Développement d'une technique d'acquisition de contraintes basée sur le nombre de solutions

Mémoire

Christopher Coulombe

Maîtrise en informatique - avec mémoire
Maître ès sciences (M. Sc.)

Québec, Canada

© Christopher Coulombe, 2023

Développement d'une technique d'acquisition de contraintes basée sur le nombre de solutions

Mémoire

Christopher Coulombe

Sous la direction de:

Claude-Guy Quimper, directeur de recherche

Résumé

Plusieurs paradigmes de programmation existent pour aider à résoudre des problèmes d'optimisation combinatoire, l'un d'entre eux étant la programmation par contraintes. L'idée de ce paradigme consiste à modéliser le problème à résoudre à l'aide de *contraintes*, c'est-à-dire des déclarations qui forcent les variables du problème à respecter une relation mathématique. Les contraintes des problèmes ont habituellement des paramètres qui permettent de préciser la relation mathématique à respecter et des variables de décision qui représentent les variables pour lesquelles la relation mathématique doit s'appliquer. Bien qu'intéressant en soi, la programmation par contraintes peut également s'étendre sur d'autres concepts, notamment la modélisation automatique. L'acquisition ou *apprentissage de contraintes* consiste à apprendre les différentes contraintes, incluant les valeurs des paramètres, qui peuvent expliquer un ensemble d'exemples fournis. L'apprentissage de contraintes peut être utile dans plusieurs situations, comme l'apprentissage de structures d'horaires d'hôpitaux à l'aide d'anciens exemples d'horaires.

L'apprentissage de contraintes est encore un domaine nouveau pour lequel les stratégies doivent encore être adaptées ou développées. Les techniques d'acquisition existantes varient en genre, incluant des méthodes qui créent des solutions artificielles pour interagir avec un utilisateur ou des approches qui se basent sur des analyses mathématiques rigoureuses de solutions pour faire des choix sans jamais communiquer avec l'utilisateur.

Dans ce mémoire, nous explorons une nouvelle méthode pour performer l'acquisition de contraintes. Le critère principal de la méthode développée est basé sur le nombre de solutions du modèle considéré et utilise des outils de *dénombrement*. Notre technique performe bien sur les problèmes essayés et ouvre la porte à une nouvelle manière d'apprivoiser les problèmes d'acquisition de contraintes.

Abstract

Several programming paradigms exist to help solve combinatorial optimization problems, one of them being constraint programming. The idea of this paradigm is to model the problems to solve using *constraints*, i.e. statements that force the variables of the problem to respect a mathematical relation. The constraints of a problem usually have parameters that allow to specify the mathematical relationship to be respected and decision variables that represent the variables on which the mathematical relationship must be applied. Although interesting in itself, constraint programming can also expand on other concepts, such as the automatisation of the modeling process. *Constraints acquisition* consists in learning the different constraints, including parameter values, which can explain a set of examples provided. Constraint acquisition can be useful in multiple situations, such as learning structures in schedules for hospitals using old schedules.

Constraint learning is still a new area for which strategies still need to be adapted or developed. The existing techniques of acquisition varies widely in style, including methods that create artificial solutions to interact with a user or approaches which are based on complex mathematical analyzes of real solutions to make choices without ever communicating with the user.

In this thesis, we explore a new method to perform the acquisition of constraints. The main criterion of the developed method is based on the number of solutions of the considered model and uses tools of *model counting*. Our technique works well on proven problems and opens the door to a new way of approaching acquisition constraint problems.

Table des matières

Résumé ii

Abstract iii

Table des matières iv

Liste des tableaux vi

Liste des figures vii

Remerciements x

Avant-propos xi

Introduction 1

1 Notions préliminaires 3
- 1.1 Programmation par contraintes . . . 3
- 1.2 Modèles SAT . . . 11
- 1.3 Dénombrement propositionnel . . . 19
- 1.4 Acquisition de contraintes . . . 28

2 Constraint Acquisition Based on Solution Counting 34
- 2.1 Résumé . . . 34
- 2.2 Abstract . . . 35
- 2.3 Introduction . . . 35
- 2.4 General Background . . . 36
- 2.5 The CABSC approach . . . 37
- 2.6 Framework . . . 39
- 2.7 Experiments . . . 43
- 2.8 Results and Discussion . . . 47
- 2.9 Conclusion . . . 52
- Références . . . 52

