tâche_{22} \\ Tâche_{22} + 26 &\leq^? Tâche_{23} \\ Tâche_{21} &\geq^? Tâche_{11} + 94 \\ Tâche_{22} &\geq^? Tâche_{12} + 66 \\ Tâche_{13} \geq^? Tâche_{23} + 15 \quad &\vee \quad Tâche_{23} \geq^? Tâche_{13} + 10 \end{aligned}$$

Ainsi, on obtient

- Première branche : Insatisfaisable.
- Deuxième branche :

$$\begin{aligned} Tâche_{11} &\in^? [0,..,19] \\ Tâche_{12} &\in^? [94,..,113] \\ Tâche_{13} &\in^? [160,..,195] \\ Tâche_{21} &\in^? [94,..,126] \\ Tâche_{22} &\in^? [160,..,179] \\ Tâche_{23} &\in^? [186,..,205] \end{aligned} \tag{5.19}$$

Celui-ci est le seul ordre possible entre les tâches de ce problème d'ordonnancement. Pour obtenir une solution quelconque ou pour obtenir la solution optimale on pourrait appliquer une technique d'énumération exhaustive.

Nous pouvons ainsi constater que notre approche réduit l'espace de recherche et élimine aussi quelques disjonctions. Il est intéressant de noter que l'idée générale de décomposer l'ensemble de contraintes disjonctives correspond, dans le cas du problème d'ordonnancement, à manipuler comme un tout les contraintes liées à une machine particulière mais de façon indépendante les contraintes liées à des machines différentes.

Finalement, deux avantages de notre approche par rapport à l'approche des points de choix peuvent être mieux compris avec l'exemple suivant.

On considère la résolution du même problème d'ordonnancement mais en utilisant l'approche des points de choix. On le résout en considérant quatre ordres différents pour l'envoi des contraintes. La première colonne dans le tableau 5.8 présente chaque ordre considéré ; la deuxième colonne présente le nombre de feuilles de l'espace de recherche qui sont visitées pour obtenir une solution.

Ordre	Échecs
1-6,7,8,9	5
1-6,8,9,7	4
1-6,9,7,8	4
1-6,8,7,9	5

TAB. 5.8 – *Application de l'approche des points de choix considérant différents ordres des contraintes*

Si l'on considère, par exemple, le premier ordre d'envoi des contraintes, son explication est la suivante : on vérifie la consistance locale pour l'ensemble de contraintes (1-6), puis on poste les composantes de la contrainte (7) créant ainsi deux sous-problèmes et à partir de ces deux sous-problèmes on continue à poster les composantes des contraintes (8) et (9).

La première conclusion est qu'en utilisant notre approche on visite un nombre plus petit de feuilles. La deuxième conclusion est qu'il est bien connu que l'ordre d'envoi des contraintes a une influence sur le nombre de feuilles visitées par l'approche des point de choix. Par contre, notre approche ne dépend pas de cet ordre.

5.4 Implantation

Dans cette section, nous présentons l'implantation de l'approche des points de choix. Nous montrons comment l'utilisation de symboles associatifs-commutatifs nous permet de simplifier l'ensemble de règles de réécriture et de mieux gérer la pose de points de choix. Nous présentons ensuite l'implantation de notre proposition basée sur la disjonction constructive. Finalement, nous montrons le comportement de notre système pour la résolution d'un problème très simple de calcul de calendriers et de l'un des problèmes les plus durs considérés par la communauté CSP : le problème d'ordonnancement disjonctif.

5.4.1 Approche des points de choix

Pour simuler le non-déterminisme du coté droit de la règle **CreateChoicePoint**, présentée dans la section 5.3.3, nous la décomposons en deux règles : **PostFirstDisjunct** et **PostSecondDisjunct**.

La règle **PostFirstDisjunct**, présentée dans la figure 5.14, correspond à la première composante du côté droit de la règle **CreateChoicePoint**.

$$\begin{array}{l}\textbf{[PostFirstDisjunct]}\\ \bigvee_{l\in L_k}(c^?_{kl}) \ \wedge \ C\\ \Rightarrow\\ \bigvee_{l\in L'_k}(c^?_{kl}) \ \wedge \ C\end{array}$$

FIG. 5.14 – *Règle pour l'envoi d'un premier ensemble des composantes d'une disjonction*

La règle **PostSecondDisjunct**, présentée dans la figure 5.15, correspond à la deuxième composante du côté droit de la règle **CreateChoicePoint**.

$$\begin{array}{l}\textbf{[PostSecondDisjunct]}\\ \bigvee_{l\in L_k}(c^?_{kl}) \ \wedge \ C\\ \Rightarrow\\ \bigvee_{l\in L''_k}(c^?_{kl}) \ \wedge \ C\end{array}$$

FIG. 5.15 – *Règle pour l'envoi d'un deuxième ensemble des composantes d'une disjonction*

Ces deux règles peuvent être gérées par l'opérateur de stratégie **dk** afin de simuler la création d'un point de choix. La spécification de la règle **PostFirstDisjunct** en ELAN est présentée ci-dessous.

```
rules for csp
        c1, c2                  : constraint;
        lmc, lec, EC, DC, store : list[constraint];
        newEC, newDC            : list[constraint];
global

[PostFirstDisjunct]
        CSP[lmc,lec,EC,c1 V c2.DC,store]
        =>
        CSP[lmc,lec,newEC,newDC,store]
        choose
        try     if      ConstraintIsElementary(c1)
                where   newEC := ()c1.EC
                where   newDC := ()DC
        try     where   newEC := ()EC
                where   newDC := ()c1.DC
        end
end
```

L'opérateur `ConstraintIsElementary` vérifie si la contrainte considérée est élémentaire. Ainsi, si la première composante de la disjonction, la contrainte `c1`, est une contrainte élémentaire, alors on l'ajoute à la liste de contraintes élémentaires. En cas contraire, on l'ajoute à la liste de contraintes disjonctives.

On définit également la règle `PostSecondDisjunct` qui fait le même traitement des composantes de la disjonction mais sur la contrainte `c2` au lieu de `c1`.

La vérification de consistance locale pour un CSP avec des disjonctions implique l'énumération de toutes les combinaisons possibles des contraintes élémentaires qui peuvent être créées lors de l'envoi des composantes des contraintes disjonctives. La stratégie ***LocalCon-***

sistencyForAllConstraints, présentée dans la section 5.3.3, est implantée par la stratégie `LocalConsistencyForECandDC` qui, en appliquant la stratégie `LocalConsistencyForEC` définie dans la section 3.6.5, vérifie la consistance locale du nouvel ensemble de contraintes élémentaires à chaque fois qu'une composante est postée.

```
stratop
global
        LocalConsistencyForECandDC : < csp -> csp >   bs;
end

strategies for csp
implicit

[] LocalConsistencyForECandDC =>
LocalConsistencyForEC ;
repeat* (dk (PostFirstDisjunct , PostSecondDisjunct) ;
         LocalConsistencyForEC)
end
```

Nous avons envisagé cette solution en considérant la manipulation efficace du non-déterminisme qui est faite dans le langage ELAN [91, 92].

En utilisant la stratégie précédente on obtient toutes les combinaisons possibles des composantes des disjonctions. Pour obtenir une seule combinaison localement consistante, il suffit de l'appliquer en utilisant un opérateur comme **first one**.

La stratégie de vérification de consistance globale ***FullLookaheadForAllConstraints***, définie dans la section 5.3.3, est implantée facilement par la stratégie `FLAChoicePointFirstToLastAll` qui retourne toutes les solutions d'un CSP.

```
stratop
global
        FLAChoicePointFirstToLastAll : < csp -> csp >   bs;
end

strategies for csp
implicit

[] FLAChoicePointFirstToLastAll =>
LocalConsistencyForECandDC ;
repeat* (dk (InstantiateFirstValueOfDomain ;
             first one (ExtractConstraintsOnEqualityVar , id) ;
             first one (Elimination , id) ;
             LocalConsistencyForEC ,
             EliminateFirstValueOfDomain)) ;
        ) ;
first one (GetSolutionCSP)
end
```

La stratégie `FLAChoicePointFirstToLastOne` ne retourne que la première solution.

```
stratop
global
        FLAChoicePointFirstToLastOne : < csp -> csp >    bs;
end

strategies for csp
implicit

[] FLAChoicePointFirstToLastOne =>
first one (FLAChoicePointFirstToLastAll)
end
```

Il est intéressant de noter que la seule différence entre ces stratégies et celles définies pour la résolution de contraintes élémentaires est l'utilisation d'une sous-stratégie qui énumère les composantes de toutes les disjonctions. On voit clairement la simplicité avec laquelle on peut étendre des solveurs en utilisant cette approche basée sur des règles et des stratégies.

5.4.2 Rôle d'un opérateur AC dans la gestion des points de choix

Si on applique la stratégie `LocalConsistencyForECandDC` sur la contrainte $C \wedge (c_1 \vee c_2 \vee c_3)$, où c_1, c_2 et c_3 sont des contraintes élémentaires, on crée d'abord un point de choix qui contient dans une branche la contrainte $C \wedge c_1$ est dans l'autre branche la contrainte $C \wedge (c_2 \vee c_3)$. Ensuite, on crée un autre point de choix à partir de la deuxième branche pour obtenir deux nouvelles branches : $C \wedge c_2$ et $C \wedge c_3$. Nous avons créé donc deux points de choix pour obtenir des CSPs ne contenant que des conjonctions de contraintes élémentaires. Ce comportement peut être amélioré si on considère un symbole de disjonction $\vee$ avec des propriétés d'associativité et de commutativité.

La définition en ELAN d'un tel symbole est la suivante.

```
operators
global
@ V @ : (constraint constraint) constraint (AC);
end
```

Ainsi, on peut remplacer les deux règles `PostFirstDisjunct` et `PostSecondDisjunct` par la règle `PostElementaryDisjunct` ci-dessous.

```
rules for csp
        c1, c2                  : constraint;
        lmc, lec, EC, DC, store : list[constraint];
global

[PostElementaryDisjunct]
        CSP[lmc,lec,EC,c1 V c2.DC,store]
        =>
        CSP[lmc,lec,c1.EC,DC,store]
        if ConstraintIsElementary(c1)
end
```

Quand on applique cette règle le langage s'occupe de trouver une contrainte élémentaire parmi les composantes de la disjonction.

On peut maintenant redéfinir la stratégie `LocalConsistencyForECandDC` de la manière suivante.

```
stratop
global
        LocalConsistencyForECandDC : < csp -> csp >   bs;
end

strategies for csp
implicit

[] LocalConsistencyForECandDC =>
LocalConsistencyForEC;
repeat* (dk (PostElementaryDisjunct);
         LocalConsistencyForEC)
end
```

Ainsi, si on applique cette nouvelle version de la stratégie `LocalConsistencyForECandDC` au même problème $C \wedge (c_1 \vee c_2 \vee c_3)$, on crée un seul point de choix qui contient trois branches : $C \wedge c_1$, $C \wedge c_2$ et $C \wedge c_3$.

Il est donc tout à fait intéressant d'utiliser des opérateurs ayant ce type de propriétés. D'une part, la spécification des règles est plus facile et plus réduite et d'autre part la gestion des points de choix est plus efficace car on a un nombre plus petit de points de choix à gérer.

5.4.3 Disjonction constructive

De la même manière que nous l'avons fait pour gérer les sous-problèmes dans le cas des CSPs décomposables, on définit une structure de données `cspCD` pour gérer les sous-ensembles de contraintes disjonctives ayant des ensembles de variables disjoints.

```
operators
global
        CSPCD@ : (set:tuple[csp, list[csp], list[csp]])      cspCD;
end
```

La première composante contient le CSP initial ; la deuxième composante contient une liste de CSPs, chacun étant un ensemble de contraintes disjonctives à résoudre ; la troisième composante contient une liste de CSPs, chacun étant un ensemble de contraintes disjonctives déjà traitées.

Afin de décomposer le CSP initial on définit la règle `DecomposeCSPCD` comme ci-dessous.

```
rules for cspCD
        lmc, lec, EC, DC, store        : list[constraint];
        lmc1                           : list[constraint];
        luCsp, luCsp1, newluCsp, lsCsp : list[csp];
global

[DecomposeCSPCD]
        CSPCD[CSP[lmc,lec,EC,DC,store],luCsp,lsCsp]
        =>
        CSPCD[CSP[nil,nil,append(EC,store),DC,nil],newluCsp,lsCsp]
        where   lmc1    := ()TransformLECintoLMC(lec)
        where   luCsp1  := ()CreateListOfCSP(append(lmc,lmc1))
        where   newluCsp:= ()DecomposeCSP(luCsp1,DC)
end
```

L'explication de cette règle est la suivante.

- Toutes les contraintes d'égalité du problème, $x = v$, sont transformées en contraintes d'appartenance $x \in^? [v, \ldots, v]$ en utilisant l'opérateur `TransformLECintoLMC`. Ce résultat est stocké dans la liste `lmc1`.
- À partir de l'union des listes `lmc` et `lmc1`, qui contient toutes les variables du problème, on crée une liste de CSPs, chacun contenant une seule contrainte d'appartenance. Pour cela on utilise l'opérateur `CreateListOfCSP`, présenté dans la section 3.6.6, et utilisé pour le traitement des CSPs décomposables. La liste `luCsp1` contient donc le nombre maximum de sous-problèmes qu'il est possible de créer.
- En appliquant l'opérateur `DecomposeCSP`, qui a aussi été présenté et utilisé pour le traitement des CSPs décomposables, sur l'ensemble de contraintes disjonctives, on obtient une liste de CSPs, chacun contenant les contraintes disjonctives liées entre elles par le partage de variables communes.
- Comme résultat final de l'application de cette règle, on obtient dans la première composante un CSP contenant seulement les ensembles de contraintes élémentaires et des contraintes disjonctives. Nous avons éliminé les contraintes d'appartenance de la première composante car elles sont maintenant inclues dans la liste de sous-ensembles de contraintes disjonctives à résoudre.

Pour traiter chaque sous-ensemble de contraintes disjonctives nous définisons ci-dessous la règle `SolveSubCSPCD`.

```
rules for cspCD
        P, Q, R             : csp;
        lmc, alllmc         : list[constraint];
        lcsp, luCsp, lsCsp  : list[csp];
global

[SolveSubCSPCD]
        CSPCD[P,Q.luCsp,lsCsp]
        =>
        CSPCD[P,luCsp,R.lsCsp]
        where  lcsp   := ()set_of(call "LocalConsistencyForDC":csp,Q)
        where  alllmc := ()GetAllLMC(nil,lcsp)
        where  lmc    := ()UnionLMC(nil,alllmc)
        where  R      := ()CSP[lmc,nil,nil,nil,nil]
end
```

L'explication de cette règle est la suivante.

- On applique la stratégie `LocalConsistencyForDC` au premier sous-ensemble de contraintes disjonctives. La spécification de cette stratégie est comme suit :

```
stratop
global
        LocalConsistencyForDC : < csp -> csp >   bs;
end

strategies for csp
implicit

[] LocalConsistencyForDC =>
repeat* (dk (PostElementaryDisjunct);
         LocalConsistencyForEC)
end
```

Cette stratégie utilise l'approche des points de choix pour vérifier la consistance locale d'un ensemble de contraintes disjonctives. Comme résultat, on obtient un ensemble de conjonctions de contraintes élémentaires localement consistantes. Chaque résultat correspond à une des combinaisons possibles des composantes des disjonctions du sous-problème. L'opérateur `set_of`, prédéfini dans ELAN, retourne chacun de ces résultats comme un élément d'une liste. Cette liste est stockée dans la variable `lcsp`.

- L'opérateur `GetAllLMC` retourne une liste dont chaque élément correspond à l'ensemble de contraintes d'appartenance de chaque élément de la liste `lcsp`.
- L'opérateur `UnionLMC` réalise l'union des tous les ensembles de valeurs possibles pour chaque variable et stocke cette liste dans la variable `lmc`.
- Finalement, la liste `lmc` est retournée comme résultat pour le sous-ensemble considéré.

Finalement, la règle `ComposeCSPCD` construit la solution du problème initial, à partir des solutions des sous-problèmes.

```
rules for cspCD
        P, Q, R      : csp;
        luCsp, lsCsp : list[csp];
global

[ComposeCSPCD]
        CSPCD[P,luCsp,Q.lsCsp]
        =>
        CSPCD[R,luCsp,lsCsp]
        where   R := ()ComposeCSP(P,Q)
end
```

Cette règle implante la même idée que la règle `ComposeCSPDis` définie et utilisée dans la section 3.6.6 pour le traitement des CSPs décomposables.

Maintenant que nous avons spécifié toutes les règles nécessaires, nous pouvons implanter une stratégie pour appliquer notre approche disjonctive. La stratégie `ConstructiveDisjunction`, présentée ci-dessous, réalise seulement l'élimination de valeurs à partir de l'information des disjonctions.

```
stratop
global
        ConstructiveDisjunction : < cspCD -> cspCD >   bs;
end

strategies for cspCD
implicit

[] ConstructiveDisjunction =>
first one (DecomposeCSPCD);
repeat* (first one (SolveSubCSPCD));
repeat* (first one (ComposeCSPCD))
end
```

Pour appliquer cette stratégie à un CSP et en raison des différentes structures de données utilisées, on définit la règle suivante.

```
rules for csp
        P, Q                    : csp;
        D1, D2                  : cspCD;
        lmc1, lmc2, lmc3, lmc4 : list[constraint];
global

[ConstructiveDisjunction]
        P
        =>
        Q
        where     D1 := ()CreateCSPCD(P)
        where     D2 := (ConstructiveDisjunction)D1
        where      Q := ()GetCSP(D2)
        where   lmc1 := ()GetLMC(P)
        where   lmc2 := ()TransformLECintoLMC(GetLEC(P))
        where   lmc3 := ()SortList(append(lmc1,lmc2),AllMembershipConstraints)
        where   lmc4 := ()SortList(GetLMC(Q),AllMembershipConstraints)
        if      not lmc3 == lmc4
end
```

L'explication de cette règle est la suivante.

- L'opérateur `CreateCSPCD` crée la structure de données `cspCD` à partir du problème initial.
- Sur le problème ainsi créé, on applique la stratégie `ConstructiveDisjunction`.
- À partir du résultat de l'application de la stratégie `ConstructiveDisjunction`, on récupère le CSP résultant et on stocke dans `lmc4` toutes ses contraintes d'appartenance.
- À partir du CSP initial, on crée une liste contenant toutes ses contraintes d'appartenance et ses contraintes d'égalité exprimées sous la forme de contraintes d'appartenance.
- Finalement, on compare les deux listes de contraintes d'appartenance afin de vérifier si l'application de la stratégie `ConstructiveDisjunction` a modifié les contraintes d'appartenance du problème initial.

On définit maintenant deux stratégies qui vérifient la consistance locale pour un ensemble de contraintes élémentaires en utilisant à différents niveaux la notion de disjonction constructive. De même que pour la définition de la stratégie ***LocalConsistencyForElementaryConstraints-WithConstructiveDisjunction***, présentée dans la section 5.3.4, ces deux stratégies vérifient

uniquement la consistance locale de l'ensemble de contraintes élémentaires et utilisent les disjonctions dans le seul but d'éliminer des valeurs prises par les variables. Après l'application de ces stratégies, il y aura donc toujours des disjonctions dans l'ensemble de contraintes.