3 Travaux subséquents avec CABSC 55
- 3.1 Problème de préséances . . . 55
- 3.2 Travaux futurs . . . 56

Conclusion 60

Bibliographie **61**

Liste des tableaux

1.1 Effets de la propagation après un branchement pour le problème de l'exemple 5. 9
1.2 Représentation des variables logiques pour l'exemple 12. 15
1.3 Représentation des variables logiques pour l'exemple 13. 17
1.4 Exemple d'assignation des variables logiques pour l'exemple 13. 18

Liste des figures

1.1 Exemple d'arbre de recherche. 7
1.2 Exemple de filtrage dans un arbre de recherche. 8
1.3 Exemple d'arbre de recherche avec des bornes inférieures optimales. 11
1.4 Exemple de fonctions monotones. 25

2.1 Simple example 41
2.2 Flow chart for the CABSC approach 43
2.3 Classification of the instances in percentage for each number of examples. CabscSolver uses `GANAK` as the only model counter. 47
2.4 Hypothetical best results for each benchmark 49
2.5 Measures of time for all instances using `GANAK` alone 50
2.6 Measures of time for all instances using `GANAK` with `ApproxMC4` 51

À mon chat Raman.
Il m'a appris la persévérance.
« Manger, manger, manger,
manger, manger, ... » - Raman

If more of us valued food and
cheer and song above hoarded
gold, it would be a merrier world.

J.R.R. Tolkien

Remerciements

Écrire un mémoire pendant la pandémie était un défi que je n'avais pas pleinement envisagé au départ. Heureusement, j'ai pu compter sur l'aide et le support de nombreuses personnes sans qui je ne sais pas où j'en serais. Cette page vous appartient et sachez que je me compte exceptionnellement chanceux de vous avoir dans mon entourage.

Je veux premièrement prendre le temps de remercier mon directeur de recherche Claude-Guy Quimper. Un des aspects stressants de la maîtrise est de comprendre si la direction que l'on prend est la bonne. Tu as toujours mis l'emphase sur l'importance que l'on se rencontre à chaque semaine pour discuter et tu as toujours répondu à mes questions, aussi tard soit-il. Savoir que semaine après semaine je pouvais compter sur toi pour m'aider était un cadeau inestimable. Merci Claude-Guy.

Je veux aussi remercier ma conjointe Frédérique Montreuil (et mon chat Raman) qui a toujours été là pour moi, même dans les moments plus difficiles ou lorsque je devais rédiger tardivement le soir. Tu es un rayon de soleil dans ma vie et je suis exceptionnellement chanceux que l'on soit ensemble. Je t'aime *Petit Coeur*.

Merci à Kim Rioux-Paradis qui a été une source d'inspiration et de courage durant mes études. J'ai tant appris avec toi, soit en travaillant à tes côtés ou juste en discutant avec toi. Tu étais toujours là pour m'aider quand j'avais des doutes et me supporter quand j'avais des questions. Merci pour tout !

Je veux aussi donner un gros merci à l'ensemble du laboratoire *PPC*, particulièrement à Yanick Ouellet, Alexandre Mercier Aubin, Frédéric Berthiaume, Nicolas Blais, Raphaël Boudreault, Anthony Deschênes, Jonathan Gaudreault, Nicolas LeBlanc, Marc-André Ménard, Jovial Cheukam Ngouonou et Pankaj Singh. Mon expérience n'aurait jamais été la même sans vous !

Finalement, je veux remercier Luc Lamontagne et Pascal Germain qui ont pris le temps d'évaluer mon mémoire. Je veux aussi remercier le FRQNT pour son support financier tout au long de ma maîtrise.