La première stratégie, `LocalConsistencyForECWithCD1`, vérifie la consistance locale de l'ensemble de contraintes élémentaires et elle applique ensuite une seule fois l'approche constructive.

```
stratop
global
        LocalConsistencyForECWithCD1 : < csp -> csp >    bs;
end

strategies for csp
implicit

[] LocalConsistencyForECWithCD1 =>
LocalConsistencyForEC ;
first one (ConstructiveDisjunction , id)
end
```

La deuxième stratégie, `LocalConsistencyForECWithCD2`, vérifie la consistance locale de l'ensemble de contraintes élémentaires et elle applique ensuite tant que possible la disjonction constructive.

```
stratop
global
        LocalConsistencyForECWithCD2 : < csp -> csp >    bs;
end

strategies for csp
implicit

[] LocalConsistencyForECWithCD2 =>
LocalConsistencyForEC ;
repeat* (first one (ConstructiveDisjunction) ; LocalConsistencyForEC)
end
```

Après avoir défini deux implantations de la disjonction constructive, on peut définir plusieurs schémas pour la vérification de consistance locale pour un ensemble de contraintes élémentaires et de contraintes disjonctives. Pour les quatre stratégies que nous allons présenter, le résultat de l'application est toujours un ensemble de conjonctions de contraintes élémentaires localement consistant.

La stratégie `LocalConsistencyForECandDCWithCD1` ci-dessous utilise la disjonction constructive seulement comme un pas de pré-traitement des contraintes disjonctives.

```
stratop
global
        LocalConsistencyForECandDCWithCD1 : < csp -> csp >     bs;
end

strategies for csp
implicit

[] LocalConsistencyForECandDCWithCD1 =>
LocalConsistencyForECWithCD1 ;
repeat* (dk (PostElementaryDisjunct) ; LocalConsistencyForEC)
end
```

Le comportement de cette stratégie est le suivant.

- En appliquant la stratégie `LocalConsistencyForECWithCD1`, on vérifie la consistance locale de tout l'ensemble de contraintes en utilisant une seule fois la notion de disjonction constructive.
- On poste une composante d'une des disjonctions et on vérifie la consistance locale de l'ensemble de contraintes élémentaires ainsi obtenu. Ce pas est itéré jusqu'à ce que toutes les combinaisons des composantes des disjonctions soient analysées.

Évidemment, en utilisant cette stratégie on obtient toutes les combinaisons possibles des composantes des disjonctions. Pour obtenir un seul résultat il suffit de l'appliquer en utilisant un opérateur comme **first one**.

Si dans la phase de pré-traitement de cette stratégie, on applique la disjonction constructive jusqu'à ce que l'on ne puisse plus l'appliquer, alors on obtient la stratégie suivante.

```
stratop
global
        LocalConsistencyForECandDCWithCD2 : < csp -> csp >     bs;
end

strategies for csp
implicit

[] LocalConsistencyForECandDCWithCD2 =>
LocalConsistencyForECWithCD2 ;
repeat* (dk (PostElementaryDisjunct) ; LocalConsistencyForEC)
end
```

Nous avons seulement changé l'application de la stratégie `LocalConsistencyForECWithCD1` par la stratégie `LocalConsistencyForECWithCD2`.

La stratégie `LocalConsistencyForECandDCWithCD3` fait une utilisation plus active de la disjonction constructive.

```
stratop
global
        LocalConsistencyForECandDCWithCD3 : < csp -> csp >     bs;
end

strategies for csp
implicit

[] LocalConsistencyForECandDCWithCD3 =>
LocalConsistencyForECWithCD1 ;
repeat* (dk (PostElementaryDisjunct) ; LocalConsistencyForECWithCD1)
end
```

Une simple modification de la stratégie `LocalConsistencyForECWithCD1` nous a permis d'obtenir cette dernière stratégie: chaque fois que l'on poste une composante d'une disjonction, on vérifie la consistance de tout l'ensemble de contraintes en utilisant la notion de disjonction constructive.

Finalement, la stratégie `LocalConsistencyForECandDCWithCD4` fait une utilisation encore plus active des disjonctions comme il est exprimé ci-dessous.

```
stratop
global
        LocalConsistencyForECandDCWithCD4 : < csp -> csp >     bs;
end

strategies for csp
implicit

[] LocalConsistencyForECandDCWithCD4 =>
LocalConsistencyForECWithCD2 ;
repeat* (dk (PostElementaryDisjunct) ; LocalConsistencyForECWithCD2)
end
```

Cette stratégie est obtenue à partir de la stratégie `LocalConsistencyForECWithCD2`: chaque fois que l'on poste une composante d'une disjonction on vérifie la consistance de tout l'ensemble de contraintes en utilisant la notion de disjonction constructive. Ainsi, la notion de disjonction constructive est utilisée dans le pas de pré-traitement et aussi après chaque envoi d'une des composantes des disjonctions.

La différence entre les quatre stratégies présentées pour la vérification de consistance locale pour un ensemble de contraintes élémentaires et de contraintes disjonctives consiste en le niveau d'utilisation des disjonctions. Puisque l'application de la notion de disjonction constructive est coûteuse et pas toujours payante, on pourrait penser que son utilisation comme un pas de prétraitement est suffisante. Néanmoins, il est intéressant de noter comment, après avoir défini les stratégies adéquates pour son implantation, il est très facile d'appliquer la notion de disjonction constructive à différents degrés en modifiant simplement les stratégies.

Évidemment, on peut intégrer ces stratégies dans un schéma d'énumération exhaustive afin d'obtenir les solution d'un problème. L'implantation de la stratégie ***FullLookaheadWithConstructiveDisjunction***, définie dans la section 5.3.4, est directe.

5.4.4 Exemple

Afin de montrer le traitement des contraintes disjonctives, nous considérons l'ensemble de contraintes ci-dessous utilisé pour le calcul du calendrier égyptien [37]:

```
Ne  =? 365(*)Ye (+) 30(*)Me (+) De &
De <=? 30 &
Me <=? 13 &
Me !=? 12 V De <? 6
Ne  =? 1000
```

Les quatre premières contraintes expriment la relation entre une date absolue (Ne) et l'année (Ye), le mois (Me) et le jour (De) dans le calendrier égyptien. Pour exprimer, par exemple, la date absolue 1000 dans le calendrier égyptien, on ajoute la cinquième contrainte. La trace de l'application de la stratégie **FLAMinimumDomainFirstToLastAll** est la suivante[29]:

```
[] start with term:
CreateCSP(Ne=?365*Ye+30*Me+De & De<=?30 & Me<=?13 & Me!=?12 V De<?6 & Ne=?1000)

'ArcConsistency on constraint 1*Ne=?1000':
CSP[Ne in?[1000,..,1000].De in?[1,..,1000000].Me in?[1,..,1000000].Ye in?[1,..,1000000].nil,
nil,1*Me<=?13.1*De<=?30.1*Ne=?365*Ye+30*Me+1*De.nil,1*Me!=?12 V 1*De<?6.nil,nil]

'Instantiation of variable Ne':
CSP[De in?[1,..,1000000].Me in?[1,..,1000000].Ye in?[1,..,1000000].nil,
Ne=1000.nil,1*Me<=?13.1*De<=?30.1*Ne=?365*Ye+30*Me+1*De.nil,1*Me!=?12 V 1*De<?6.nil,nil]

'Elimination of variable Ne':
CSP[De in?[1,..,1000000].Me in?[1,..,1000000].Ye in?[1,..,1000000].nil,
Ne=1000.nil,1*De<=?30.1*Me<=?13.1000=?365*Ye+30*Me+1*De.nil,1*Me!=?12 V 1*De<?6.nil,nil]

'ArcConsistency on constraint 1*De<=?30':
CSP[De in?[1,..,30].Me in?[1,..,1000000].Ye in?[1,..,1000000].nil,
Ne=1000.nil,1*Me<=?13.1000=?365*Ye+30*Me+1*De.nil,1*Me!=?12 V 1*De<?6.nil,nil]

'ArcConsistency on constraint 1*Me<=?13':
CSP[Me in?[1,..,13].Ye in?[1,..,1000000].De in?[1,..,30].nil,
Ne=1000.nil,1000=?365*Ye+30*Me+1*De.nil,1*Me!=?12 V 1*De<?6.nil,nil]

'ArcConsistency on constraint 1000=?365*Ye+30*Me+1*De':
CSP[Ye in?[2,..,2].Me in?[8,..,13].De in?[1,..,30].nil,
Ne=1000.nil,nil,1*Me!=?12 V 1*De<?6.nil,1000=?365*Ye+30*Me+1*De.nil]

'Instantiation of variable Ye':
CSP[Me in?[8,..,13].De in?[1,..,30].nil,
Ye=2.Ne=1000.nil,nil,1*Me!=?12 V 1*De<?6.nil,1000=?365*Ye+30*Me+1*De.nil]

'ExtractConstraintsOnVar Ye':
CSP[Me in?[8,..,13].De in?[1,..,30].nil,
Ye=2.Ne=1000.nil,1000=?365*Ye+30*Me+1*De.nil,1*Me!=?12 V 1*De<?6.nil,nil]
```

29. Nous initialisons à [1,..,100000] l'ensemble de valeurs possibles pour chaque variable.

```
'Elimination of variable Ye':
CSP[Me in?[8,..,13].De in?[1,..,30].nil,
Ne=1000.Ye=2.nil,1000=?30*Me+1*De+730.nil,1*Me!=?12 V 1*De<?6.nil,nil]

'ArcConsistency on constraint 1000=?30*Me+1*De+730':
CSP[De in?[30,..,30].Me in?[8,..,8].nil,
Ne=1000.Ye=2.nil,nil,1*Me!=?12 V 1*De<?6.nil,1000=?30*Me+1*De+730.nil]

'Instantiation of variable De':
CSP[Me in?[8,..,8].nil,
De=30.Ne=1000.Ye=2.nil,nil,1*Me!=?12 V 1*De<?6.nil,1000=?30*Me+1*De+730.nil]

'ExtractConstraintsOnVar De':
CSP[Me in?[8,..,8].nil,
De=30.Ne=1000.Ye=2.nil,1000=?30*Me+1*De+730.nil,1*Me!=?12 V 1*De<?6.nil,nil]

'Elimination of variable De':
CSP[Me in?[8,..,8].nil,
Ne=1000.Ye=2.De=30.nil,1000=?30*Me+760.1*Me!=?12.nil,nil,nil]

'Instantiation of variable Me':
CSP[nil,Me=8.Ne=1000.Ye=2.De=30.nil,1000=?30*Me+760.1*Me!=?12.nil,nil,nil]

'Elimination of variable Me':
CSP[nil,Ne=1000.Ye=2.De=30.Me=8.nil,nil,nil,nil]

'GetSolutionCSP':
CSP[nil,Ye=2.Me=8.De=30.Ne=1000.nil,nil,nil,nil]

[] result term:
CSP[nil,Ye=2.Me=8.De=30.Ne=1000.nil,nil,nil,nil]
```

Dans l'annexe A.3, nous montrons des résultats expérimentaux de l'application de notre solveur pour la résolution du problème d'ordonnancement disjonctif.

La figure 5.16 montre le comportement de quatre stratégies pour la résolution des problèmes d'ordonnancement disjonctif 3×3 et 5×5. Nous pouvons observer que, d'une part, l'approche *splitting* produit de meilleurs résultats que l'approche *satisfaisabilité à insatisfaisabilité*, et d'autre part, l'ordre d'énumération *first to last* ou *last to first* a une grande influence sur le comportement de l'approche *satisfaisabilité à insatisfaisabilité*.

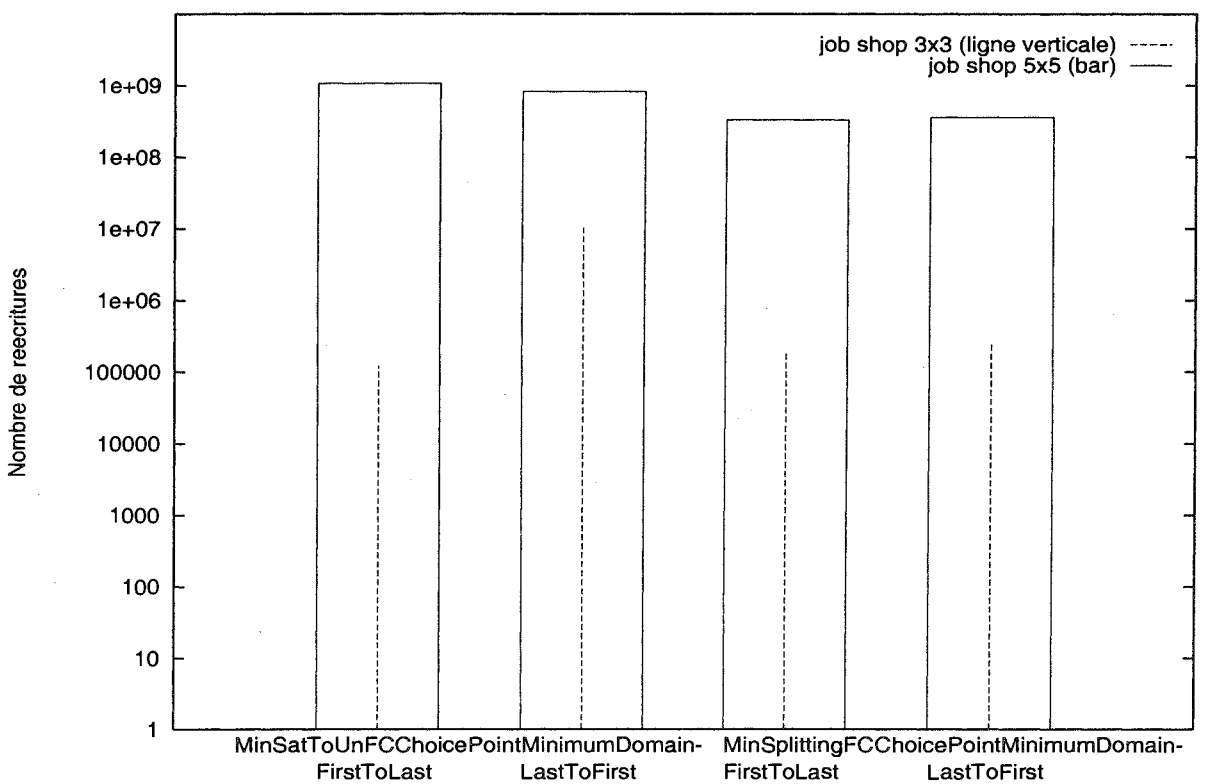

FIG. 5.16 – *Résolution du problème d'ordonnancement disjonctif*

Chapitre 6

Coopération de solveurs

Sommaire

Comme la performance d'un solveur de contraintes dépend fortement de l'utilisation qu'il fait de l'information spécifique concernant les variables et les contraintes, on s'est intéressé, depuis les années quatre-vingt, à intégrer plusieurs solveurs dans un même système pour les faire collaborer afin de profiter au mieux des performances de chacun. Un critère général consiste à faire collaborer des solveurs lorsque les contraintes ne peuvent pas être résolues par un solveur unique ou lorsqu'elles peuvent être résolues d'une manière plus efficace quand on les traite en utilisant plusieurs solveurs [3, 81].

Nous croyons qu'une approche basée sur la programmation par règles et l'utilisation d'un langage de stratégies puissant et flexible peut être d'un très grand intérêt pour la modélisation et l'implantation de la collaboration de solveurs. Pour cette raison, nous avons attaqué le problème de la collaboration de solveurs en utilisant les systèmes de calcul. Nous avons modélisé divers types de collaboration de solveurs d'une manière très abstraite. À partir de ces résultats, il nous semble que le cadre logique utilisé pourrait être très utile pour mieux comprendre cette problématique.

Dans ce chapitre, nous rappelons tout d'abord les concepts de collaboration, de coopération et de combinaison de solveurs. Nous présentons ensuite les opérateurs de stratégies concurrents et les primitives de bas niveau du langage ELAN qui permettent de prototyper la collaboration de plusieurs systèmes de calcul. Finalement, nous utilisons ces facilités pour présenter la mise en œuvre de solveurs qui travaillent sur un même domaine, ce type de collaboration étant appelée coopération de solveurs. À partir de solveurs que nous avons déjà présenté dans les chapitres précédents pour la résolution de CSPs décomposables et de problèmes d'optimisation, nous analysons divers schémas de coopération entre eux.

6.1 Notions de collaboration, coopération et combinaison de solveurs

Le besoin de faire collaborer des solveurs est aujourd'hui largement connu. Vu que la conception et l'implantation de solveurs est en général une tâche très coûteuse, le besoin de la réutilisation d'un solveur, ou de certaines de ses caractéristiques, est évident [106]. Ce critère est notamment pratique, mais un argument plus important encore pour intégrer des solveurs consiste à avoir la possibilité, par la collaboration de solveurs, de résoudre un problème plus efficacement et, dans des cas extrêmes, d'arriver à résoudre des problèmes qui ne peuvent pas être résolus de façon réaliste par un seul solveur [3, 81]. Dans cette section, nous rappelons quelques concepts relatifs à la collaboration de solveurs et nous précisons le type particulier de problème de collaboration auquel on s'intéresse. Les concepts présentés dans cette section sont basés sur la définition du langage pour la collaboration de solveurs **BALI**, proposé par Monfroy initialement dans [84] et mise au point dans [86]. Pour avoir encore plus de détails sur **BALI**, on peut consulter [85], qui par ailleurs nous semble être la référence la plus complète actuellement disponible sur la problématique générale de la collaboration de solveurs.

Coopération de solveurs Informellement, la coopération de solveurs consiste à construire un solveur à partir d'un ensemble de solveurs élémentaires, ou composants, qui sont définis sur un même domaine d'interprétation. Le solveur ainsi construit est donc un solveur qui travaille sur le même domaine que tous les solveurs élémentaires [81, 87]. La coopération de solveurs concerne principalement les problèmes de communication entre solveurs qui appliquent des procédures de résolution différents ou qui acceptent en entrée des contraintes différentes (quant aux symboles de prédicats).

Combinaison de solveurs La combinaison de solveurs consiste à construire un solveur à partir de solveurs élémentaires qui travaillent sur des domaines d'interprétation différents. Un solveur ainsi construit est dédié à une forme d'union des domaines relatifs aux solveurs élémentaires [95, 70, 100]. La combinaison concerne principalement les problèmes qui apparaissent lors du mélange des théories.

La collaboration de solveurs fait référence indistinctement à la coopération et à la combinaison.

Dans le cadre général de la collaboration, nous nous sommes intéressés aux problèmes d'intégration de plusieurs solveurs travaillant toujours sur le même domaine d'interprétation, c'est-à-dire la coopération de solveurs. Nous nous sommes principalement intéressés aux problèmes liés au contrôle de l'application des solveurs du point de vue du langage de stratégies.

L'intégration de solveurs requiert des mécanismes appelés *primitives de collaboration* par Monfroy. La séquentialité, le parallélisme et la concurrence ont été proposées comme primitives de collaboration dans le langage **BALI**.

Séquentialité Lorsque l'on applique un solveur S_1 sur un ensemble de contraintes en entrée et sur le résultat de cette application on applique un solveur S_2, on dit que S_1 et S_2 sont des *solveurs séquentiels.*

Parallélisme Lorsque l'on applique plusieurs solveurs $S_1, \ldots, S_n$ sur des sous-ensembles d'un ensemble de contraintes d'entrée et que les résultats obtenus par l'application de chaque solveur soient composés par des conjonctions, on dit que $S_1, \ldots, S_n$ sont des *solveurs parallèles.*

Il est important de signaler que la notion de solveurs parallèles n'implique rien sur le mode d'exécution temporel des solveurs. Au niveau temporel, ils peuvent être exécutés séquentiellement ou bien en parallèle, cela n'est pas important pour la définition d'un schéma de coopération parallèle. En particulier, dans la section 6.3.1, on verra comment un schéma de coopération parallèle peut être implanté par des solveurs exécutés séquentiellement ou en parallèle.

Concurrence Lorsque l'on applique plusieurs solveurs $S_1, \ldots, S_n$ indépendement sur un même ensemble de contraintes d'entrée et qu'on choisit comme résultat final un seul des résultats obtenus par l'application de chaque solveur, on dit que $S_1, \ldots, S_n$ sont des *solveurs concurrents*.