Avant-propos

Ce mémoire présente les travaux de recherche faits au cours de ma maîtrise sous la direction de Claude-Guy Quimper. Dans le cadre de ces travaux, un article intitulé *Constraint Acquisition Based on Solution Counting* [24] a été écrit par Christopher Coulombe et Claude-Guy Quimper puis présenté par Christopher Coulombe à la conférence scientifique sur la programmation par contraintes *28th International Conference on Principles and Practice of Constraint Programming* (CP2022) le 4 août 2022 de même qu'au *Doctoral Program* de cette même conférence. Dans cet article, je suis auteur principal alors que mon directeur de recherche Claude-Guy Quimper est auteur secondaire. Pour ces travaux, j'ai participé activement à l'établissement d'une direction de recherche, au développement de la stratégie d'acquisition de contraintes ainsi que la création du code prototype nécessaire pour exécuter les tests et valider notre stratégie. Claude-Guy Quimper a particpé en tant que superviseur de la recherche. L'article écrit en anglais est inséré tel quel dans ce mémoire au chapitre 2, à l'exception d'une section en français qui est ajoutée pour décrire l'article.

Introduction

En informatique moderne, l'une des difficultés qui attirent l'attention des chercheurs de partout dans le monde est la résolution efficace des *problèmes combinatoires*, c'est-à-dire un problème dont l'espace possible de solutions est discret et sujet à un ensemble de restrictions. De tels problèmes nous entourent régulièrement, incluant dans les processus de sécurité Internet, la modélisation d'outils communs comme les automobiles ou la génération d'horaire comme dans les hôpitaux.

Considérant que l'espace de recherche pour plusieurs de ces problèmes est exponentiel, même avec un ordinateur performant, ces problèmes prendraient des années à résoudre si aucune technique particulière n'est utilisée. C'est pour cette raison que divers paradigmes se sont développés pour résoudre efficacement ces problèmes. Cependant, il existe des problèmes pour lesquels des experts trouvent des solutions depuis des années, parfois avant même le développement de stratégies particulières. C'est le cas notamment pour la création d'horaires d'hôpitaux, un problème qui devait être résolu bien avant la création des technologies informatiques modernes.

Un horaire est généralement une assignation de tâches étalée sur une période de temps variable. Chacune de ces tâches doit respecter des restrictions particulières. Par exemple, il est intuitif de dire que certaines tâches doivent être faites par un médecin tandis que d'autres tâches pourraient être faites par des infirmiers ou infirmières. D'autres types de restrictions pourraient exister, par exemple des limitations au niveau du temps. Une tâche A pourrait nécessiter d'être fait avant une tâche B, ce qui devrait être visualisé lors de la création de l'horaire. L'assignation des salles d'opération constitue également un aspect de la gestion d'horaires qui peut s'avérer compliqué à manipuler. Le grand nombre de règles à considérer explique l'aspect combinatoire des horaires où chaque règle contraint les éléments du problème à respecter les demandes.

En considérant une restriction comme une *contrainte* et un moment de l'horaire comme une *variable*, la *programmation par contraintes* devient un paradigme intuitif pour résoudre ce type de problèmes combinatoires. Ce type de programmation représente un problème à résoudre avec des variables qui ont chacune un *domaine* attribué, c'est-à-dire un ensemble de valeurs qu'il est possible de prendre. Dans le cas d'un horaire, une variable pourrait être le moment où une tâche est exécutée et son domaine est l'ensemble des plages horaires où les ressources

sont disponibles. Ces variables sont assujetties à des *contraintes*, c'est à dire des relations mathématiques entre les variables qui empêchent les affectations jugées non valides pour le problème. Il existe une grande variété de contraintes et chacune d'entre elles possède des algorithmes qui parcourent efficacement les domaines de variables pour examiner rapidement l'espace de solutions.

Typiquement, en programmation par contrainte, un expert souhaite trouver ou optimiser une solution à un problème combinatoire et tente alors de *modéliser* ce problème, c'est-à-dire lister les contraintes qui forment une représentation efficace du problème à résoudre. Un *solveur* peut alors retourner une solution. Dans le cas d'un hôpital, des horaires existent déjà, mais la modélisation est absente. Dans ces situations, l'intérêt devient donc de tenter de faire de l'apprentissage automatique de modèles à partir de ces exemples d'horaires. C'est ce que l'on appelle l'*acquisition de contraintes*.

L'acquisition de contraintes est un procédé où un programme peut interagir avec un utilisateur et analyser des exemples de solutions. Selon le contexte d'apprentissage, des exemples *positifs* peuvent être fournis, c'est-à-dire des exemples dont on sait qu'ils sont des solutions au problème à apprendre. Un horaire d'hôpital serait un exemple positif tandis qu'un exemple *négatif* serait un cas où l'assignation de variables n'est pas une solution au problème.