Il faut signaler à nouveau que cette notion de solveurs concurrents n'implique rien sur le mode d'exécution temporel des solveurs. L'idée de base est la sélection du résultat d'un des solveurs par rapport à un critère. Si on considère des solveurs qui s'exécutent en parallel, un critère naturel est de choisir le résultat du premier solveur qui termine.

6.2 Collaboration de systèmes de calcul dans le langage ELAN

En considérant une architecture parallèle, il est évident que certains opérateurs de stratégies peuvent être parallélisés. L'opérateur **dk** (S_1 , ... , S_n), par exemple, qui retourne tous les résultats de l'application de toutes les sous-stratégies $S_1, \ldots, S_n$, cache évidemment une notion implicite de concurrence: l'application de chaque sous-stratégie S_i ne dépend absolument pas de l'application d'une autre. Par contre, la sémantique de l'opérateur **first** (S_1 , ... , S_n) est strictement séquentielle et le gain apporté par sa parallélisation éventuelle n'est pas évident.

Même si ce point est important, quelques considérations pratiques le sont encore davantage. Quand on s'intéresse, par exemple, à un seul résultat ou à tous les résultats possibles de l'application d'une règle quelconque ℓ_i (ou bien une stratégie quelconque S_i) en utilisant l'opérateur **dc** ($\ell_1, \ldots, \ell_n$) ou l'opérateur **dc one** ($\ell_1, \ldots, \ell_n$), respectivement, on court le risque de ne jamais les obtenir. Dans l'implantation séquentielle actuelle de ces opérateurs dans le langage ELAN, il peut se produire que l'on essaie d'appliquer une règle ℓ_i qui ne termine pas. Dans ce cas, le processus de calcul tout entier va attendre indéfiniment. En pratique, l'utilisateur doit prévoir cette situation et soit être sûr que toutes les sous-stratégies terminent, soit essayer de les appliquer dans un ordre qui assure que le processus terminera toujours.

En plus, si on prend en compte les considérations d'efficacité, même si toutes les sous-stratégies terminent, la performance du processus de calcul dépend de l'ordre dans lequel elles sont essayées. Évidemment, quand les résultats de toutes les sous-stratégies sont équivalents, l'utilisateur a intérêt à essayer d'abord celle qui est la plus efficace. En utilisant les opérateurs disponibles dans la version actuelle du langage ELAN, la seule façon pour l'utilisateur d'attaquer cette situation est d'appliquer un opérateur **first** ou un opérateur **first one**. Néanmoins, l'ordre de spécification des sous-stratégies dépendra toujours d'une estimation *a priori* de leur efficacité.

D'ailleurs, lorsqu'on s'intéresse à intégrer plusieurs systèmes de calcul afin de les faire collaborer pour résoudre ensemble un problème, on a besoin de facilités de communication pour contrôler l'interaction entre les différents systèmes de calcul. Cet intérêt n'est pas purement théorique, le besoin de l'intégration de solveurs de contraintes est devenu de plus en plus clair lorsque l'utilisation d'un seul solveur ne donne pas de résultats ou bien si sa performance est inférieure à celle d'un ensemble de solveurs travaillant en collaboration.

La possibilité d'utiliser des opérateurs de stratégies concurrents et des facilités de communication avec des processus externes nous donne encore certainement une plus grande flexibilité de prototypage.

En tenant compte de ces considérations, nous avons participé au développement des nouvelles facilités de contrôle de processus et des opérateurs de stratégies concurrents incorporés récemment dans ELAN. Ces extensions à deux niveaux du langage nous ont permis de prototyper la coopération de solveurs de contraintes par l'intégration de plusieurs systèmes de calcul et par l'application des opérateurs de stratégies concurrents toujours dans le même cadre de la réécriture [12]. L'implantation de ces opérateurs est pour l'instant en court de prototypage et ils ne peuvent pas être encore compilés. Dans cette section, nous décrivons brièvement ces nouvelles facilités du langage ELAN.

6.2.1 Stratégies concurrentes

Le langage de stratégies élémentaires d'ELAN est étendu par les opérateurs suivants

- choix concurrent de tous les résultats de toutes les branches : **dk**
- choix concurrent d'un résultat de chaque branche : **dk one**
- choix concurrent de tous les résultats d'une branche : **dc**
- choix concurrent d'un résultat : **dc one**
- déclenchement d'un processus externe pour obtenir tous les résultats : **dkcall**
- déclenchement d'un processus externe pour obtenir un seul résultat : **dccall**

La syntaxe et la sémantique opérationnelle de ces nouveaux opérateurs de stratégies élémentaires, de façon informelle, sont les suivantes.

dk La stratégie **dk** (S_1 || ... || S_n) donne tous les résultats de l'application de toutes les stratégies $S_1, \ldots, S_n$. C'est-à-dire elle a la même sémantique que sa version séquentielle, mais les stratégies $S_1, \ldots, S_n$ sont appliquées simultanément. Si toutes les stratégies S_i échouent alors la stratégie **dk** échoue.

dk one La stratégie **dk one** (S_1 || ... || S_n) donne le premier résultat de l'application de chaque stratégie $S_1, \ldots, S_n$. C'est-à-dire elle a la même sémantique que sa version séquentielle mais les stratégies $S_1, \ldots, S_n$ sont appliquées simultanément. Si toutes les stratégies S_i échouent alors la stratégie **dk one** échoue.

dc La stratégie **dc** (S_1 || ... || S_n) donne tous les résultats de l'application d'une des stratégies $S_1, \ldots, S_n$. Sa sémantique est donc la même que sa version séquentielle, mais les stratégies $S_1, \ldots, S_n$ sont calculées simultanément. Le critère d'implantation choisit la stratégie appliquée comme étant la première à retourner un résultat. Si toutes les stratégies S_i échouent alors la stratégie **dc** échoue.

dc one La stratégie **dc one** (S_1 || ... || S_n) donne le premier résultat de l'application d'une des stratégies $S_1, \ldots, S_n$. Sa sémantique est donc la même que sa version séquentielle, mais les stratégies $S_1, \ldots, S_n$ sont calculées simultanément. Le critère d'implantation choisit le résultat à partir de la première stratégie qui retourne un résultat. Si toutes les stratégies S_i échouent alors la stratégie **dc one** échoue.

dkcall La stratégie **dkcall** (P) appelle un processus externe, identifié par P, et retourne tous les résultats de son application. Si le processus retourne un ensemble vide de résultats, alors la stratégie **dkcall** échoue.

dccall La stratégie **dccall** (P) appelle un processus externe, identifié par P, et retourne le premier résultat obtenu. Si le processus retourne un ensemble vide de résultats, alors la stratégie **dccall** échoue.

L'implantation de la stratégie **dc** $(S_1 \mid\mid \ldots \mid\mid S_n)$ mérite un commentaire. Actuellement les stratégies $S_1, \ldots, S_n$ sont lancées en parallèle et dès qu'une d'entre elles retourne un résultat toutes les autres sont arrêtées. On attend alors tous les résultats de la stratégie choisie. Ceci est fait pour des raisons d'efficacité, mais il faut considérer deux situations importantes.

- La stratégie choisie pourrait ne pas être la plus efficace. Elle est la première à retourner un résultat mais une des stratégies qui sont arrêtées pourrait être plus efficace pour retourner tous les résultats, même si ce n'est pas le cas quand on considère seulement le premier résultat.
- Plus grave encore, la stratégie choisie, en se basant seulement sur le premier résultat, pourrait ne jamais terminer. Dans ce cas, tout le calcul ne terminera pas, même s'il existe une autre stratégie terminante mais qui a été arrêtée.

Évidemment une solution a ces deux problèmes est de récupérer les résultats de toutes les stratégies et choisir finalement la première à retourner tous ses résultats, c'est-à-dire la première qui termine. Malheureusement cette solution pose des problèmes d'efficacité quant à la gestion de la mémoire et une implantation satisfaisante devrait être étudiée dans le futur.

Une explication détaillée des opérateurs **dk**, **dk one**, **dc** et **dc one** se trouve dans [11]. Pour une explication plus précise des opérateurs **dccall** et **dkcall** on peut consulter [113].

En utilisant ces opérateurs nous pouvons maintenant réaliser un prototypage plus flexible, par exemple

- l'exécution séquentielle de deux processus externes P_1 et P_2 peut être réalisée par

$$\textbf{dkcall}\ (P_1)\ ;\ \textbf{dkcall}\ (P_2)$$

  qui, à chaque résultat retourné par P_1, applique le processus P_2 ;
- le choix du premier résultat de l'application d'un des processus externes P_1 et P_2, qui sont lancés en concurrent, est exprimé par

$$\textbf{dc one}\ (\textbf{dccall}\ (P_1) \mid\mid \textbf{dccall}\ (P_2))$$

Néanmoins, pour certaines applications, comme celles que l'on montrera dans la section 6.3, nous avons besoin d'opérateurs encore plus fins afin de contrôler l'interaction des systèmes de calcul d'une manière adéquate. Ces opérateurs sont appelés primitives de bas niveau.

6.2.2 Primitives de bas niveau

Nous décrivons par la suite les possibilités de contrôle de processus par l'utilisation de primitives de bas niveau. Ces primitives consistent en des opérations similaires à celles du système d'exploitation UNIX pour la création de processus et la communication avec eux via des canaux. Les primitives ont été intégrées de manière uniforme avec les opérations d'entrée/sortie du langage ELAN.

Le module `stdio` de la librairie ELAN contient quatre primitives pour l'ouverture et la fermeture de fichiers et pour la création et la destruction de processus. Elles sont spécifiées ci-dessous, où la sorte *Pid* représente l'identification d'un processus ou d'un fichier.

- La primitive **create** (`id`) crée un processus bloquant, identifié par la chaîne de caractères `id`, en utilisant deux canaux de communication.

$$\textbf{create}\ (@)\ :\ (\mathit{string})\ \mathit{Pid}$$

- La primitive **create_noblock** (`id`) crée un processus non-bloquant, identifié par la chaîne de caractères `id`, en utilisant deux canaux de communication.

$$\textbf{create\_noblock}\ (@)\ :\ (\mathit{string})\ \mathit{Pid}$$

- La primitive **open** (`id1`,`id2`) ouvre un fichier, identifié par la chaîne de caractères `id1`, pour lecture, écriture ou ajout, selon la chaîne de caractères `id2` spécifiée.

$$\textbf{open}\ (@,@)\ :\ (\mathit{string}\ \mathit{string})\ \mathit{Pid}$$

- La primitive **close** (`pid`) ferme un fichier ou détruit un processus identifié par le terme `pid` de sorte *Pid*.

$$\textbf{close}\ (@)\ :\ (\mathit{Pid})\ \mathit{Pid}$$

Une fois qu'un fichier est ouvert ou un processus est créé, la communication avec lui est réalisée en utilisant les primitives définies dans le module paramétré `io[X]` de la librairie ELAN. La spécification de ces primitives est présentée ci-dessous, où le paramètre X désigne la sorte de la donnée communiquée, laquelle est enrichie par une valeur d'erreur définie de la manière suivante

$$\textbf{Error}\ (@)\ :\ (\mathit{int})\ X$$

- La primitive **write** (`pid`,`t`) envoie un terme `t` au fichier ou au processus, identifié par `pid`, et retourne soit le même terme `t` soit un terme d'erreur **Error** (**errno**).

$$\textbf{write}\ (@,@)\ :\ (\mathit{Pid}\ X)\ X$$

- La primitive **read** (`pid`) retourne soit un terme lu d'un fichier ou d'un processus, identifié par `pid`, soit un terme d'erreur.

$$\textbf{read}\ (@)\ :\ (\mathit{Pid})\ X$$

  Lorsque l'on utilise la primitive **read** pour communiquer avec un processus qui a été créé par une primitive **create_noblock**, il peut se produire qu'il n'y a rien à lire. Dans ce cas l'essai de lecture retourne un terme d'erreur `No_term`, mais le processus de calcul continue. Par contre, si le processus a été créé par une primitive **create**, on reste en attente, on bloque le processus de calcul, jusqu'à obtenir un résultat.
- La primitive **iserror** (t) retourne `true` s'il y a une erreur.

$$\textbf{iserror}\ (@)\ :\ (X)\ \mathit{bool}$$

- La primitive **errno** (t) retourne le numéro de l'erreur.

$$\textbf{errno}\ (@)\ :\ (X)\ \mathit{int}$$

  Il y a plusieurs codes d'erreur **errno** pre-définis dans le module `io[X]`. Par exemple, `No_more` désigne la fin d'un fichier ou d'un processus, `No_term` indique qu'il n'y a pas de terme disponible dans un canal associé à un processus non-bloquant.

Dans la section 6.3, nous illustrons quelques applications de ces deux types de primitives d'ELAN par la mise en œuvre de plusieurs schémas de coopération de solveurs de contraintes.

6.3 Applications de la coopération de solveurs

Dans cette section, nous présentons plusieurs schémas de coopération entre des solveurs qui ont été définis dans les chapitres précédents. Tout d'abord, nous réalisons que la résolution de CSPs décomposables, présentée dans la section 3.6.6, rentre déjà dans le cadre de la coopération de solveurs en parallèle. Néanmoins, nous utilisons maintenant des primitives de bas niveau pour permettre une majeure flexibilité lors de la résolution de ce type de problèmes, toujours dans le même cadre de la coopération de solveurs en parallèle. Ensuite, nous montrons comment divers schémas de coopération entre les solveurs définis dans la section 4.4 nous permettent de mieux attaquer la résolution de CSOPs. En utilisant des opérateurs de stratégies concurrents nous modélisons la coopération de solveurs séquentiels et concurrents et en utilisant des primitives de bas niveau nous présentons un schéma de coopération qui nous permet encore plus de flexibilité.

6.3.1 CSPs décomposables

Dans la section 3.6.6, nous avons présenté la stratégie `SolveCSPDis`, pour le traitement de CSPs décomposables, laquelle est rappelée ci-dessous.

```
[] SolveCSPDis =>
first one (DecomposeCSPDis);
repeat* (first one (SolveSubCSPDis));
repeat* (first one (ComposeCSPDis))
end
```

En utilisant cette stratégie nous décomposons un CSP en plusieurs sous-problèmes, on résout ensuite chaque sous-problème et finalement, à partir de ces solutions, on compose la solution au problème initial. Cette stratégie rentre déjà dans le cadre de la coopération de solveurs en parallèle: la solution du CSP est obtenue par conjonction des solutions des sous-problèmes. Les caractéristiques les plus importantes de ce solveur sont que tout le travail est réalisé par le même processus de calcul et que la résolution des sous-problèmes est réalisée d'une manière séquentielle. Dans cette section, on présente deux stratégies qui utilisent des solveurs externes pour la résolution des sous-problèmes ce qui nous permet de mieux partager la charge de travail et, plus important encore, cela nous permet de résoudre les sous-problèmes en parallèle avec un gain évident en efficacité. Ces deux stratégies rentrent toujours dans le cadre de la coopération de solveurs parallèles.

Pour l'implantation des deux stratégies, la structure de données `cspdis`, utilisée dans la section 3.6.6, est étendue de la manière suivante.

```
operators
global
  CSPDisII@ : (set:tuple[csp, list[csp], list[csp], list[Pid], list[Pid]]) cspdisII;
end
```

Les trois premières composantes sont utilisées de la même façon que dans la section 3.6.6: stockage du CSP initial, liste de sous-problèmes à résoudre et liste de sous-problèmes résolus. On ajoute simplement une liste contenant l'identification des processus qui s'exécutent et une liste contenant l'identification des processus en attente, dans la quatrième et la cinquième composante, respectivement.

Pour la création des processus externes on utilise la règle `CreateSolverCSPDisIIBlock` présentée ci-dessous.

```
rules for cspdisII
        P            : csp;
        pid          : Pid;
        luCsp, lsCsp : list[csp];
        laPid, liPid : list[Pid];
global

[CreateSolverCSPDisIIBlock]
        CSPDisII[P,luCsp,lsCsp,laPid,liPid]
        =>
        CSPDisII[P,luCsp,lsCsp,laPid,pid.liPid]
        if      size(luCsp) > size(liPid)
        where   pid := ()create("solverCSPDisII")
        if      not iserr(pid)
end
```

Comme on a intérêt à associer à chaque sous-problème un solveur externe, nous vérifions en utilisant la règle `CreateSolverCSPDisIIBlock` si le nombre de processus externes déjà créé, initialement zéro, est inférieure aux besoins, c'est-à-dire le nombre de sous-problèmes à résoudre. Dans ce cas, on crée un processus appelé `solverCSPDisII` qui correspond à un programme ELAN spécifiant un solveur de conjonctions de contraintes élémentaires et on ajoute son identification (`pid`) à la liste de processus en attente.

La règle `SendToSolverCSPDisII` prend le premier élément (`Q`) de la liste de sous-problèmes à résoudre et l'envoie au solveur identifié par le premier élément (`pid`) de la liste de solveurs en attente. L'identification de ce processus est placée dans la liste de processus actifs.

```
rules for cspdisII
        P, Q, R      : csp;
        pid          : Pid;
        string1      : string;
        luCsp, lsCsp : list[csp];
        laPid, liPid : list[Pid];
global

[SendToSolverCSPDisII]
        CSPDisII[P,Q.luCsp,lsCsp,laPid,pid.liPid]
        =>
        CSPDisII[P,luCsp,lsCsp,pid.laPid,liPid]
        where   R       := ()write(pid,Q)
        where   string1 := ()write(pid,"\nend\n")
        if      not iserror(R) and not iserror(string1)
end
```

Il faut noter que l'application de cette règle requiert d'une part l'existence d'un sous-problème à résoudre et d'autre part de l'existence d'un processus en attente.

Afin de récupérer les résultats d'un solveur, on définit la règle `GetFirstOutputOfSolverCSP-DisII` comme ci-dessous.

```
rules for cspdisII
        P, Q          : csp;
        pid           : Pid;
        luCsp, lsCsp  : list[csp];
        laPid, liPid  : list[Pid];
global

[GetFirstOutputOfSolverCSPDisII]
        CSPDisII[P,luCsp,lsCsp,pid.laPid,liPid]
        =>
        CSPDisII[P,luCsp,Q.lsCsp,laPid,pid.liPid]
        where   Q := ()read(pid)
        if not iserror (Q)
end
```

Comme les processus externes sont créés en utilisant le mode bloquant, cette règle essaie d'obtenir un résultat du solveur identifié par le premier élément (`pid`) de la liste de processus actifs jusqu'à ce qu'elle en obtient un, c'est-à-dire que tout le processus de calcul reste en attente. Le résultat obtenu (`Q`) est ajouté à la liste de CSPs résolus (`lsCsp`) et l'identification du processus est ajoutée à la liste de processus en attente (`liPid`).

La règle `CloseSolverCSPDisII`, présentée ci-dessous, ferme le processus identifié par le premier élément de la liste de processus actifs.

```
rules for cspdisII
        P             : csp;
        pid, pid1     : Pid;
        luCsp, lsCsp  : list[csp];
        laPid, liPid  : list[Pid];
global

[CloseSolverCSPDisII]
        CSPDisII[P,luCsp,lsCsp,laPid,pid.liPid]
        =>
        CSPDisII[P,luCsp,lsCsp,laPid,liPid]
        where   pid1 := ()close(pid)
end
```

Finalement, la stratégie `SolveCSPDisIIBlock` exprime un schéma de coopération utilisant des solveurs en parallèle pour la résolution de CSPs décomposables.

```
stratop
global
        SolveCSPDisIIBlock : < cspdisII -> cspdisII >     bs;
end

strategies for cspdisII
implicit

[] SolveCSPDisIIBlock =>
first one (DecomposeCSPDisII);
repeat* (first one (CreateSolverCSPDisIIBlock));
repeat* (first one (SendToSolverCSPDisII));
repeat* (first one (GetFirstOutputOfSolverCSPDisII));
repeat* (first one (ComposeCSPDisII));
repeat* (first one (CloseSolverCSPDisII))
end
```

Le comportement de cette stratégie est le suivant

- on décompose le CSP initial en plusieurs sous-problèmes ;
- on crée un nombre de solveurs externes, tous identiques, égal au nombre de sous-problèmes créés ;
- on envoie séquentiellement chaque sous-problème à un solveur externe ;
- on lit séquentiellement le résultat de chaque solveur externe ;
- à partir des solutions des sous-problèmes, on compose la solution au problème initial.

Les règles de décomposition et composition d'un CSP sont similaires à celles présentées dans la section 3.6.6, sauf que dans ce cas elles considèrent la structure de données `cspdisII` au lieu de la structure de données `cspdis`.

Concernant ce schéma de coopération, deux remarques sont importantes.

- Si le CSP initial est décomposé en des sous-problèmes $csp_1, \ldots, csp_n$ qui sont stockés dans une liste $csp_1 \ldots . csp_n.nil$, les sous-problèmes sont envoyés aux solveurs dans l'ordre $csp_1, \ldots, csp_n$, mais leurs résultats seront lus dans l'ordre $csp_n, \ldots, csp_1$: le dernier sous-problème à être envoyé est le premier à être lu. On pourrait penser à un autre mécanisme pour gérer les listes, car on peut supposer que le plus logique est de récupérer d'abord le résultat du solveur que fut le premier à être lancé. Néanmoins, vu que l'on ne considère pas la taille des sous-problèmes, ce qui pourrait nous donner un indice sur la complexité de leur résolution, il n'est pas évident que cela soit une bonne solution.
- Plus important est la façon de lire les résultats : chaque fois que l'on essaie de lire les résultats provenant d'un solveur on attend jusqu'à ce que l'on en obtient un. Considérant la nature bloquante de ce schéma de coopération, la complexité du processus de calcul est donc égale à la somme des complexités de tous les sous-problèmes.

Une solution plus efficace consiste à utiliser un schéma non-bloquant où on essaie de lire les résultats d'un solveur et s'il n'y en a pas on essaie de lire les résultats d'un autre solveur. De cette façon on peut attendre une complexité du processus de calcul égale à la complexité de la résolution du sous-problème le plus difficile à résoudre.

Pour implanter ce schéma de coopération il nous faut simplement modifier la règle pour la création des processus externes et ajouter une règle pour gérer les priorités dans la liste de

processus actifs. La règle `CreateSolverCSPDisIINoBlock`, présentée ci-dessous, est similaire à la règle `CreateSolverCSPDisIIBlock` sauf que le processus `solverCSPDisII` est créé en mode non-bloquant grâce à la primitive `create_noblock`.

```
rules for cspdisII
        P              : csp;
        pid            : Pid;
        luCsp, lsCsp   : list[csp];
        laPid, liPid   : list[Pid];
global

[CreateSolverCSPDisIINoBlock]
        CSPDisII[P,luCsp,lsCsp,laPid,liPid]
        =>
        CSPDisII[P,luCsp,lsCsp,laPid,pid.liPid]
        if      size(luCsp) > size(liPid)
        where   pid := ()create_noblock("solverCSPDisII")
        if      not iserr(pid)
end

end
```

Pour gérer la liste de processus actifs on définit la règle `ChangePriority` comme suit, laquelle simplement envoie à la fin de la liste des processus actifs le premier élément.

```
rules for cspdisII
        P            : csp;
        pid          : Pid;
        luCsp, lsCsp : list[csp];
        laPid, liPid : list[Pid];
global

[ChangePriority]
        CSPDisII[P,luCsp,lsCsp,pid.laPid,liPid]
        =>
        CSPDisII[P,luCsp,lsCsp,append(laPid,pid.nil),liPid]
end

end
```

Ainsi, on peut maintenant définir la stratégie `SolveCSPDisIINoBlock` comme ci-dessous.

```
stratop
global
        SolveCSPDisIINoBlock : < cspdisII -> cspdisII >     bs;
end

strategies for cspdisII
implicit

[] SolveCSPDisIINoBlock =>
first one (DecomposeCSPDisII);
repeat* (first one (CreateSolverCSPDisIINoBlock));
repeat* (first one (SendToSolverCSPDisII));
repeat* (first one (GetFirstOutputOfSolverCSPDisII , ChangePriority));
repeat* (first one (ComposeCSPDisII));
repeat* (first one (CloseSolverCSPDisII))
end
end
```

Le comportement de ce schéma de coopération de solveurs en parallèle est le suivant

- on décompose le CSP initial en plusieurs sous-problèmes ;
- on crée un nombre de solveurs externes, tous identiques, égal au nombre de sous-problèmes créés ;
- on envoie séquentiellement chaque sous-problème à un solveur externe ;
- on essaie de lire le résultat du solveur identifié par le premier élément de la liste de solveurs actifs, s'il ne donne pas encore de résultat on le met à la fin de la liste et on essaie le suivant, et ainsi de suite jusqu'à obtenir le résultat de chaque solveur ;
- à partir des solutions des sous-problèmes, on compose la solution au problème initial.

Dans cette section, nous avons présenté l'implantation de deux solveurs qui utilisent un nombre *a priori* indéterminé de solveurs externes, tous identiques, pour la résolution de CSPs décomposables. Ces deux solveurs rentrent dans le cadre de la coopération de solveurs en parallèle. Le premier solveur a été implanté en utilisant un mode bloquant pour la communication avec les solveurs externes. Nous avons implanté le deuxième solveur par une simple modification de la règle chargée de la création des processus externes, afin de spécifier un mode de communication non-bloquant, et l'ajout d'une règle pour gérer les priorités entre les processus externes. L'implantation de ces deux solveurs nous a permis de montrer l'intérêt de disposer de facilités de communication avec de processus externes : sans aucune modification nous avons pu réutiliser un solveur déjà implanté pour la résolution de conjonctions de contraintes élémentaires. Dans les deux cas présentés dans cette section, les solveurs externes ont été toujours des programmes ELAN, mais il est intéressant de signaler qu'ils pouvaient très bien être de solveurs implantés dans un autre langage.

6.3.2 Optimisation

Comme nous l'avons signalé dans la section 4.5, la distance entre la borne supérieure initiale et la localisation de la solution optimale a une grande influence sur la performance des deux solveurs présentés pour la résolution de CSOPs. L'un, `MinSatToUnsat`, présente un bon comportement quand la solution optimale se trouve proche de la borne supérieure, mais elle est inefficace quand la solution optimale est localisée loin de la borne supérieure. L'autre, `MinSplitting`, converge

très rapidement quand la solution optimale est loin de la borne supérieure, mais elle fait beaucoup de calculs quand la solution optimale est proche de la borne supérieure. Une idée naturelle donc est de les faire coopérer de façon telle de profiter des avantages de chacun et d'éviter leurs désavantages. Dans cette section, nous traitons la résolution de CSOPs en faisant coopérer ces deux solveurs suivant des schémas séquentiel et concurrent.

En analysant les stratégies `MinSatToUnsat` et `MinSplitting`, on peut se rendre compte que la seule différence entre elles consiste en la boucle **repeat***, mais toutes les deux réalisent un pas d'initialisation des bornes. En fait, la stratégie `MinSatToUnsat` n'initialise pas la borne inférieure, cette borne est affectée d'une valeur d'entrée qui n'est modifiée que lorsque l'on trouve la solution optimale. Mais, afin d'avoir la même structure générale que la stratégie `MinSplitting`, on peut ajouter une estimation initiale de la borne inférieure qui ne modifie pas le comportement général de la stratégie `MinSatToUnsat`. En factorisant ce pas initial, on obtient de nouvelles versions de ces deux stratégies qui nous allons détailler.

Nous commençons par définir la sous-stratégie `SetUpperAndLowerBounds` qui réalise le pas d'initialisation commun aux deux stratégies.

```
stratop
global
      SetUpperAndLowerBounds : < csop -> csop >     bs;
end

[] SetUpperAndLowerBounds =>
PostLTEUpperBound ;
LocalConsistencyForEC ;
GetNewUpperBound ;
UndervalueLowerBound
end
```

Ainsi, on peut maintenant exprimer la stratégie `MinSatToUnsat` de la manière suivante, où la sous-stratégie `strat1` décrit le pas réalisé dans chaque itération de la boucle **repeat***.

```
stratop
global
      MinSatToUnsat : < csop -> csop >     bs;
      strat1        : < csop -> csop >     bs;
end

[] MinSatToUnsat =>
SetUpperAndLowerBounds ;
repeat* (strat1) ;
FCFirstToLastOne ;
GetSolutionCSOP
end

[] strat1 =>
first one (PostLTUpperBound ;
           LocalConsistencyForEC ;
           GetNewUpperBound
           ,
           SetLowerBoundToUpperBound
          )
end
```

Nous exprimons également la stratégie **MinSplitting** comme ci-dessous, où la sous-stratégie **strat2** décrit le pas réalisé dans chaque itération de la boucle **repeat***.

```
stratop
global
      MinSplitting : < csop -> csop >     bs;
      strat2       : < csop -> csop >     bs;
end

[] MinSplitting =>
SetUpperAndLowerBounds ;
repeat* (strat2) ;
FCFirstToLastOne ;
GetSolutionCSOP
end

[] strat2 =>
first one (SetUpperBoundToMiddle ;
           PostLTUpperBound ;
           LocalConsistencyForEC ;
           GetNewUpperBound ;
           UndervalueLowerBound
           ,
           SetLowerBoundToMiddle
          )
end
```

Comme la seule différence entre les solveurs **MinSatToUnsat** et **MinSplitting** consiste en l'utilisation des sous-stratégies **strat1** et **strat2**, on considère désormais les sous-stratégies **strat1** et **strat2** comme deux solveurs élémentaires décrivant l'essentiel des solveurs originaux. Par la suite, on présente plusieurs schémas de coopération entre ces deux solveurs élémentaires.

Un premier schéma de coopération entre les solveurs **strat1** et **strat2** est exprimé par la stratégie **CooperationI**.

```
stratop
global
      CooperationI : < csop -> csop >     bs;
end

[] CooperationI =>
SetUpperAndLowerBounds ;
repeat* (strat1 ; first one (strat2 , id)) ;
FCFirstToLastOne ;
GetSolutionCSOP
end
```

Cette stratégie applique successivement les solveurs **strat1** et **strat2** : la sortie d'un solveur devient l'entrée de l'autre. La sous-stratégie **first one (strat2 , id)** mérite une explication.

- Si le solveur **strat1** trouve une solution qui n'est pas optimale on pourra donc toujours appliquer le solveur **strat2**.
- Si le solveur **strat1** trouve la solution optimale, alors l'application du solveur **strat2** va échouer car il ne peut pas améliorer cette solution. Si le solveur **strat2** échoue, toute la concaténation **strat1 ; first one (strat2 , id)** échoue et ainsi on perd la solution

optimale obtenue par le solveur `strat1`. Donc afin d'éviter cette situation on ajoute une stratégie `id` qui nous permet de finir la concaténation sans échouer dans le cas où le solveur `strat1` trouve la solution optimale. La prochaine fois que l'on essaie la stratégies `strat1` elle va échouer mais nous avons déjà gardé la solution optimale.

En utilisant la stratégie `CooperationI` tout le travail est fait par le même processus de calcul, le solveur est consideré commme un tout. En appliquant une approche plus modulaire, on peut considérer les solveurs élémentaires comme étant externes et même implantés dans un autre langage. Cette approche a l'avantage évidente de faciliter la reutilisation des solveurs élémentaires. Dans le schéma de coopération suivant, nous utilisons ces solveurs élémentaires comme des processus externes. À ce fin, nous définissons deux programmes ELAN externes, appelés `solver1CSOP` et `solver2CSOP`, qui appliquent les stratégies `strat1` et `strat2`, respectivement. La stratégie `CooperationII` implante cette idée.

```
stratop
global
      CooperationII : < csop -> csop >      bs;
end

[] CooperationII =>
SetUpperAndLowerBounds ;
repeat* (dccall (solver1CSOP) ; first one (dccall (solver2CSOP) , id)) ;
FCFirstToLastOne ;
GetSolutionCSOP
end
```

Le comportement de la stratégie `CooperationII` est similaire à celui de la stratégie `CooperationI`, les solveurs élémentaires sont exécutés toujours en séquentiel: $S_1; S_2; S_1; S_2; \ldots$. On est donc dans le cadre défini par Monfroy [85] des solveurs séquentiels. Dans notre cas, ce schéma a l'inconvénient évident de laisser un solveur inactif lorsque l'autre fait des calculs. Plus important encore, dû à la complexité exponentielle des problèmes considerés, il peut se produire que tout le processus de calcul reste en attente parce qu'un solveur ne trouve pas de solution. Nous voyons ici l'importance de l'ordre d'application des sous-stratégies. Même si un des solveurs pourrait être appliqué et retourner un résultat, le processus de calcul peut être bloqué à cause de l'application d'une sous-stratégie trop coûteuse.

Pour éviter cette situation, on pourrait envisager une exécution concurrente des solveurs: on lance les deux solveurs puis, dès que l'un des solveurs trouve une solution, on la récupère et on arrête l'autre. À partir de la nouvelle solution, on met à jour les bornes et on relance à nouveau les deux solveurs. La stratégie `CooperationIII` implante ce schéma de coopération:

```
stratop
global
      CooperationIII : < csop -> csop >      bs;
end

[] CooperationIII =>
SetUpperAndLowerBounds ;
repeat* (dc one (strat1 || strat2)) ;
FCFirstToLastOne ;
GetSolutionCSOP
end
```

Nous nous trouvons à nouveau dans une situation où tout le travail est fait par le même

processus de calcul. En plus, dans ce cas les deux solveurs tournent en même temps et par conséquent la charge du processus de calcul est plus importante. Puisque l'on peut utiliser les solveurs comme des processus externes, on définit donc la stratégie `CooperationIV` afin de mieux partager le travail.

```
stratop
global
      CooperationIV : < csop -> csop >      bs;
end

[] CooperationIV =>
SetUpperAndLowerBounds ;
repeat* (dc one (dccall (solver1CSOP) || dccall (solver2CSOP))) ;
FCFirstToLastOne ;
GetSolutionCSOP
end
```

En utilisant les stratégies `CooperationIII` et `CooperationIV`, un solveur ne va jamais attendre une solution venant de l'autre pour pouvoir travailler. Par exemple, une trace possible de l'application de ces stratégies est : $S_2; S_2; S_1; S_2; S_1; \ldots$. Dans un cas extrême, où toutes les solutions consécutives viennent du même solveur jusqu'à ce que l'on trouve la solution optimale, la performance de ce solveur sera équivalente à sa performance lorsqu'il est lancé indépendamment. Le processus de calcul global sera bloqué uniquement dans le cas où tous les solveurs sont bloqués. Ce schéma de coopération rentre donc dans le cadre des solveurs concurrents défini par Monfroy [85].

Néanmoins, dans certains cas, il peut se produire que lorsqu'un solveur retourne une solution on arrête un autre solveur qui est pourtant en train de calculer, plus lentement, une meilleure solution que celle retournée par le premier solveur. Pour mieux comprendre cette situation nous nous ramenons au problème de transition de phase.

L'étude du phénomène de transition de phase est l'approche pratique de la théorie de la complexité des algorithmes. À partir des résultats de la résolution pratique de problèmes NP-difficiles, on s'est rendu compte qu'il existe un grand nombre de tels problèmes qui peuvent être résolus très efficacement [119]. Au cours des années, de nombreuses recherches ont été menées pour vérifier l'existence de ce phénomène dans le cas spécifique des CSPs. Le travail de Cheeseman, Kanefsky et Taylor semble être la première référence spécifiquement dédiée à ce problème [29]. Les travaux de Smith pour identifier la zone où se trouvent les problèmes les plus difficiles sont aussi une des premières références [104, 105]. Dans l'hypothèse où ce phénomène dépend de la nature du problème, Grant et Smith ont étudié le cas particulier de la transition de phase lors de la vérification de consistance d'arc [52] et de la vérification de consistance de chemin [53]. D'ailleurs, Kwan, Tsang et Borrett ont étudié le problème de la recherche de toutes les solutions d'un CSP [73]. Finalement, on peut aussi citer les travaux de Gent et Walsh sur l'étude du phénomène [50, 51].

Nous expliquons ce phénomène à partir de la figure 6.1 qui s'inspire du travail de Smith [105].

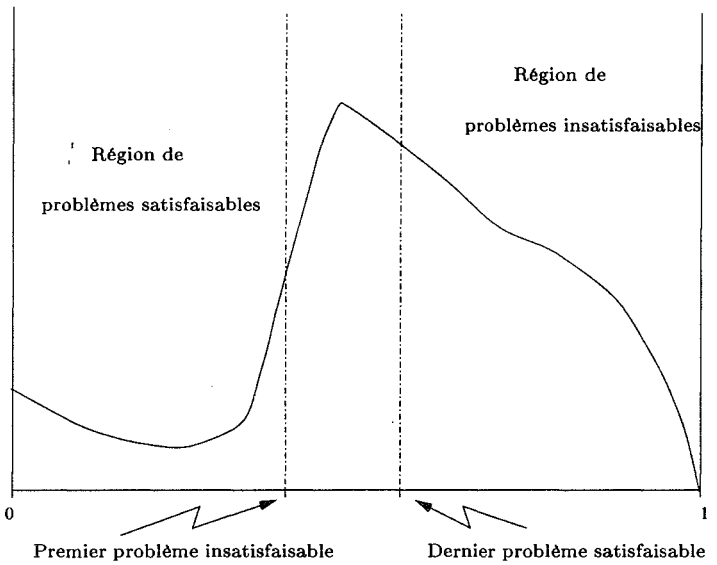

FIG. 6.1 – *Phénomène de transition de phase*

La courbe représente le coût de trouver une solution à un échantillon de CSPs binaires en fonction du niveau de restrictivité de l'ensemble de contraintes. L'axe horizontal représente le niveau de restrictivité des contraintes : le 0 indique un niveau de restrictivité nul ; le 1 indique un niveau de restrictivité maximal. L'axe vertical représente le coût pour trouver une solution ou démontrer que le problème n'a pas de solution (ce coût est normalement mesuré d'après le nombre de vérifications de consistance réalisées, le nombre de retour arrière, le temps). Dans la région des problèmes satisfaisables, tous les CSPs ont au moins une solution ; dans la région des problèmes insatisfaisables aucun problème n'a de solution ; dans la région centrale il existe les deux types de problèmes. Lorsque l'on augmente le niveau de restrictivité des contraintes, à partir du niveau zéro, le coût de la résolution croît. Cette tendance se maintient jusqu'à ce que l'on trouve la première instance d'un problème insatisfaisable. À partir de ce niveau de restrictivité on se trouve dans la zone des problèmes les plus difficiles à résoudre, où il existe un sommet représentant le coût maximal. Si on continue à augmenter le niveau de restrictivité, la résolution des contraintes devient de moins en moins coûteuse, jusqu'à ce qu'on trouve la première instance d'un problème insatisfaisable. À partir de ce point-là, on quitte la zone des problèmes les plus difficiles et la résolution devient une tâche relativement simple. La conclusion générale est très simple : trouver une solution d'un problème qui en a beaucoup est très facile, démontrer qu'un problème n'a pas de solution quand le niveau de restrictivité est très élevé est également une tâche simple. La tâche la plus difficile est donc de résoudre des problèmes qui se trouvent dans la zone où il y a plus ou moins 50% de probabilité que le problème admette des solutions. Comme la résolution des CSOPs en utilisant les techniques de résolution de contraintes consiste à résoudre une séquence de CSPs où on cherche une seule solution, nous pouvons espérer un comportement similaire à celui montré dans la figure 6.1. Par l'ajout de contraintes, on change le niveau de restrictivité de l'ensemble initial jusqu'à arriver à une situation où le problème a une seule solution : la solution optimale (ou peut-être plusieurs également optimales).