Plusieurs techniques existent pour apprendre les contraintes d'un problème à partir d'exemples positifs ou négatifs. Certaines stratégies exploitent les interactions avec les utilisateurs alors que d'autres se concentrent sur l'exploitation de corrélations entre une base de données et les exemples fournis. Une technique qui a été testée récemment dans la littérature consiste à utiliser le nombre de solutions des contraintes pour déduire la probabilité qu'une contrainte explique les exemples donnés.

Nous proposons dans ce mémoire une nouvelle manière de faire de l'acquisition de contraintes basée sur ce même principe. Notre méthode permet de considérer les interactions entre les variables et les contraintes, incluant les contraintes connues qui ne sont pas à apprendre. Cette méthode, appelée CABSC, a été testée sur des exemples d'horaires d'hôpitaux simplifiés et a montré des résultats prometteurs. Des travaux ultérieurs sur l'acquisition de contraintes à partir du dénombrement du nombre de solutions des contraintes seront une venue intéressante pour les chercheurs du domaine.

Ce mémoire est divisé en trois chapitres. Le premier chapitre contient l'ensemble complet des notions nécessaires à la compréhension de la problématique. Le deuxième chapitre contient un article décrivant l'état de l'art en acquisition de contraintes et comment notre approche fonctionne comparrément à ces techniques. Le dernier chapitre explore quelques chemins de pensées par rapport aux travaux subséquents possibles avec CABSC et comment cette technique pourrait être améliorée.

Chapitre 1

Notions préliminaires

Ce chapitre introduit les notions nécessaires à la compréhension des problèmes que nous avons étudiés dans le cadre de ce mémoire ainsi que les contributions apportées. Premièrement, nous présentons la programmation par contraintes dans la section 1.1. Par la suite, nous introduisons les modèles booléens SAT dans la section 1.2, puis le dénombrement de modèles dans la section 1.3. Finalement, nous décrivons à la section 1.4 le concept d'acquisition de contraintes.

1.1 Programmation par contraintes

1.1.1 Définition et concepts

La *programmation par contraintes* (PPC) est un paradigme puissant pour résoudre les problèmes combinatoires qui s'appuit sur un large éventail de techniques touchant à l'intelligence artificielle, à l'informatique, aux bases de données, aux langages de programmation et à la recherche opérationnelle [59]. Pour ce type de paradigme, la résolution des problèmes se fait en deux étapes distinctes. La première consiste en une *modélisation* du problème à résoudre, c'est-à-dire établir à l'aide d'un langage de modélisation les expressions qui caractérisent le problème. Cette modélisation mène ensuite à la *résolution* du problème où un *solveur de contraintes* cherche à trouver le plus efficacement possible une solution dans l'*espace de recherche*. Les algorithmes utilisés par le solveur de contraintes sont dépendants de la modélisation écrite.

En programmation par contraintes, un problème est décrit à l'aide d'un ensemble de *variables de décision*. Une variable de décision est une variable dont la valeur est inconnue, mais dont les valeurs possibles sont connues. L'ensemble des valeurs possibles d'une variable de décision s'appelle le *domaine* d'une variable. L'étape de la résolution s'occupe d'associer ces variables à l'une des valeurs du domaine pour résoudre le problème.

Les variables de décision sont mises en relation à l'aide de *contraintes*. Une contrainte sert à

caractériser le problème en forçant des interactions entre les variables. Une contrainte peut être de nature arithmétique ($x + 1 \geqslant y$), logique ($p \wedge q, p \vee q$) ou globale. Les contraintes globales sont des classes de contraintes définies par une formule pouvant interagir avec un nombre arbitraire de variables de décision [59]. L'ensemble des variables qui sont affectées par une contrainte est appelé la *portée* de la contrainte.

1.1.2 Problème de satisfaction de contraintes

Alors que les contraintes peuvent être vues comme des relations entre des variables de décision, un *problème de satisfaction de contraintes*, en anglais *constraint satisfaction problem* (CSP), indique quelles relations doivent être maintenues entre les variables de décision données [59]. Un problème de satisfaction de contraintes est résolu lorsque chaque variable a une valeur qui satisfait toutes les contraintes sur cette variable [60]. La définition 1 décrit formellement un problème de satisfaction de contraintes suivant la définition offerte par [59].