Afin de gérer des situations où un des solveurs cherche une solution dans une zone de problèmes plus difficiles, nous pourrions lancer les solveurs et dès qu'une nouvelle solution est trouvée, on la compare avec la solution courante sans arrêter l'autre solveur. Si la nouvelle solution est meilleure que la solution courante, alors on met à jour la solution courante. Dans le cas contraire,

la nouvelle solution n'est pas prise en compte. Dans tous les cas, le solveur est relancé à partir de la solution courante (modifiée ou non). Suivant cette approche, les solveurs tournent d'une façon asynchrone et doivent être contrôlés. Puisque l'on ne veut pas arrêter un solveur quand l'autre retourne un résultat, nous sommes obligés d'implanter les solveurs par des processus externes.

Pour implanter cette approche, nous définissons la structure de données suivante.

```
global

CSOPbis@          : (set:tuple[csop, list[Pid]])          csopbis;

end
```

Dans la première composante, nous stockons la solution courante du CSOP. La deuxième composante contient une liste avec l'identification des processus externes utilisés.

La règle `CreateSolver1NoBlock` ci-dessous crée un processus externe, appelé `solver1CSOP`, qui, quand il est appelé, correspond à la stratégie `strat1`.

```
[CreateSolver1NoBlock]

CSOPbis[P1,lPid]
=>
CSOPbis[P1,pid.lPid]
where   pid := ()create_noblock("solver1CSOP")
if      not iserr(pid)
end
```

L'opérateur `create_noblock` spécifie que nous créons le processus comme étant non-bloquant. L'identification du processus `pid`, qui est retournée comme résultat de l'application de l'opérateur `create_noblock`, est ajoutée à la liste de processus externes `lPid`.

De la même manière, nous définissons la règle `CreateSolver2NoBlock` qui crée un processus externe appelé `solver2CSOP` et dont son identification `pid` est ajoutée à la liste `lPid`.

Une fois les solveurs lancés, ils restent en attente. Pour les faire travailler, nous définissons la règle `SendToSolver` qui prend l'identification d'un solveur (le premier élément de la liste de processus externes) et lui envoie le CSOP courant.

```
[SendToSolver]
CSOPbis[P1,pid.lPid]
=>
CSOPbis[P1,pid.lPid]
where   P2      := ()write(pid,P1)
where   string1 := ()write(pid,"\nend\n")
if      not iserror(P2) and not iserror(string1)
end
```

Dans tous les schémas précédents, à chaque fois qu'une nouvelle solution est trouvée, on sait qu'elle a soit une borne supérieure plus petite que la solution courante, soit une borne inférieure plus grande que la solution courante, car elle a été calculée en considérant la solution courante comme entrée. Maintenant, on est dans une situation différente, il n'y a plus cette information disponible et donc il nous faut la vérifier pour décider la mise à jour ou non de la solution courante. À ce fin, on définit la règle `GetOutputOfSolver`.

```
[GetOutputOfSolver]
CSOPbis[CSOP[P,x,lb,ub],pid.lPid]
=>
CSOPbis[R,pid.lPid]
if      lb != ub
where   Q :=()read(pid)
if      errno(P1) != No_term
choose
try     if GetLowerBound(Q) > lb or GetUpperBound(Q) < ub
        where R := ()Q
try     where R := ()CSOP[P,x,lb,ub]
end
end
```

Lorsqu'une nouvelle solution est trouvée, la règle `GetOutputOfSolver` laisse le solveur en tête de la liste de processus externes. Afin de pouvoir récupérer les résultats de l'autre solveur il nous faut donc un moyen pour gérer cette liste. Pour bien faire la différence entre les règles de calcul et ce type de procédure de gestion des priorités, nous avons envisagé la règle `ChangePriority` présentée ci-dessous.

```
[ChangePriority]
CSOPbis[CSOP[P,x,lb,ub],pid.lPid]
=>
CSOPbis[CSOP[P,x,lb,ub],append(lPid,pid.nil)]
if      lb != ub
end
```

Dans cet exemple, la règle `ChangePriority` envoie à la fin de la liste de processus externes l'identification du processus en tête. Cette approche nous donne la possibilité de changer très facilement la priorité des processus sans avoir à modifier les règles concernant le vrai calcul. Cette règle sera appliquée chaque fois que l'on lit un résultat d'un solveur ainsi que chaque fois que l'on essaie de lire un résultat alors que le premier solveur dans la liste n'a pas encore fini de le calculer.

Finalement, nous arrêtons un processus en utilisant la règle suivante.

```
[CloseSolver]
CSOPbis[P1,pid.lPid]
=>
CSOPbis[P1,lPid]
where pid1 := ()close(pid)
end
```

La stratégie `CooperationV` implante cette dernière approche.

```
stratop
global
      CooperationV : < csop -> csop >      bs;
end

[] CooperationV =>
SetUpperAndLowerBounds ;
CreateSolver1NoBlock ;
CreateSolver2NoBlock ;
repeat* (SendToSolver) ;
repeat* (first one (GetOutputOfSolver ; SendToSolver ; ChangePriority
                    ,
                    ChangePriority)
        ) ;
repeat* (CloseSolver)
end
```

Il est important de remarquer que les solveurs `strat1` et `strat2` retournent un seul résultat car ils utilisent un opérateur **first one**. Pour cette raison, chaque fois que l'on lit un résultat, on consomme tous les résultats (le seul) et donc le processus reste en attente et nous pouvons l'envoyer une nouvelle requête. Il est clair que si on définit un processus externe qui peut retourner plusieurs résultats il faudra gérer cette situation de manière à ne pas envoyer des requêtes au processus tant qu'il n'a pas fini de retourner tous les résultats.

Les traces possibles de ce schéma de coopération sont similaires à celles des solveurs concurrents, par exemple: $S_2; S_1; S_1; S_2; \ldots$ Néanmoins, ce dernier schéma de coopération va au delà du schéma concurrent car les solveurs travaillent sur des données d'entrée différentes et surtout parce qu'une solution venant d'un solveur n'élimine pas nécessairement une solution venant d'un autre solveur. Ce schéma de coopération n'est pas traité dans le langage BALI de Monfroy.

Évidemment, un choix *a priori* entre tous les schémas de coopération présentés dans cette section est impossible dans le cas général. Leur performance, et peut-être la performance d'autres schémas de coopération, doit être analysée pour chaque application particulière. L'analyse de performance des différents schémas n'est pas le sujet de cette section. Notre but a été de montrer comment, en utilisant les nouvelles facilités d'ELAN, nous pouvions prototyper différents types de coopération qui sont normalement utilisés par la communauté CLP. Les schémas de coopération des solveurs séquentiels et des solveurs concurrents ont été simulés d'une façon très abstraite et très flexible en utilisant les opérateurs de stratégies concurrents. Pour avoir plus de contrôle, comment le requiert le dernier schéma de coopération, nous avons dû utiliser les nouvelles primitives de communication avec des processus externes.

Chapitre 7

Conclusions et perspectives

La contribution principale de ce travail est la formalisation d'un nombre important de techniques de résolution de CSPs sous une forme qui fait une distinction claire entre les transformations élémentaires réalisées par ces techniques et leur contrôle.

> L'intérêt d'une telle formalisation a été renforcé récemment par le travail de K.R. Apt, qui, lui aussi, modélise la résolution de contraintes par des règles d'inférence : le calcul est associé à une preuve constructive d'une formule, la requête [2]. Dans notre travail, nous avons exprimé les transformations par des règles de réécriture. Ces règles ont été définies le plus simplement possible afin d'isoler les transformations essentielles réalisées par les techniques de résolution de CSPs. Cette décomposition nous a permis de réaliser des preuves de correction et de complétude très facilement, car il nous a suffit d'étudier chaque transformation individuellement. Nous avons utilisé de plus un moyen pour exprimer le contrôle : la notion de stratégie. Désormais, nous pouvons dire
>
> Solveur de contraintes = Règles + Stratégies
>
> Néanmoins, il faut signaler que, comme dans toutes les implantations de solveurs existantes, il y a parfois des différences entre la formalisation et l'implantation. Par exemple, notre formalisation considère toujours un CSP comme une conjonction de contraintes mais lors de l'implantation, la séparation de cette conjonction en des sous-ensembles de contraintes d'appartenance, d'égalité, etc., nous permet une implantation plus efficace.

Au cours de notre recherche, nous avons validé dans le cas particulier des CSPs, l'idée qu'un solveur de contraintes peut être vu comme une stratégie qui spécifie l'ordre d'application d'un ensemble de règles de réécriture, proposée originalement par Kirchner, Kirchner et Vittek [62, 63]. Cette validation a été faite du point de vue théorique, à travers la formalisation des techniques de résolution de CSPs sous la forme de systèmes de calcul, et aussi d'un point de vue pratique, avec l'implantation de COLETTE dans le langage ELAN.

> Les systèmes de réécriture se trouvent à la frontière entre théorie et pratique. L'idée d'exprimer des calculs sous la forme de règles de transformation (ou de transition) est largement connue en informatique. L'étude de la réécriture a apporté de bonnes connaissances quant à voir un processus d'inférence comme la

dérivation d'une forme normale par rapport à un ensemble de axiomes. Mais les règles toutes seules ne suffisent pas. Avoir un ensemble de règles correctes, complétes, terminantes et confluentes sont des propriétés très souhaitables, mais en terme pratique on s'intéresse surtout à avoir un moyen de contrôler ces dérivations. De notre point de vue, la réponse théorique *il existe une solution* n'est ni suffisante ni satisfaisante. Lors de la résolution de problèmes pratiques, la question vraiment importante à se poser est *comment trouve-t-on une solution*? Les caractéristiques de complétude et de correction nous semblent fondamentales, mais un langage de stratégies puissant nous permet, non seulement de spécifier l'ordre d'application des transformations, mais aussi de résoudre des problèmes de terminaison. Notre avis est cohérent avec celle de Van den Brand, Klint et Verhoef: la personne qui spécifie un système de réécriture a peu ou nul besoin de prouver des propriétés telles que la confluence et la terminaison car le modèle mental la guide dans la réalisation d'une exécution confluente et terminante du système [108]. Notre opinion est que les systèmes de calcul rendent vraiment praticable l'utilisation de la réécriture au travers de la notion fondamentale de stratégies.

D'ailleurs, nous avons vu comment, très facilement, on peut maintenant justifier l'existence d'une solution ou son absence: il nous suffit de regarder le terme de preuve pour reconstruire exactement une preuve du calcul réalisé. Ce problème, mentionné d'une part par Puget [99] et d'autre part par Montanari et Rossi [90], comme étant une des directions de recherche dans le domaine de la résolution de contraintes, est évidemment difficile à traiter sous l'approche impérative, alors que l'on obtient une réponse très satisfaisante lorsqu'on considère la résolution de contraintes comme un processus de déduction.

Le langage utilisé nous a permis de prototyper d'une manière flexible des techniques de résolution de contraintes. Mais cette affirmation pose déjà quelques questions: qu'est-ce qu'un prototypage flexible? Pour l'instant, seule l'expérience de la programmation permet d'appréhender le vrai sens de la flexibilité, mais il serait intéressant de définir des critères de flexibilité. Dans cette direction, une des questions intéressantes à se poser est *peut-t-on décrire tous les termes de preuve avec le langage considéré*? Concernant le même sujet du langage de stratégies, il nous semble intéressant de disposer d'outils formels pour prouver, par exemple, l'équivalence de deux stratégies (égalité des ensembles de termes de preuve des deux stratégies), l'implication d'une stratégie par une autre (inclusion de l'ensemble de termes de preuve d'une stratégie dans l'ensemble des termes de preuve d'une autre stratégie), etc. Par ailleurs, il nous semble nécessaire de disposer d'outils pour mesurer la complexité de l'application d'un ensemble de règles de réécriture afin de pouvoir analyser la complexité d'une stratégie.

Un des objectifs de notre travail a été d'étudier la possibilité d'utiliser les techniques de réécriture et les systèmes de calcul pour développer des applications en dehors de leur domaine. Nous pensons qu'avec le développement de COLETTE, nous avons contribué à répondre positivement à cette question et nous avons aussi contribué à la définition de nouveaux problèmes, qu'il faudra attaquer dans l'avenir pour faire de la réécriture et des systèmes de calcul des outils faciles à appliquer et utilisables efficacement dans la résolution de problèmes réels.

L'utilisation du langage ELAN comme outil de développement nous a posé quelques difficultés. D'une part, la façon de penser la programmation n'est pas la même

qu'avec des langages impératifs, logiques ou fonctionnels. D'autre part, au début de ce travail, ELAN était le seul outil disponible basé sur le paradigme de réécriture et stratégies. Évidemment, s'agissant d'un langage en constante évolution, nous avons dû surmonter des problèmes que normalement on ne rencontre pas lorsqu'on utilise un produit fini « sur étagère ». Néanmoins, il est particulièrement enrichissant de participer au développement d'un produit nouveau qui émerge dans un milieu en plein essor. Le développement existant aujourd'hui autour des systèmes MAUDE [31], ASF+SDF [38] et CafeObj [46] montre bien que le paradigme de la réécriture est un des sujets actuels de recherche en informatique fondamentale parmi les plus chauds. Malgré ces difficultés, il faut reconnaître clairement que la formalisation des techniques présentées dans ce travail aurait été impossible sans un outil pour tester le vrai comportement des systèmes proposés. Le processus de formalisation a toujours été enrichi par le prototypage et les résultats présentés ici sont le fruit d'une itération constante de formalisation et de mise en œuvre.

Concernant les aspects spécifiques relatifs à la simulation des techniques de résolution de CSPs, notre formalisation a été basée principalement sur l'utilisation du langage de stratégies d'ELAN, car il n'existe pas de formalisme standard indépendant du langage de programmation. Le langage de stratégies d'ELAN, basé sur un mécanisme de retour arrière, permet le prototypage simple et direct des techniques du style *Backtracking*. Cette manière d'aborder le non-déterminisme facilite considérablement la spécification des techniques de résolution pour trouver une ou toutes les solutions d'un CSP, des techniques pour le traitement de la disjonction basées sur des points de choix et des techniques d'optimisation basées sur la résolution incrémentielle de CSPs. Néanmoins, nous n'avons pas pu simuler des techniques de recherche qui considèrent des sauts dans l'arbre de recherche, comme par exemple, le *Intelligent Backtracking* [15] ou le *Conflict-Directed Backjumping* [98]. Vu l'intérêt de ces techniques de recherche il nous semble de grand intérêt d'étendre le langage pour permettre leur description de façon naturelle.

L'implantation actuelle du langage ELAN est fortement orientée vers le prototypage des systèmes de déduction complètement automatique : on applique un système de calcul sur une requête est on attend jusqu'à ce que le système retourne une solution. Considérant la très haute complexité inhérente à la résolution des CSPs et le fait que cette résolution est une tâche fortement heuristique, il nous semble intéressant de pouvoir disposer des facilités pour une interaction plus forte entre l'utilisateur et le système pendant le processus de calcul. On pourrait envisager, par exemple, un environnement où l'utilisateur pourrait arrêter le processus de calcul, voir son état et à partir de cela recommencer la recherche dans des points différents de l'arbre de recherche. Si on considère en plus une interface graphique, comme dans le système OZ [103], on pourrait obtenir un environnement pour le prototypage de systèmes de déduction semi-automatique qui, sans aucun doute, rend cette approche plus appropriée pour le traitement de problèmes pratiques de très haute complexité.

Le manque d'une méthodologie de programmation est évident et il nous semble que c'est un problème à résoudre si on veut appliquer ce paradigme pour la résolution de problèmes de taille réelle. Ainsi, comme il existe des méthodologies pour guider l'utilisation des langages impératifs ou basés à objets, il faut des critères généraux de conception dès qu'il s'agit d'applications de taille moyenne. Durant

la formalisation des techniques de résolution de CSPs et leur implantation, nous avons toujours essayé de faire une différence, la plus précise et stricte possible, entre la partie transformation et la partie contrôle. Mais il nous semble que la frontière entre ces deux aspects peut être ambiguë. On se pose souvent la question de comment répartir règles et stratégies et les réponses que nous avons donné expriment sans aucun doute notre façon particulière de penser. Pour bien réussir à transmettre cette approche de programmation basée sur des règles et des stratégies, il nous faut de critères plus clairs pour décider ce qui va être réalisé par des règles et ce qui va être réalisé par des stratégies.
Une des approches générales utilisée en informatique pour traiter la complexité, consiste à *diviser pour régner*. Lorsque l'on a un problème difficile on essaie de le décomposer en plusieurs sous-problèmes plus simples de façon telle de qu'on puisse résoudre plus facilement chaque sous-problème et à partir de leurs solutions construire celle du problème initial. En théorie de la réécriture, le problème de la décomposition de systèmes de réécriture reste un sujet important. En pratique, cela permet de combiner des outils relatifs aux différents sous-systèmes. Les problèmes d'héritage des propriétés des sous-systèmes demeurent un sujet de recherche.

Quant au problème spécifique de la résolution de contraintes, nous avons contribué par un nouvel algorithme pour le traitement des contraintes disjonctives.

Cet algorithme est basé sur l'idée générale que la manipulation de contraintes doit considérer d'une manière plus importante la structure mathématique du problème. Voir la résolution de contraintes comme une simple itération de vérification de consistance, quelque soit son niveau, et d'énumération de valeurs, nous semble trop naïf. Pour l'instant, on se base uniquement sur l'information des valeurs prises par les variables, le nombre d'occurrences d'une variable dans l'ensemble de contraintes, etc. Il est vrai qu'il existe beaucoup de techniques qui considèrent la structure du problème, mais ce sont des problèmes particuliers, le problème d'ordonnancement étant peut-être le meilleur exemple [17]. L'intégration de techniques de RO et de résolution de contraintes a été déjà reconnue comme un élément important dans cette direction. Pour cette raison l'idée d'intégrer des solveurs spécifiques pour certains types de problèmes ouvre d'après nous des perspectives de recherche.

Quant à la coopération de solveurs, nous avons contribué sous deux aspects.

Dans le cadre des systèmes de calcul, nous avons montré avec des exemples pratiques le besoin de disposer d'outils permettant l'intégration de plusieurs systèmes de calcul d'une manière abstraite et flexible. Grâce à ce besoin et au travail de P. Borovanský, nous avons désormais à disposition des opérateurs de stratégies concurrents dans le langage ELAN [12, 11].
Du point de vue de la communauté contraintes, nous avons contribué par la modélisation et la mise en œuvre, sous la forme de stratégies, des schémas de coopération de solveurs utilisés couramment [85]. Néanmoins, il reste beaucoup de travail à faire. Quelques schémas de coopération ont été modélisés à un très haut niveau d'abstraction en utilisant les opérateurs de stratégies concurrents, tandis que d'autres ont eu besoin des primitives de bas niveau.

D'une part, il reste encore à étudier la différence entre les opérateurs de bas niveau et ceux de haut niveau. L'implantation de la stratégie concurrente **dc** devrait aussi être étudiée dans le futur. D'autre part, il nous semble intéressant de mieux étudier la collaboration de solveurs qui échangent des informations pendant leurs exécutions. Les facilités existantes pour la collaboration de solveurs nous donnent des critères préétablis pour la sélection des solutions : la première solution obtenue par l'un des solveurs, une solution de chaque solveur, etc.. Cependant, afin de pouvoir prendre des décisions avant d'avoir toutes les solutions, il est important de disposer d'information concernant l'exécution de chaque solveur : nombre de règles appliquées, nombre de points de choix créés, taille du terme. Avec cette information on pourrait prendre de décisions lorsqu'aucun des solveurs n'est assez efficace pour résoudre un problème.

Finalement, à partir de l'expérience acquise au cours de cette thèse et comme fruit d'un travail réalisé avec Eric Monfroy, nous sommes en train de développer un langage de stratégies qui pourrait être considéré comme une extension au langage de stratégies d'ELAN.

Le but principal de ce travail est de proposer un cadre général pour la définition de solveurs élémentaires dans un formalisme qui permet de spécifier le traitement sémantique des contraintes et d'exprimer également des considérations syntaxiques souvent cachées dans les implantations des solveurs existants. Notre intérêt est de définir ce cadre de manière à ce qu'il s'étende naturellement à la spécification de la collaboration de solveurs. Ainsi comme un solveur de contraintes peut être vu comme une stratégie qui spécifie l'ordre d'application d'un ensemble de règles de transformation, nous voyons la collaboration de solveurs comme étant une stratégie qui spécifie l'ordre d'application de solveurs élémentaires ou composants.
Pour cela, nous introduisons d'abord les notions de filtres, de séparateurs et de tri de contraintes, qui principalement expriment des transformations syntaxiques sur les contraintes.

- Les filtres sont utilisés pour choisir des parties spécifiques d'une contrainte. Cette tâche est essentielle quand on veut appliquer un solveur sur un sous-ensemble d'un ensemble de contraintes (par exemple, parmi un ensemble de contraintes non-linéaires, on peut vouloir isoler les contraintes d'égalité sur les rationnels afin d'appliquer un solveur, comme par exemple, la méthode du simplexe).
- Les séparateurs sont définis principalement pour manipuler les éléments d'une conjonction ou d'une disjonction de contraintes comme des éléments d'une liste. En fait, l'idée derrière cette notion est de permettre la réalisation de calculs en parallèle.
- Les tris sont définis pour classer les éléments d'une liste de contraintes par rapport à un certain critère (par exemple, on peut être intéressé à trier les contraintes par rapport à leur cardinalité).

Basés sur ces notions, nous proposons ensuite trois types d'opérateurs de stratégies

- application d'un solveur sur une composante spécifique d'une contrainte : un opérateur pour choisir une composante par rapport à un critère donné et un autre opérateur pour choisir la composante de manière aléatoire ;

- application de plusieurs solveurs sur une contrainte : un opérateur qui choisit seulement un résultat par rapport à un critère donné et un autre opérateur qui compose le résultat final à partir des résultats de l'application de chaque solveur ;
- application d'un solveur sur chaque composante d'une conjonction de contraintes ou d'une disjonction de contraintes : un opérateur pour obtenir le résultat par conjonction des résultats obtenus en appliquant chaque solveur et un autre opérateur pour obtenir le résultat final par disjonction.

Une version préliminaire de ce langage est présentée dans [27] et [26]. Ce langage nous permet de simuler la résolution de contraintes sur des domaines finis, la résolution de contraintes non-linéaires sur des domaines continus et le calcul des bases de Gröbner. Nous travaillons actuellement à la mise au point du langage et le prototypage de la collaboration de solveurs.

Annexe A

Jeux d'essais

Tous les problèmes présentés dans la suite ont été décrits plus en détail dans la section 1.1. Pour les résoudre, nous avons utilisé la version 3.3 du compilateur d'ELAN sur une machine *DEC/Alpha* de type 500/333 avec une mémoire vive de 128 méga-octets sous $OSF1$. Le détail de tous les résultats est disponible dans `http://www.loria.fr/~castro/COLETTE/index.html`

A.1 Résolution de CSPs élémentaires

Nous présentons des statistiques qui montrent la performance de notre solveur pour la résolution de quatre problèmes crypto-arithmétiques, du problème des N reines et du problème de coloriage de graphes.

A.1.1 Puzzles crypto-arithmétiques

On présente dans la suite, des résultats expérimentaux pour la résolution de quatre puzzles crypto-arithmétiques. Nous appliquons douze stratégies pour obtenir la première solution et toutes les solutions. La caractéristique générale de ces problèmes consiste en la présence de conjonctions de contraintes élémentaires d'arité quelconque. De plus, tous les problèmes ont une seule solution.

Le premier puzzle consiste en la résolution de l'équation suivante

$$CROSS + ROADS = DANGER$$

où toutes les lettres doivent prendre des valeurs dans $[0, \ldots, 9]$ et doivent toutes être différentes. Afin de modéliser ce problème, nous exprimons cette équation de la manière suivante

$$\begin{array}{cccccccc} & R_5 & R_4 & R_3 & R_2 & R_1 & & \\ & & C & R & O & S & S \\ + & & R & O & A & D & S \\ \hline & D & A & N & G & E & R \end{array}$$

Notre modélisation consiste ainsi en un ensemble de 7 contraintes d'égalité, 5 contraintes d'inégalité et 38 contraintes de diségalité qui portent sur 14 variables.

La seule solution à ce problème est la suivante : $A = 5$, $C = 9$, $D = 1$, $E = 4$, $G = 7$, $N = 8$, $O = 2$, $R = 6$, $S = 3$, $R_1 = 0$, $R_2 = 0$, $R_3 = 0$, $R_4 = 0$ et $R_5 = 1$.

Dans la table A.1, nous présentons le comportement de la résolution de ce problème pour douze stratégies, exprimé par le temps (en secondes) et le nombre de réécritures.

Stratégie	Première solution		Toutes les solutions	
	Temps [sec.]	Nombre de réécritures	Temps [sec.]	Nombre de réécritures
FCFirstToLast	0,48	68.900	1,13	254.104
FCLastToFirst	0,94	213.908	1,04	254.326
FCMaximumDegreeInECFirstToLast	1,58	304.473	3,10	698.492
FCMaximumDegreeInECLastToFirst	1,87	425.355	3,16	700.026
FCMinimumDomainFirstToLast	0,31	38.015	0,87	179.245
FCMinimumDomainLastToFirst	0,86	170.122	0,84	179.256
FLAFirstToLast	0,67	108.053	1,85	385.318
FLALastToFirst	1,72	337.332	1,98	385.280
FLAMaximumDegreeInECFirstToLast	1,32	259.936	2,36	495.479
FLAMaximumDegreeInECLastToFirst	1,46	292.447	2,48	495.671
FLAMinimumDomainFirstToLast	0,47	69.976	1,73	375.016
FLAMinimumDomainLastToFirst	1,69	355.651	1,77	375.016

TAB. A.1 – *Résolution du puzzle* $CROSS + ROADS = DANGER$

À partir de cette table, nous pouvons tirer les conclusions suivantes

- l'heuristique de *Full Lookahead* montre de meilleurs résultats que l'heuristique de *Forward Checking* uniquement lorsque l'on considère le critère de sélection de variables *Maximum Degree* ;
- le critère de sélection des variables d'énumération qui choisit celle qui a le plus petit domaine produit clairement les meilleurs résultats ;
- lors de la recherche d'une seule solution, l'ordre d'énumération *first to last* est clairement plus performant que l'ordre d'énumération *last to first* quand ils sont utilisés avec une même stratégie. Au cours de la recherche de toutes les solutions, l'ordre d'énumération n'est absolument pas significatif.

Le deuxième puzzle consiste en la résolution de l'équation suivante

$$DONALD + GERALD = ROBERT$$

qui peut être exprimée de la façon suivante

$$\begin{array}{cccccccc} & R_5 & R_4 & R_3 & R_2 & R_1 & \\ & D & O & N & A & L & D \\ + & G & E & R & A & L & D \\ \hline & R & O & B & E & R & T \end{array}$$

Notre modélisation consiste en un ensemble de 6 contraintes d'égalité, 5 contraintes d'inégalité et 48 contraintes de diségalité portant sur 15 variables.

Ce problème a une seule solution : $A = 4$, $B = 3$, $D = 5$, $E = 9$, $G = 1$, $L = 8$, $N = 6$, $O = 2$, $R = 7$, $T = 0$, $R_1 = 1$, $R_2 = 1$, $R_3 = 0$, $R_4 = 1$ et $R_5 = 1$.

Nous décrivons dans la table A.2 le comportement des douze stratégies déjà mis en œuvre pour le premier puzzle.

Stratégie	Première solution		Toutes les solutions	
	Temps [sec.]	Nombre de réécritures	Temps [sec.]	Nombre de réécritures
FCFirstToLast	2,23	476.144	2,45	502.547
FCLastToFirst	0,46	62.006	2,31	502.626
FCMaximumDegreeInECFirstToLast	4,27	824.542	6,58	1.330.949
FCMaximumDegreeInECLastToFirst	2,80	545.046	7,03	1.329.744
FCMinimumDomainFirstToLast	0,65	94.702	0,60	96.911
FCMinimumDomainLastToFirst	0,32	37.702	0,58	96.911
FLAFirstToLast	5,83	1.128.251	6,71	1.193.608
FLALastToFirst	1,01	150.996	6,19	1.193.535
FLAMaximumDegreeInECFirstToLast	3,91	711.491	6,46	1.144.817
FLAMaximumDegreeInECLastToFirst	3,19	517.704	6,31	1.144.657
FLAMinimumDomainFirstToLast	1,44	221.991	1,34	221.999
FLAMinimumDomainLastToFirst	0,58	75.597	1,41	221.999

TAB. A.2 – *Résolution du puzzle* $DONALD + GERALD = ROBERT$

Nous pouvons tirer les mêmes conclusions que dans l'exemple précédent, sauf qu'ici, lors de la recherche d'une seule solution, l'ordre d'énumération *last to first* est clairement plus performant que l'ordre d'énumération *first to last* .

Le troisième puzzle consiste cette fois en la résolution de l'équation suivante

$$LIONNE + TIGRE = TIGRON$$

qu'on peut modéliser ainsi

$$\begin{array}{cccccccc} & R_5 & R_4 & R_3 & R_2 & R_1 & & \\ & L & I & O & N & N & E & \\ + & & T & I & G & R & E & \\ \hline & T & I & G & R & O & N & \end{array}$$

Notre modélisation consiste en un ensemble de 6 contraintes d'égalité, 5 contraintes d'inégalité et 30 contraintes de diségalité qui portent sur 13 variables.

La seule solution à ce problème est la suivante: $E = 1$, $G = 3$, $I = 6$, $L = 8$, $N = 2$, $O = 7$, $R = 5$, $T = 9$, $R_1 = 0$, $R_2 = 0$, $R_3 = 0$, $R_4 = 1$ et $R_5 = 1$.

Stratégie	Première solution		Toutes les solutions	
	Temps [sec.]	Nombre de réécritures	Temps [sec.]	Nombre de réécritures
FCFirstToLast	0,75	152.189	0,75	172.677
FCLastToFirst	0,38	66.130	0,80	172.689
FCMaximumDegreeInECFirstToLast	0,46	87.981	0,70	141.345
FCMaximumDegreeInECLastToFirst	0,55	99.967	0,67	141.323
FCMinimumDomainFirstToLast	0,36	46.697	0,46	83.287
FCMinimumDomainLastToFirst	0,52	82.122	0,53	83.287
FLAFirstToLast	1,06	246.372	1,19	288.773
FLALastToFirst	0,48	98.265	1,20	288.752
FLAMaximumDegreeInECFirstToLast	0,45	82.372	0,77	169.058
FLAMaximumDegreeInECLastToFirst	0,67	143.482	0,82	168.960
FLAMinimumDomainFirstToLast	0,33	51.301	0,58	114.757
FLAMinimumDomainLastToFirst	0,65	113.836	0,64	114.757

TAB. A.3 – *Résolution du puzzle* $LIONNE + TIGRE = TIGRON$

À partir des résultats figurant dans la table A.3, on peut constater que

- l'heuristique de *Full Lookahead* montre de meilleurs résultats que l'heuristique de *Forward Checking* uniquement pour le critère de sélection de variables *Maximum Degree*;
- le critère de sélection des variables d'énumération qui choisit celle qui a le plus petit domaine produit les meilleurs résultats quand il est utilisé avec un ordre d'énumération *first to last*;
- lors de la recherche d'une seule solution, l'ordre d'énumération *first to last* est clairement plus performant que l'ordre d'énumération *last to first* quand ils sont utilisés avec les critères de sélection de variables *Maximum Degree* et *Minimum Domain*. Au cours de la recherche de toutes les solutions, l'ordre d'énumération n'est absolument pas significatif.

Le quatrième puzzle consiste en la résolution de l'équation suivante

$$SEND + MORE = MONEY$$

qu'on peut écrire également

$$\begin{array}{cccccc} & R_4 & R_3 & R_2 & R_1 & \\ & & S & E & N & D \\ + & & M & O & R & E \\ \hline & M & O & N & E & Y \end{array}$$

Notre modélisation consiste en un ensemble de 6 contraintes d'égalité, 4 contraintes d'inégalité et 29 contraintes de diségalité.

La seule solution à ce problème est : $S = 9$, $E = 5$, $N = 6$, $D = 7$, $M = 1$, $O = 0$, $R = 8$, $Y = 2$, $R_1 = 1$, $R_2 = 1$, $R_3 = 0$ et $R_4 = 1$.

Stratégie	Première solution		Toutes les solutions	
	Temps [sec.]	Nombre de réécritures	Temps [sec.]	Nombre de réécritures
FCFirstToLast	0,25	34.244	0,48	106.934
FCLastToFirst	0,49	101.105	0,50	106.923
FCMaximumDegreeInECFirstToLast	0,29	39.561	0,31	50.106
FCMaximumDegreeInECLastToFirst	0,28	39.032	0,31	50.084
FCMinimumDomainFirstToLast	0,27	37.814	0,29	41.513
FCMinimumDomainLastToFirst	0,24	31.525	0,29	41.513
FLAFirstToLast	0,26	38.640	0,93	210.509
FLALastToFirst	0,83	205.115	0,91	210.509
FLAMaximumDegreeInECFirstToLast	0,30	45.376	0,34	58.644
FLAMaximumDegreeInECLastToFirst	0,27	42.799	0,33	58.644
FLAMinimumDomainFirstToLast	0,35	54.400	0,34	56.012
FLAMinimumDomainLastToFirst	0,28	34.848	0,35	56.012

TAB. A.4 – *Résolution du puzzle* $SEND + MORE = MONEY$

À partir des informations de la table A.4, on peut remarquer que

- l'heuristique de *Forward Checking* est plus performante que l'heuristique de *Full Lookahead* lors que l'on considère les mêmes critères de choix de variables et de choix de valeurs ;
- l'absence d'un critère de sélection des variables d'énumération produit les meilleurs résultats pour un ordre d'énumération *first to last* alors que pour un ordre d'énumération *last to first*, la non-utilisation d'un critère de sélection des variables d'énumération produit clairement les pires résultats ;
- lors de la recherche d'une seule solution, l'ordre d'énumération *first to last* est plus performant que l'ordre d'énumération *last to first* uniquement lorsqu'il est utilisé sans critère de sélection des variables. L'ordre d'énumération n'est absolument pas significatif.

En général, à partir des résultats obtenus lors de la résolution des quatre puzzles, nous pouvons observer que

- l'heuristique de *Forward Checking* est plus performante que l'heuristique de *Full Lookahead*, dans presque tous les puzzles, lorsque l'on considère les mêmes critères de choix de variables et de choix de valeurs ;
- le critère de sélection des variables d'énumération qui choisit celle qui a le plus petit domaine produit clairement de meilleurs résultats ;
- lors de la recherche d'une seule solution, il n'y a pas d'ordre d'énumération (*first to last* ou *last to first*) qui soit clairement plus performant et lors de la recherche de toutes les solutions, il n'est absolument pas significatif.

A.1.2 *N* Reines

Nous présentons des statistiques qui montrent la performance du solveur pour la résolution du problème des n reines. La caractéristique générale de ce problème consiste en la présence de conjonctions de contraintes binaires élémentaires. Nous considérons ici le problème de trouver une solution et le problème de trouver toutes les solutions.

La table A.5 présente le comportement, exprimé par le temps (T), en secondes, et le nombre de réécritures (#), de six stratégies pour la résolution de ces deux problèmes (recherche d'une solution et recherche de toutes les solutions) considérant entre 3 et 12 reines.

Le problème des 3 reines est insatisfaisable et le nombre de solutions pour les problèmes suivants est respectivement : 2, 10, 4, 40, 92, 352, 724, 2.680 et 14.200.

À partir de ces résultats, nous pouvons observer que

- lorsque le problème est insatisfaisable (le cas des 3 reines) les stratégies de recherche d'une solution et de toutes les solutions ont un comportement similaire car évidemment dans les deux cas il nous faut parcourir tout le espace de recherche ;
- pour des problèmes très simples (moins de 7 reines) la stratégie `BacktrackingFirstToLast` produits les meilleurs résultats, mais à partir du problème des 8 reines le gain d'une vérification de consistance locale limitée se manifeste car la stratégie `FCMinimumDomainFirstToLast` produit alors les meilleurs résultats.

A.1.3 Coloriage de graphes : satisfaisabilité

Nous présentons des statistiques qui montrent la performance du solveur pour la résolution de problèmes de coloriage de graphes. La caractéristique générale de ce problème consiste en la présence de conjonctions de contraintes binaires élémentaires.

Nous considérons ici le problème de déterminer la satisfaisabilité d'affecter une couleur, parmi un maximum de trois, à chaque nœud d'un graphe. Afin de prendre en compte différentes topologies des graphes considérant un même nombre de nœuds, nous créons de manière aléatoire une contrainte imposant que deux nœuds n'aient pas la même couleur. Nous avons considéré des probabilités variant entre 10% et 90% et pour chaque valeur de probabilité nous avons résolu dix instances du problème.

Les tables A.6 et A.7 présentent le comportement, exprimé par le temps (T), en secondes, et le nombre de réécritures (#) en moyenne, de quatre stratégies pour la résolution de ce problème considérant, respectivement, 10 et 20 nœuds.

En général, nous pouvons observer que

- la stratégie `FCMinimumDomainFirstToLastOne` produit dans les deux ensembles de problèmes considérés les meilleurs résultats ;
- aucune stratégie ne permet de vérifier la présence du phénomène de transition de phase car l'effort nécessaire pour résoudre un problème est d'autant plus grand que l'ensemble de contraintes est restrictif.

Nous croyons que l'absence du phénomène de transition de phase est dûe au fait que notre système effectue des calculs de pré-traitement du problème qui dépendent de la taille du problème. Vu le temps très petit de résolution de chaque problème, ce phénomène n'est pas assez important pour se manifester.

A.2 Résolution de CSOPs

Nous appliquons notre solveur à la résolution de problèmes de coloriage de graphes et d'ordonnancement d'activités.