> **Définition 1** (Problème de satisfaction de contraintes)**.** « Un CSP $\mathcal{P}$ est un triplet $\mathcal{P} = \langle X, \text{dom}, C \rangle$ où X est un n-tuple de variables de décision $X = \langle X_1, X_2, \ldots, X_n \rangle$, dom est une fonction qui fait correspondre une variable $X_i \in X$ à son domaine respectif, C est un t-tuples de contraintes $C = \langle C_1, C_2, \ldots, C_t \rangle$. Une contrainte C_j est une paire $\langle R_j, S_j \rangle$ où $S_j \subseteq X$ est la portée de la contrainte et R_j est une relation sur les variables dans S_j. En d'autres mots, R_j est un sous-ensemble du produit cartésien des domaines des variables dans S_j. Une solution à un CSP $\mathcal{P}$ est une assignation aux variables $X_1 = v_1, \ldots, X_n = v_n$ telle que $v_j \in \text{dom}(X_j)$ $\forall 1 \leqslant j \leqslant n$ et chaque C_j est satisfait de manière à ce que le tuple $\langle v_1, \ldots, v_n \rangle$ projeté sur S_j soit un tuple dans R_j. » [59]

Pour résoudre un problème de satisfaction de contraintes, il faut d'abord effectuer un modèle du problème. Nous discutons des principes de modélisation dans la section 1.1.3 puis des principes de résolution dans la section 1.1.4.

1.1.3 Modélisation d'un problème de satisfaction de contraintes

La modélisation est une étape importante, car la chaîne de calculs qu'effectue le solveur de contraintes pour effectuer la résolution en est dépendante. La modélisation d'un CSP consiste à choisir comment représenter le problème avec des variables de décision et comment intégrer les relations entre ces variables à l'aide de contraintes. Les contraintes doivent être choisies parmi un catalogue de contraintes qui est propre à chaque solveur. Il est très fréquent que plusieurs combinaisons de contraintes soient équivalentes du point de vue des relations entre les variables de décision. L'exemple 1 présente une figure de cas simple où deux modélisations sont possibles pour résoudre un CSP.

Exemple 1. Imaginons un problème simple où on voudrait choisir trois nombres parmi 1, 2 et 3 avec la restriction qu'aucun de ces nombres ne doit se répéter. Pour ce problème, une modélisation simple pourrait être de définir trois variables de décision x, y et z dont les domaines sont $\{1, 2, 3\}$. Un premier modèle utiliserait des contraintes arithmétiques. Nous pourrions mettre les contraintes $x \neq y, x \neq z$ et $y \neq z$ dans le modèle, ce qui forcerait les trois variables à prendre des valeurs distinctes. De manière équivalente, nous pourrions modéliser ce problème avec une contrainte globale appelée AllDifferent telle que AllDifferent(x, y, z). Cette contrainte s'assure que toutes les variables dans sa portée auront des valeurs différentes [46].

En pratique, un modèle doit être écrit avec un langage de modélisation. Il existe plusieurs langages tels MiniZinc [53], Essence [34, 35] ou Numberjack [43]. Chaque langage a ses particularités, mais certains principes de base sont typiquement maintenus. Pour la majorité des langages, les variables de décision doivent être déclarées. Chaque variable peut avoir un domaine spécifié. Chaque langage supporte également un ensemble de contraintes qui peuvent être utilisées pour représenter le problème à modéliser sous la forme d'un CSP. L'exemple 2 modélise un CSP simple avec trois langages différents.

Exemple 2. Reprenons la dernière modélisation faite dans l'exemple 1 où on voudrait choisir trois nombres parmi 1, 2 et 3 de manière à ce qu'aucun de ces nombres ne se répète. On peut écrire ce modèle simple avec plusieurs langages de modélisation. Le code 1.1 est écrit en MiniZinc [53], le code 1.2 est écrit en Essence [34, 35] et le code 1.3 est écrit en Python utilisant Numberjack [43].

```
include "globals.mzn";
set of int: domaine = 1..3;
var domaine : x;
var domaine : y;
var domaine : z;

constraint all_different([x,y,z]);
solve satisfy;

```

Extrait de fichier 1.1 – Exemple de modèle avec MiniZinc

```
letting Domaine be domain int(1..3)
find x : Domaine
find y : Domaine
find z : Domaine
such that allDiff([x,y,z])

```

Extrait de fichier 1.2 – Exemple de modèle avec Essence

```
from Numberjack import *

def get_model():
    x = Variable(1,3);
    y = Variable(1,3);
    z = Variable(1,3)
    model = Model(AllDiff([x,y,z]))
    return [x,y,z], model

def solve(param):
    (x, y, z), model = get_model()
    solver = model.load(param['solver'])
    solver.solve()
    x_sol = x.get_value()
    y_sol = y.get_value()
    z_sol = z.get_value()
    sol = (f"x: {x.get_value()}" +
           f", y: {y.get_value()}" +
           f", z:{z.get_value()}")
    return sol

default = {'solver': 'MiniSat'}

if __name__ == '__main__':
    param = input(default)
    print(solve(param))

```

Extrait de fichier 1.3 – Exemple de modèle avec Numberjack

Lorsque la modélisation est complétée, le CSP obtenu peut être procédé par un solveur de contraintes qui trouve alors une solution au modèle. Nous présentons dans la section 1.1.4 comment les solveurs de contraintes font pour résoudre un CSP.

1.1.4 Résolution d'un problème de satisfaction de contraintes

Pour trouver une solution à un problème de satisfaction de contraintes, il faut que chaque variable de décision soit fixée à une valeur de son domaine. Pour y arriver, un solveur de contraintes s'occupe de créer un arbre de recherche qui représente l'espace de recherche du CSP modélisé. Pour créer un tel arbre, le solveur de contraintes crée une *racine*, c'est-à-dire le premier noeud de l'arbre où aucune variable n'est associée à une valeur. Le solveur de contraintes effectue alors un *branchement* et crée un nouveau noeud enfant en faisant une *assignation* où il fixe l'une des variables de décision à une valeur de son domaine. Un noeud dont seulement certaines variables de décision sont associées à une valeur est appelé une *solution partielle*. Une solution au CSP est donc un noeud feuille dont toutes les variables de décision sont fixées.

Chaque contrainte possède un *algorithme de vérification*, c'est-à-dire un mécanisme qui permet de vérifier si les variables dans la portée forment une solution à la contrainte. À chaque fois que l'ensemble des variables dans la portée d'une contrainte sont assignées, ces algorithmes de vérification sont appelés pour vérifier que le branchement effectué par le solveur est valide. Dans le cas où l'algorithme de vérification accepte les branchements effectués, le solveur peut continuer sa recherche en effectuant davantage de branchements jusqu'à trouver une solution. Dans le cas contraire, le solveur ne peut plus continuer à faire des branchements, car il n'existe pas de solution au CSP dans les noeuds descendants. Il s'agit d'une *incohérence*.

Exemple 3. Considérons les variables x, y et z sujettes aux contraintes AllDifferent(x, y, z) et $x < y$ avec $\text{dom}(x) = \{1, 2\}$, $\text{dom}(y) = \{1, 3\}$ et $\text{dom}(z) = \{2, 3\}$. La figure 1.1 illustre un arbre de recherche pour ce CSP en appliquant les algorithmes de vérification. Les noeuds en rouge correspondent aux branchements ayant mené à une incohérence. Les noeuds en vert correspondent aux solutions possibles. Les noeuds du deuxième niveau sont vérifiés par la contrainte $x < y$ tandis que les noeuds du dernier niveau sont vérifiés par la contrainte AllDifferent(x, y, z).