# de reines	Stratégie											
	Backtracking-FirstToLast		BacktrackingMaximum-DegreeInECFirstToLast		FCMaximumDegree-InECFirstToLast		FCMinimumDomain-FirstToLast		FLAMaximumDegree-InECFirstToLast		FLAMinimumDomain-FirstToLast	
	1ère solution	Toutes les solutions	1ère solution	Toutes les solutions	1ère solution	Toutes les solutions	1ère solution	Toutes les solutions	1ère solution	Toutes les solutions	1ère solution	Toutes les solutions
3 T	0,11	0,11	0,12	0,12	0,11	0,12	0,11	0,11	0,13	0,12	0,12	0,12
#	4.146	4.146	5.475	5.475	5.981	5.981	5.923	5.923	6.889	6.889	6.849	6.849
4 T	0,13	0,16	0,14	0,20	0,16	0,17	0,15	0,16	0,15	0,20	0,16	0,19
#	8.398	16.397	12.379	26.538	11.747	19.914	11.125	18.773	17.365	31.417	17.229	31.246
5 T	0,15	0,42	0,18	0,60	0,15	0,34	0,15	0,29	0,19	0,48	0,20	0,50
#	8.138	70.896	15.916	126.626	12.347	70.840	12.012	62.831	21.393	122.138	21.101	121.462
6 T	0,51	1,70	1,06	2,98	0,34	0,90	0,30	0,73	0,57	1,83	0,58	1,74
#	68.263	327.773	166.119	576.484	54.786	231.289	49.159	194.429	128.214	500.248	123.957	482.857
7 T	0,38	11,07	1,09	19,08	0,37	3,79	0,33	2,36	0,68	8,33	0,57	7,47
#	26.204	1.550.690	126.578	2.950.991	33.583	940.477	32.413	634.904	90.172	2.028.805	89.113	1.902.863
8 T	4,55	70,26	4,41	126,91	1,13	16,37	0,99	9,03	2,69	41,69	2,71	35,00
#	454.660	7.778.480	475.745	15.816.800	174.557	3.943.527	170.957	2.370.678	470.516	8.538.796	486.931	7.724.604
9 T	3,45	436,56	17,07	989,38	1,06	76,49	0,94	43,32	2,16	220,81	2,12	182,28
#	220.918	40.988.870	1.307.064	94.247.962	97.473	18.951.549	98.117	10.142.299	228.412	41.256.738	225.618	34.821.655
10 T	12,03	3.197,44	40,68	8.807,20	2,58	423,71	1,76	190,94	11,13	1.221,40	8,04	916,58
#	689.496	222.378.411	2.762.325	581.533.289	335.464	93.208.307	198.459	41.791.023	1.501.485	207.843.171	1.022.747	156.584.443
11 T	10,89	21.048,98	245,39	69.998,55	3,66	2.887,36	2,53	932,29	18,76	8.083,83	12,19	4.964,33
#	437.166	1.290.883.740	8.791.206	3.871.809.745	406.513	512.954.438	221.927	194.209.014	2.092.264	1.078.758.937	1.281.759	758.715.228
12 T	68,31	166.368,88	134,91	?	4,06	16.565,29	3,19	4.950,20	16,30	48.093,74	13,46	29.025,57
#	2.706.965	3.693.528.864	5.479.603	?	221.905	3.061.737.472	127.307	971.663.752	1.080.173	2.057.913.063	874.642	3.974.893.696

TAB. A.5 – *Résolution du problème des N Reines*

Stratégie		Probabilité d'existence d'une contrainte entre deux nœuds								
		10%	20%	30%	40%	50%	60%	70%	80%	90%
FCMaximumDegreeInEC-	T	0,113	0,119	0,124	0,196	0,245	0,241	0,239	0,269	0,296
FirstToLastOne	#	4.075	5.033	6.699	25.257	38.585	34.799	31.143	34.953	35.298
FCMinimumDomain-	T	0,108	0,118	0,128	0,163	0,177	0,198	0,216	0,225	0,242
FirstToLastOne	#	3.595	5.712	8.061	17.484	21.578	22.014	26.310	26.890	28.653
FLAMaximumDegreeInEC-	T	0,112	0,122	0,168	0,189	0,220	0,269	0,324	0,351	0,398
FirstToLastOne	#	3.362	5.773	17.179	22.931	29.595	39.147	46.781	51.548	57.678
FLAMinimumDomain-	T	0,111	0,127	0,147	0,192	0,239	0,264	0,262	0,298	0,359
FirstToLastOne	#	3.322	6.205	11.357	25.432	36.931	39.790	36.297	42.440	50.587

TAB. A.6 – *Résolution du problème de coloriage de graphes : satisfaisabilité considérant 10 nœuds*

Stratégie		Probabilité d'existence d'une contrainte entre deux nœuds								
		10%	20%	30%	40%	50%	60%	70%	80%	90%
FCMaximumDegreeInEC-	T	0,184	0,861	2,102	1,538	2,433	2,427	2,944	3,377	3,858
FirstToLastOne	#	13.188	123.653	277.128	148.757	197.254	151.122	151.322	149.337	146.130
FCMinimumDomain-	T	0,176	0,566	0,657	0,696	0,872	1,228	1,518	2,049	2,583
FirstToLastOne	#	13.632	96.543	81.867	63.055	66.430	78.208	85.564	99.169	112.851
FLAMaximumDegreeInEC-	T	0,207	0,501	1,235	2,103	2,533	4,205	5,87	7,843	11,068
FirstToLastOne	#	17.258	60.229	135.219	175.924	181.297	254.300	289.391	348.450	411.868
FLAMinimumDomain-	T	0,200	0,495	1,053	1,529	2,159	2,766	4,111	6,023	8,338
FirstToLastOne	#	16.415	62.367	114.536	137.983	152.567	155.273	210.548	269.297	333.850

TAB. A.7 – *Résolution du problème de coloriage de graphes : satisfaisabilité considérant 20 nœuds*

A.2.1 Coloriage de graphes : minimisation

Nous considérons ici le problème de déterminer le nombre minimum de couleurs qui sont nécessaires pour affecter une couleur à chaque nœud d'un graphe en respectant des contraintes qui imposent que certaines paires de nœuds n'aient pas la même couleur. Afin de prendre en compte différentes topologies des graphes considérant un même nombre de nœuds, nous créons de manière aléatoire une contrainte imposant que deux nœuds n'aient pas la même couleur. Nous avons considéré des probabilités variant entre 10% et 90% et pour chaque valeur de probabilité nous avons résolu dix instances du problème. Nous considérons comme borne supérieure pour commencer la minimisation, le nombre de nœuds dans le problème afin de s'assurer que le problème est satisfaisable.

La table A.8 présente le comportement, exprimé par le temps (T), en secondes, et le nombre de réécritures (#) en moyenne, de quatre stratégies pour la résolution de ce problème considérant 10 nœuds.

Nous pouvons observer que

- la stratégie `MinSplittingFCMinimumDomainFirstToLast` produit, en général, les meilleurs résultats ;
- aucune stratégie ne permet de vérifier la présence du phénomène de transition de phase car l'effort nécessaire pour résoudre un problème est d'autant plus grand que l'ensemble de contraintes est restrictif.

Comme nous l'avons signalé dans le cas de la satisfaisabilité du problème de coloriage de graphes, nous pensons que l'absence du phénomène de transition de phase s'explique par le fait que notre système effectue des calculs de pré-traitement du problème qui dépendent de la taille du problème. Vu le temps négligeable de résolution de chaque problème, ce phénomène n'est pas assez important pour se manifester.

A.2.2 Ordonnancement d'activités

Nous considérons ici le problème de déterminer le temps minimum de fin d'un ensemble de n activités en respectant des contraintes de précédence entre certaines paires d'activités. Afin de prendre en compte des précédences différentes considérant un même nombre d'activités, nous créons de manière aléatoire une contrainte de précédence entre deux activités. Nous avons considéré des probabilités variant entre 10% et 90%. Nous créons de manière aléatoire la durée de chaque activité considérant comme intervalle de valeurs possibles $[1, \ldots, 100]$. Nous considérons comme borne supérieure, pour commencer la minimisation, le temps nécessaire pour finir toutes les n activités lorsqu'elles sont exécutées en série afin de s'assurer que le problème est satisfaisable. Cette borne est donc déterminée par $n \times 100$. Pour chaque valeur de probabilité, nous avons résolu dix instances du problème.

La table A.9 présente le comportement, exprimé par le temps (T), en secondes, et le nombre de réécritures (#) en moyenne, de huit stratégies pour la résolution de ce problème considérant 10 activités.

Nous pouvons observer que

- la stratégie `MinSplittingFCMaximumDegreeInECFirstToLast` produit, en général, les meilleurs résultats ;
- l'approche *splitting* produit, en général, de meilleurs résultats que l'approche *satisfaisabilité à insatisfaisabilité* ;

Stratégie		Probabilité d'existence d'une contrainte entre deux nœuds								
		10%	20%	30%	40%	50%	60%	70%	80%	90%
MinSatToUnFCMaximum-	T	0,163	0,222	0,308	0,537	1,013	1,393	9,078	25,123	65,883
DegreeInECFirstToLast	#	15.779	32.040	49.926	110.083	211.727	301.314	2.167.690	6.129.530	16.838.400
MinSatToUnFCMinimum-	T	0,161	0,205	0,281	0,581	0,816	2,236	2,053	16,506	83,039
DomainFirstToLast	#	16.768	28.634	47.442	119.004	183.069	531.027	488.947	4.415.820	20.762.400
MinSplittingFCMaximum-	T	0,180	0,261	0,358	0,574	0,701	1,273	3,865	27,596	55,502
DegreeInECFirstToLast	#	21.591	42.119	64.680	107.742	137.454	269.421	894.385	6.937.040	13.743.900
MinSplittingFCMinimum-	T	0,187	0,355	0,272	0,365	0,508	0,576	3,359	14,236	63,951
DomainFirstToLast	#	23.557	69.570	43.542	68.272	101.919	119.808	830.707	3.475.140	15.924.500

TAB. A.8 – *Résolution du problème de coloriage de graphes : minimisation considérant 10 nœuds*

Stratégie		Probabilité d'existence d'une contrainte de précédence entre deux activités								
		10%	20%	30%	40%	50%	60%	70%	80%	90%
MinSatToUnFCMaximum-	T	0,148	0,175	0,220	0,261	0,331	0,416	0,491	0,550	0,682
DegreeInECFirstToLast	#	12.614	18.668	26.630	36.549	49.650	63.052	75.180	86.120	108.565
MinSatToUnFCMaximum-	T	18,317	24,464	28,600	35,945	45,671	48,665	58,000	63,707	71,383
DegreeInECLastToFirst	#	4.396.730	5.991.165	6.692.177	8.016.745	9.485.323	9.560.392	11.219.685	11.615.707	12.395.334
MinSatToUnFCMinimum-	T	0,146	0,177	0,214	0,272	0,314	0,394	0,379	0,545	0,690
DomainFirstToLast	#	13.083	19.441	28.649	40.161	49.366	63.596	63.742	94.496	118.889
MinSatToUnFCMinimum-	T	16,819	20,469	29,676	31,806	34,573	38,059	40,386	50,440	52,806
DomainLastToFirst	#	4.641.711	5.454.427	7.046.732	7.809.478	8.358.846	8.741.738	8.970.014	10.561.336	10.825.612
MinSplittingFCMaximum-	T	0,145	0,181	0,206	0,237	0,282	0,380	0,442	0,563	0,758
DegreeInECFirstToLast	#	11.580	19.490	24.995	32.391	40.412	58.031	68.581	87.094	117.318
MinSplittingFCMaximum-	T	0,343	0,460	0,578	0,785	0,943	1,206	1,473	1,808	2,068
DegreeInECLastToFirst	#	66.788	88.375	117.661	157.755	191.837	233.271	282.548	321.024	363.826
MinSplittingFCMinimum-	T	0,146	0,175	0,213	0,246	0,318	0,378	0,451	0,532	0,671
DomainFirstToLast	#	12.498	19.329	27.003	33.860	48.673	58.224	74.127	87.331	112.458
MinSplittingFCMinimum-	T	0,314	0,443	0,555	0,694	0,961	0,936	1,255	1,573	1,891
DomainLastToFirst	#	60.520	90.992	117.646	147.888	210.115	200.730	260.191	311.872	368.158

TAB. A.9 – *Résolution du problème d'ordonnancement :* 10 *activités*

- le critère de sélection des variables n'a pas d'influence sur le comportement de chaque approche ;
- l'ordre d'énumération *first to last* produit toujours les meilleurs résultats ;
- l'ordre d'énumération *last to first* produit clairement des très mauvais résultats quand il est utilisé dans l'approche *satisfaisabilité à insatisfaisabilité* ;
- aucune stratégie ne permet de vérifier la présence du phénomène de transition de phase.

Comme nous l'avons déjà signalé dans le cas des problèmes de coloriage de graphes (le problème de satisfaisabilité et le problème de minimisation), nous pensons que l'absence du phénomène de transition de phase est dûe aux calculs de pré-traitement qui sont effectués par notre système et qui dépendent de la taille du problème.

A.3 Résolution de CSOPs avec des disjonctions

Nous présentons des statistiques qui montrent la performance du solveur pour la résolution du problème d'ordonnancement disjonctif.

A.3.1 Ordonnancement disjonctif

Nous considérons ici le problème de déterminer le temps minimum de fin d'un ensemble de n jobs sur n machines. Nous créons de manière aléatoire le temps nécessaire pour exécuter chaque tâche sur chaque machine considérant comme intervalle de valeurs possibles $[1, \ldots, 100]$. Nous considérons comme borne supérieure, pour commencer la minimisation, le temps nécessaire pour finir toutes les tâches lorsqu'elles sont exécutées en série afin de s'assurer que le problème est satisfaisable. Cette borne est donc déterminée par $n \times n \times 100$. Nous avons résolu trois instances de chaque problème.

La table A.10 présente le comportement, exprimé par le temps (T), en secondes, et le nombre de réécritures (#) en moyenne, de quatre stratégies pour la résolution des problèmes 3×3 et 5×5.

Stratégie	3x3		5x5	
	Temps [sec.]	Nombre de réécritures	Temps [sec.]	Nombre de réécritures
MinSatToUnFCChoicePoint-MinimumDomainFirstToLast	0,671	121.368	14.017,900	1.063.430.000
MinSatToUnFCChoicePoint-MinimumDomainLastToFirst	44,947	10.303.600	10.532,200	819.329.982
MinSplittingFCChoicePoint-MinimumDomainFirstToLast	0,939	183.499	4.246,110	329.642.000
MinSplittingFCChoicePoint-MinimumDomainLastToFirst	1,173	243.742	5.262,630	358.538.000

TAB. A.10 – *Résolution du problème d'ordonnancement disjonctif*

En général, nous pouvons observer que

- l'approche *splitting* produit de meilleurs résultats que l'approche *satisfaisabilité à insatisfaisabilité* ;

- l'ordre d'énumération *first to last* ou *last to first* a une grande influence sur le comportement de l'approche *satisfaisabilité à insatisfaisabilité.*

Il faut signaler que l'approche utilisée pour modéliser ce problème d'ordonnancement est très naïve. Par exemple, pour le problème 5×5, on utilise 25 contraintes élémentaires et 50 contraintes disjonctives qui portent sur 25 variables.

Nos performances sont inférieures de 50 à 60 fois par rapport aux performances présentées par Zhou dans [123], où des heuristiques spécifiques sont appliquées à la résolution de modèles très élaborés de puzzles et de problèmes d'ordonnancement. Tenant compte que nous utilisons une modélisation très naïve et des heuristiques générales, nous pensons que les performances sont assez bonnes.

Bibliographie

[1] A. Aho and J. Ullman. *Concepts fondamentaux de l'informatique.* DUNOD, Paris, France, 1993.

[2] K. R. Apt. A Proof Theoretic View of Constraint Programming. *Fundamenta Informaticae*, 34(3) :295–321, June 1998.

[3] H. Beringer and B. DeBacker. Combinatorial Problem Solving in Constraint Logic Programming with Cooperative Solvers. In C. Beierle and L. Plümer, editors, *Logic Programming : Formal Methods and Practical Applications*, Studies in Computer Science and Artificial Intelligence. North Holland, 1995.

[4] C. Bessière. A fast algorithm to establish arc-consistency in constraint networks. Technical Report TR-94-003, LIRMM Université de Montpellier II, January 1994.

[5] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65 :179–190, 1994.

[6] C. Bessière and J.-C. Régin. An arc-consistency algorithm optimal in the number of constraint checks. In *Proceedings of the Workshop on Constraint Processing, ECAI'94*, pages 9–16, Amsterdam, The Netherlands, 1994.

[7] J. J. Bitner and E. M. Reingold. Backtrack Programming Techniques. *Communications of the ACM*, 18(11) :651–656, 1975.

[8] D. G. Bobrow and B. Raphael. New Programming Languages for Artificial Intelligence Research. *Computing Surveys*, 6(3) :153–174, September 1974.

[9] A. Bockmayr and T. Kasper. A unifying framework for integer and finite domain constraint programming. Research Report MPI-I-97-2-008, Max Planck Institut für Informatik, Saarbrücken, Germany, August 1997.

[10] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartošek, J. Staudek, and J. Wiedermann, editors, *Proceedings of the SOFSEM'95 : Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 363–368. Springer-Verlag, 1995.

[11] P. Borovanský. *Le contrôle de la réécriture : étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, October 1998.

[12] P. Borovanský and C. Castro. Cooperation of Constraint Solvers : Using the New Process Control Facilities of ELAN. In C. Kirchner and H. Kirchner, editors, *Proceedings of The Second International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, pages 379–398, Pont-à-Mousson, France, September 1998. Electronic Notes in Theoretical Computer Science. Also available as Technical Report 98-R-305 of the Laboratoire lorrain de recherche en informatique et ses applications, LORIA.

[13] P. Borovanský, C. Kirchner, and H. Kirchner. A functional view of rewriting and strategies for a semantics of ELAN. In M. Sato and Y. Toyama, editors, *The Third Fuji International*

Symposium on Functional and Logic Programming, pages 143–167, Kyoto, Japan, April 1998. World Scientific.

[14] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. *ELAN Version 3.00 User Manual.* CRIN & INRIA Lorraine, Nancy, France, first edition, January 1998.

[15] M. Bruynooghe. Solving Combinatorial Search Problems by Intelligent Backtracking. *Information Processing Letters*, 12(1) :36–39, 1981.

[16] M. Bruynooghe and R. Venken. Backtracking. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992. Second Edition.

[17] Y. Caseau and F. Laburthe. Improved CLP Scheduling with Task Intervals. In P. van Hentenryck, editor, *Proceedings of The 11th International Conference on Logic Programming, ICLP'94.* The MIT press, 1994.

[18] Y. Caseau and F. Laburthe. *Introduction to the CLAIRE Programming Language.* Département Mathématiques et Informatique, Ecole Normale Supérieure, Paris, France, September 1996.

[19] C. Castro. Binary CSP Solving as an Inference Process. In *Proceedings of The Eigtht International Conference on Tools in Artificial Intelligence, ICTAI'96*, pages 462–463, Toulouse, France, November 1996. Also available as Technical Report 96-R-270 of the Centre de Recherche en Informatique de Nancy, CRIN.

[20] C. Castro. Solving Binary CSP Using Computational Systems. In J. Meseguer, editor, *Proceedings of The First International Workshop on Rewriting Logic and its Applications, RWLW'96*, volume 4, pages 245–264, Asilomar, Pacific Grove, CA, USA, September 1996. Electronic Notes in Theoretical Computer Science. Also available as Technical Report 96-R-190 of the Centre de Recherche en Informatique de Nancy, CRIN.

[21] C. Castro. Constraint Manipulation using Rewrite Rules and Strategies. In A. Drewery, G.-J. M. Kruijff, and R. Zuber, editors, *Proceedings of The Second ESSLLI Student Session, 9th European Summer School in Logic, Language and Information, ESSLLI'97*, pages 45–56, Aix-en-Provence, France, August 1997. Also available as Technical Report 97-R-279 of the Centre de Recherche en Informatique de Nancy, CRIN.

[22] C. Castro. Manipulation de contraintes par des systèmes de calcul. In *Journées du Pôle Contraintes et Programmation Logique du PRC/GDR Programmation*, pages 1–10, Rennes, France, November 1997.

[23] C. Castro. An Arc-Consistency Algorithm for CSPs Involving Disjunctive Constraints. Technical report, Centre de Recherche en Informatique de Nancy, CRIN, 1998. To appear.

[24] C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34(3) :263–293, June 1998. Also available as Technical Report 98-R-308 of the Laboratoire lorrain de recherche en informatique et ses applications, LORIA.

[25] C. Castro. COLETTE, Prototyping CSP Solvers Using a Rule-Based Language. In J. Calmet and J. Plaza, editors, *Proceedings of The Fourth International Conference on Artificial Intelligence and Symbolic Computation, Theory, Implementations and Applications, AISC'98*, volume 1476 of *Lecture Notes in Artificial Intelligence*, pages 107–119, Plattsburgh, NY, USA, September 1998. Also available as Technical Report 98-R-304 of the Laboratoire lorrain de recherche en informatique et ses applications, LORIA.

[26] C. Castro and E. Monfroy. A Strategy Language for Solving CSPs. In K. Apt, P. Codognet, and E. Monfroy, editors, *Proceedings of The Third Workshop of the Working Group*

on Constraints of the European Research Consortium for Informatics and Mathematics, ERCIM'98, Amsterdam, The Netherlands, September 1998. Also available as Technical Report 98-R-307 of the Laboratoire lorrain de recherche en informatique et ses applications, LORIA.

[27] C. Castro and E. Monfroy. A Strategy Language for Specifying Constraint Solvers and their Collaborations. In J. Plaza, editor, *Poster Session Proceedings of The Fourth International Conference on Artificial Intelligence and Symbolic Computation, Theory, Implementations and Applications, AISC'98*, pages 1–10, Plattsburgh, NY, USA, September 1998. Also available as Technical Report 98-R-306 of the Laboratoire lorrain de recherche en informatique et ses applications, LORIA.

[28] B. F. Caviness. On Canonical Forms and Simplification. *Journal of the ACM*, 17(2) :385–396, April 1970.

[29] P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the Really Hard Problems Are. In *Proceedings of The Twelfth International Joint Conference on Artificial Intelligence, IJCAI'91, Sidney, Australia*, pages 331–337, 1991.

[30] H. Cirstea and C. Kirchner. Theorem proving using computational systems : The case of the B Predicate prover. In *Proceedings of The International Conference on Constraints in Computational Logics, Dagstuhl, Germany*, September 1997.

[31] M. Clavel, F. Durán, S. Eker, P. Lincoln, and J. Meseguer. An Introduction to Maude (Beta Version). Technical report, SRI International, Menlo Park, USA, March 1998.

[32] J. Cohen. Non-Deterministic Algorithms. *Computing Surveys*, 11(2) :79–94, June 1979.

[33] H. Comon, M. Dincbas, J.-P. Jouannaud, and C. Kirchner. A Methodological View of Constraint Solving. *Constraints*, 1998. To appear.

[34] G. Dantzig. *Integer Programming and Extensions.* Princeton University Press, 1972.

[35] R. Debruyne. Les algorithmes d'arc-consistance dans les CSP dynamiques. *Revue d'Intelligence Artificielle*, 9(3) :239–267, 1995.

[36] R. Dechter. Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41 :273–312, 1990.

[37] N. Dershowitz and E. M. Reingold. *Calendrical Calculations.* Cambridge University Press, 1997.

[38] A. Deursen, J. Heering, and P. Klint. *Language Prototyping.* World Scientific, 1996. ISBN 981-02-2732-9.

[39] Y. Deville and P. V. Hentenryck. An efficient arc consistency algorithm for a class of csp problems. In *Proceedings of The Twelfth International Joint Conference on Artificial Intelligence, IJCAI'91, Sidney, Australia*, pages 325–330, 1991.

[40] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8 :75–93, January-March 1990. Special Issue : Logic Programming Applications.

[41] F. Fages. *Programmation Logique Par Contraintes.* Cours de l'Ecole Polytechnique. Ellipses, Paris, France, 1996.

[42] R. E. Fikes. REF-ARF : A System for Solving Problems Stated as Procedures. *Artificial Intelligence*, 1 :27–120, 1970.

[43] E. C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21(11) :958–966, November 1978.

[44] E. C. Freuder. Using Metalevel Constraint Knowledge to Reduce Constraint Checking. In

Proceedings of the Workshop on Constraint Processing, ECAI'94, Amsterdam, The Netherlands, pages 27–33, 1994.

[45] T. Frühwirth. Constraint Handling Rules. In A. Podelski, editor, *Constraint Programming : Basic and Trends. Selected Papers of the 22nd Spring School in Theoretical Computer Sciences*, volume 910 of *Lecture Notes in Computer Science*, pages 90–107. Springer-Verlag, Châtillon-sur-Seine, France, May 1994.

[46] K. Futatsugi and A. Nakagawa. An Overview of Cafe Project. In *Proceedings of First CafeOBJ Workshop*, Yokohama, Japan, August 1996.

[47] F. Gadducci. *On the Algebraic Approach to Concurrent Term Rewriting.* PhD thesis, Università di Pisa, Italy, January 1996.

[48] J. H. Gallier. *Logic for Computer Sciences, Foundations of Automatic Theorem Proving.* Harper and Row, 1986.

[49] T. Genet and I. Gnaedig. Solving GPO ordering constraints with shared term data structure. Technical Report 95-R-363, CRIN & INRIA Lorraine, 1995.

[50] I. P. Gent and T. Walsh. Easy Problems are Sometimes Hard. Research Paper 642, Department of Artificial Intelligence, University of Edinburgh, June 1993.

[51] I. P. Gent and T. Walsh. The SAT Phase Transition. Research Paper 679, Department of Artificial Intelligence, University of Edinburgh, January 1994.

[52] S. A. Grant and B. M. Smith. The Phase Transition Behaviour of Maintaining Arc Consistency. Technical Report 95.25, School of Computer Studies, University of Leeds, August 1995.

[53] S. A. Grant and B. M. Smith. The Arc and Path Consistency Phase Transitions. Technical Report 96.09, School of Computer Studies, University of Leeds, March 1996.

[54] H. Greenberg. *Integer Programming.* Mathematics in science and engineering. Academic Press, New York/London, 1971.

[55] C.-C. Han and C.-H. Lee. Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence*, 36 :125–130, 1988.

[56] R. M. Haralick and G. L. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14 :263–313, 1980.

[57] F. S. Hillier and G. J. Liebermann. *Introducción a la Investigación de Operaciones.* McGraw-Hill, 1991.

[58] ILOG S.A. *ILOG Solver 4.0, User's Manual*, May 1997.

[59] J.-P. Jouannaud and C. Kirchner. Solving Equations in Abstract Algebras : A Rule-Based Survey of Unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge, MA, USA, 1991.

[60] J. Jourdan and T. Sola. The Versatility of Handling Disjunctions as Constraints. In M. Bruynooghe and J. Penjam, editors, *Proceedings of The 5th International Symposium on Programming Language Implementation and Logic Programming, PLILP'93, Tallinn, Estonia*, number 714 in Lecture Notes in Computer Science, pages 60–74. Springer-Verlag, August 1993.

[61] C. Kirchner and H. Kirchner. Rewriting, Solving, Proving. Notes de cours GRECO, INRIA Lorraine and CRIN, Nancy, France, March 1995.

[62] C. Kirchner, H. Kirchner, and M. Vittek. Implementing Computational Systems with Constraints. In P. Kanellakis, J.-L. Lassez, and V. Saraswat, editors, *Proceedings of The*

First Workshop on Principles and Practice of Constraint Programming, Providence, R.I., USA, pages 166–175, 1993.

[63] C. Kirchner, H. Kirchner, and M. Vittek. Designing Constraint Logic Programming Languages using Computational Systems. In P. V. Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers*, pages 131–158. The MIT press, 1995.

[64] C. Kirchner, H. Kirchner, and M. Vittek. *ELAN Version 1.17 User Manual.* CRIN & INRIA Lorraine, Nancy, France, first edition edition, November 1995.

[65] C. Kirchner, C. Lynch, and C. Scharff. A Fine-grained Concurrent Completion Procedure. In H. Ganzinger, editor, *Proceedings of The 6th Conference on Rewriting Techniques and Applications, Kaiserslautern, Germany*, volume 1103 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, September 1996.

[66] C. Kirchner and C. Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3) :225–262, June 1998.

[67] H. Kirchner. On the Use of Constraints in Automated Deduction. In A. Podelski, editor, *Constraint Programming : Basics and Trends*, volume 910 of *Lecture Notes in Computer Science*, pages 128–146. Springer-Verlag, 1995.

[68] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings of The 5th Conference on Rewriting Techniques and Applications, Montreal, Canada*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[69] H. Kirchner and P.-E. Moreau. A reflective extension of ELAN. In J. Meseguer, editor, *Proceedings of The First International Workshop on Rewriting Logic and its Applications, RWLW'96*, volume 4, Asilomar, Pacific Grove, CA, USA, September 1996. Electronic Notes in Theoretical Computer Science.

[70] H. Kirchner and C. Ringeissen. Combining Symbolic Constraint Solvers on Algebraic Domains. *Journal of Symbolic Computation*, 18(2) :113–155, 1994.

[71] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7) :424–435, July 1979.

[72] V. Kumar. Algorithms for Constraint-Satisfaction Problems : A Survey. *Artificial Intelligence Magazine*, 13(1) :32–44, Spring 1992.

[73] A. Kwan, E. Tsang, and J. Borrett. Phase transition in finding multiple solutions in constraint satisfaction problems. In *Workshop on Studying and Solving Really Hard Problems, First International Conference on Principles and Practice of Constraint Programming*, pages 119–126, 1995.

[74] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(497), 1960.

[75] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1993.

[76] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8 :99–118, 1977.

[77] A. K. Mackworth. Constraint Satisfaction. In S. C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, volume 1. Addison-Wesley Publishing Company, 1992. Second Edition.

[78] A. K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58 :3–20, 1992.

[79] A. K. Mackworth and E. C. Freuder. The Complexity of Some Polynomial Network Consis-

tency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25 :65–74, 1985.

[80] A. K. Mackworth and E. C. Freuder. The complexity of constraint satisfaction revisited. *Artificial Intelligence*, 59 :57–62, 1993.

[81] P. Marti and M. Rueher. A Distributed Cooperating Constraints Solving System. *International Journal of Artificial Intelligence Tools*, 4(1-2) :93–113, 1995.

[82] J. Meseguer. General Logics. In H.-D. e. a. Ebbinghaus, editor, *Logic Colloquium '87*, pages 275–329. Elsevier Science Publishers B. V., North-Holland, 1989.

[83] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[84] E. Monfroy. An Environment for Designing/Executing Constraint Solver Collaborations. Rapport de Recherche 96-R-044, Centre de Recherche en Informatique de Nancy, CRIN, Vandœuvre-lès-Nancy, France, February 1996.

[85] E. Monfroy. *Collaboration de solveurs pour la programmation logique à contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, November 1996. Also available in english.

[86] E. Monfroy. The Constraint Solver Collaboration Language of BALI. Short version of Research Report 96-R-044, CRIN, Centre de Recherche en Informatique de Nancy, Vandoeuvre-lès-Nancy, February 1996, 1997.

[87] E. Monfroy, M. Rusinowitch, and R. Schott. Implementing Non-Linear Constraints with Cooperative Solvers. In K. M. George, J. H. Carroll, D. Oppenheim, , and J. Hightower, editors, *Proceedings of The 11th ACM Annual Symposium on Applied Computing, SAC'96*, pages 63–72, Philadelphia, PA, USA, February 1996. ACM Press.

[88] U. Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Sciences*, 7 :95–132, 1974.

[89] U. Montanari and F. Rossi. An Efficient Algorithm for the Solution of Hierarchical Networks of Constraints. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proceedings of Third International Workshop on Graph-Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 440–457. Springer-Verlag, 1986.

[90] U. Montanari and F. Rossi. Constraint Solving and Programming : What next ? *ACM Computing Surveys*, 28(4), 1996. Also in CONSTRAINTS : An international journal, vol.2, n.1, 1997.

[91] P.-E. Moreau and H. Kirchner. Compilation Techniques for Associative-Commutative Normalisation. In A. Sellink, editor, *Proceedings of The Second International Workshop on the Theory and Practice of Algebraic Specifications, ASF+SDF'97*, number CRIN 97-R-129 in Electronic Workshops in Computing, eWiC web site : http ://ewic.springer.co.uk/, Amsterdam, The Netherlands, September 1997. Springer-Verlag. 12 pages.

[92] P.-E. Moreau and H. Kirchner. A Compiler for Rewrite Programs in Associative-Commutative Theories. In *Proceedings of Programming Language Implementation and Logic Programming, PLILP'98*, Lecture Notes in Computer Science. Springer-Verlag, September 1998. To appear.

[93] B. Nadel. Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.

[94] B. A. Nadel. Representation Selection for Constraint Satisfaction: A Case Study Using n-Queens. *IEEE Expert*, 5(3):16–23, June 1990.

[95] G. Nelson and D. C. Oppen. Simplifications by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[96] N. J. Nilsson. *Problem-Solving Methods in Artificial Intelligence.* Computer Science SERIES. McGraw-Hill, 1971.

[97] M. Perlin. Arc consistency for factorable relations. *Artificial Intelligence*, 53:329–342, 1992.

[98] P. Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings of The 13th International Joint Conference on Artificial Intelligence, IJCAI'93, Chambéry, France*, pages 262–267, August 29 - September 3 1993.

[99] J.-F. Puget. Future of Constraint Programming. *ACM Computing Surveys*, 28(4), 1996.

[100] C. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and K. Schulz, editors, *Proceedings of The First International Workshop Frontiers of Combining Systems, FroCoS'96*, pages 121–139. Kluwer Academic Publishers, 1996.

[101] C. Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings of The 8th Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

[102] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley Interscience Series in Discrete Mathematics. John Wiley & Sons, 1986.

[103] C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In L. Naish, editor, *Proceedings of The 14th International Conference on Logic Programming, ICLP'97*, pages 286–300, Leuven, Belgium, July 1997. The MIT press.

[104] B. M. Smith. The Phase Transition in Constraint Satisfaction Problems: A Closer Look at the Mushy Region. Technical Report 93.41, School of Computer Studies, University of Leeds, 1993.

[105] B. M. Smith. In Search of Exceptionally Difficult Constraint Satisfaction Problems. Technical Report 94.2, School of Computer Studies, University of Leeds, 1994.

[106] G. Smolka. Problem Solving with Constraints and Programming. *ACM Computing Surveys*, 28(4es), December 1996. Electronic Section.

[107] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[108] M. van den Brand, P. Klint, and C. Verhoef. Term Rewriting for Sale. In C. Kirchner and H. Kirchner, editors, *Proceedings of The Second International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15, pages 139–161, Pont-à-Mousson, France, September 1998. Electronic Notes in Theoretical Computer Science.

[109] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT press, 1989.

[110] P. van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective for Constraint Logic Programming. *Proceedings of The Eighth International Conference on Logic Programming, ICLP'91*, 2:745–759, 1991.

[111] P. van Hentenryck, V. Saraswat, and Y. Deville. Design, Implementation, and Evaluation of the Constraint Language cc(FD). In *Constraint Programming: Basic and Trends. Selected Papers of the 22nd Spring School in Theoretical Computer Sciences.* Springer-Verlag, Châtillon-sur-Seine, France, May 1994.

[112] G. Verfaillie and T. Schiex. Maintien de solution dans les problèmes dynamiques de satisfaction de contraintes: bilan de quelques approches. *Revue d'Intelligence Artificielle*, 9(3):269–309, 1995.

[113] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy 1, Nancy, France, October 1994.

[114] R. L. Walker. An Enumerative Technique for a Class of Combinatorial Problems. In *Combinatorial Analysis, Proceedings of Symposium in Applied Mathematics, Vol X, Amer. Math. Soc.*, pages 91–94, Providence, RI, USA, 1960.

[115] M. G. Wallace. Applying Constraints For Scheduling. In M. B and P. J, editors, *Constraint Programming*, NATO ASI Series. Springer-Verlag, 1994.

[116] R. J. Wallace. Why AC-3 is almost always better than AC-4 for establishing arc concsistency in CSPs. In *Proceedings of The 13th International Joint Conference on Artificial Intelligence, IJCAI'93, Chambéry, France*, pages 239–245, 1993.

[117] D. Waltz. Understanding lines drawings of scenes with shadows. In P. H. Winston, editor, *The Psychology of Computer Vision*, Computer Science Series, pages 19–91. McGraw-Hill, 1975.

[118] D. Warren. The Andorra Model. Presented al Gigalips Project Workshop, University of Manchester, 1988.

[119] C. Williams and T. Hogg. Exploiting the Deep Structure of Constraint Problems. *Artificial Intelligence*, 70:73–117, 1994.

[120] H. Williams. *Model Building in Mathematical Programming.* J. Wiley and Sons, New York, 1978.

[121] N. Wirth. *Algorithms + Data Structures = Programs.* Automatic Computation. Prentice Hall, 1976.

[122] J. Würtz and T. Müller. Constructive Disjunction Revisited. In *Proceedings of The Twentieth German Annual Conference on Artificial Intelligence, KI-96*, September 1996.

[123] J. Zhou. *Calcul de plus petits produits cartésiens d'intervalles: application au problème d'ordonnancement d'atelier.* PhD thesis, Université de la Mediterranée, Aix-Marseille II, France, March 1997.

Université HENRI POINCARÉ
NANCY 1

FACULTÉ DES SCIENCES

Monsieur **CASTRO VALDEBENITO Carlos Miguel**

DOCTORAT de l'UNIVERSITE HENRI POINCARE, NANCY-I

en INFORMATIQUE

VU, APPROUVÉ ET PERMIS D'IMPRIMER

Nancy, le 11 janvier 1999 n° 171

Le Président de l'Université

UNIVERSITÉ HENRI POINCARÉ, NANCY
LE PRÉSIDENT
J.P. FINANCE

3C-F9

Résumé

Dans cette thèse, nous modélisons la résolution de problèmes de satisfaction de contraintes (CSPs) comme un processus de déduction. Nous formalisons les techniques de résolution de CSPs sur des domaines finis en utilisant des règles de réécriture et des stratégies. Les règles expriment les transformations réalisées par ces techniques sur les contraintes et les stratégies contrôlent l'application de ces règles. Cette approche nous permet de vérifier facilement des propriétés telles que la correction, la complétude et la terminaison d'un solveur. L'utilisation d'un langage de stratégies puissant nous permet de décrire les heuristiques d'une manière très abstraite et flexible. En utilisant cette approche déductive, nous pouvons prouver l'existence ou l'inexistence d'une solution : il nous suffit de regarder le terme de preuve pour reconstruire exactement une preuve du calcul réalisé. Afin de valider notre approche, nous avons implanté le système COLETTE dans le langage ELAN.

Nous étudions tout d'abord le problème de la résolution d'un ensemble de contraintes élémentaires qui sont combinées par des opérateurs de conjonction. Nous nous intéressons ensuite à la résolution de problèmes d'optimisation en utilisant des techniques de résolution de CSPs. Nous abordons l'extension du langage de contraintes afin de considérer la combinaison des contraintes élémentaires avec les connecteurs logiques de disjonction, d'implication et d'équivalence. Le traitements des contraintes disjonctives est décrit en détail. Finalement, nous étudions la coopération de solveurs. En utilisant les opérateurs de stratégies concurrents du langage ELAN, nous décrivons d'une manière abstraite des schémas de coopération séquentiels et concurrents. Pour d'autres types de coopération, nous utilisons des primitives de bas niveau qui sont disponibles dans ELAN pour la communication entre processus.

Mots-clés: Problème de satisfaction de contraintes, Règle de réécriture, Stratégie, Système de calcul

Abstract

In this thesis, we model the resolution of constraint satisfaction problems (CSPs) as a deductive process. We formalise CSP solving techniques over finite domains using rewrite rules and strategies. The rules express the transformations over the set of constraints carried out by these techniques. The strategies control the application of these rules. This approach allows us to easily verify properties like correctness, completeness, and termination of a solver. The use of a powerful strategy language allows us to describe heuristics in an abstract and flexible way. Furthermore, using this deductive approach we can justify the existence or absence of a solution: we just have to look at the proof term in order to reconstruct exactly a proof of the computation carried out. To validate our approach, we have implemented the system COLETTE using the ELAN language.

Firstly, we study the resolution of a set of elementary constraints combined by conjunction operators. Next, we focus on the resolution of optimisation problems using CSP solving techniques. Then, we extend the constraint language in order to consider the combination of elementary constraints by the logical operators of disjunction, implication, and equivalence. The treatment of disjunctive constraints is described in detail. Finally, we study the cooperation of solvers. Using the concurrent strategy operators of the ELAN language we describe in an abstract way some schemes of sequential and concurrent cooperation. For other types of cooperation, we use low-level primitives for process communication available in ELAN.

Keywords: Constraint Satisfaction Problem, Rewrite Rule, Strategy, Computational System