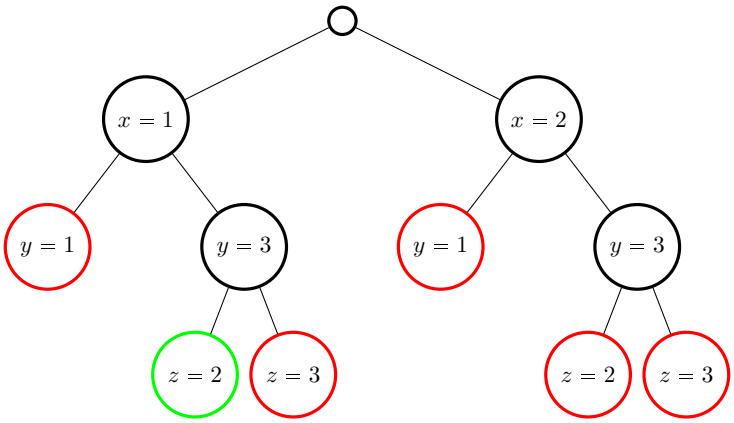

Figure 1.1 – Exemple d'arbre de recherche.

Lorsqu'une incohérence survient, le solveur de contraintes peut alors provoquer un *retour arrière*. Un retour arrière permet au solveur de retourner au noeud parent où il n'y a pas d'incohérence de notée.

En plus des algorithmes de vérification, chaque contrainte possède un *algorithme de filtrage*. Les algorithmes de filtrage servent à retirer les valeurs du domaine des variables qui rendent impossible l'obtention d'une solution si on la sélectionnait. Ce type de procédé permet de drastiquement réduire la taille de l'arbre de recherche. Si le filtrage produit un domaine vide pour une variable, alors il s'agit d'une incohérence et un retour arrière doit être fait.

Exemple 4. Considérons les variables x, y et z sujettes aux contraintes $x \neq y$, $x \neq z$ et $y \neq z$ avec $\text{dom}(x) = \{1, 2\}$, $\text{dom}(y) = \{1, 2\}$ et $\text{dom}(z) = \{1, 2, 3\}$. La figure 1.2 présente un arbre de recherche où le filtrage des domaines est indiqué par une barre rouge (~~exemple~~), ce qui mène à une incohérence lorsque z est fixé à 1. L'incohérence vient du fait qu'avec $z = 1$, il est impossible que x ou y puisse prendre la valeur 1. x et y doivent donc être assignés à la valeur restante dans leur domaine, c'est-à-dire 2, mais il est impossible d'avoir x et y avec la même valeur à cause de la contrainte $x \neq y$. La même chose se produit lorsque l'on sélectionne $z = 2$.

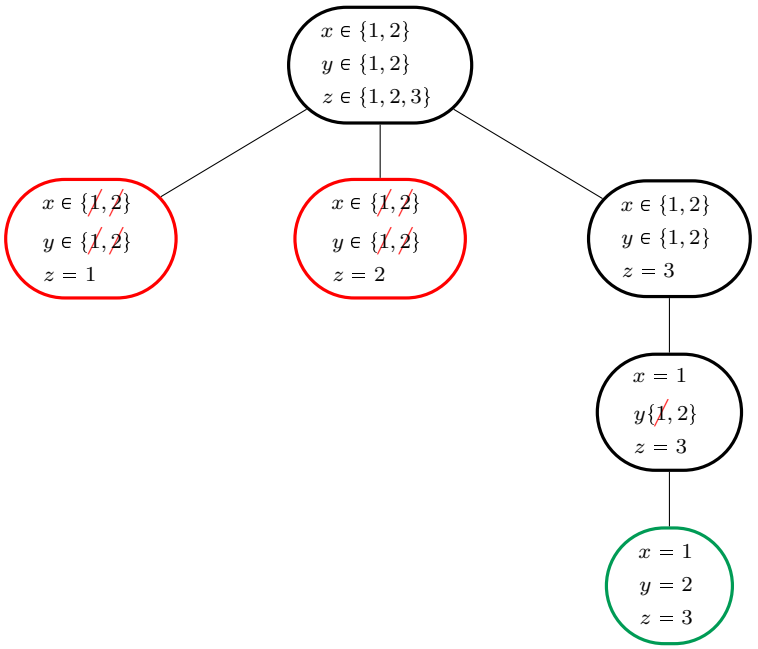

FIGURE 1.2 – Exemple de filtrage dans un arbre de recherche.

Lorsque le domaine d'une variable change, par exemple lors du branchement sur une variable, une *propagation* est faite à travers les contraintes du problème. Une propagation est un procédé où le solveur exécute l'algorithme de filtrage de chaque contrainte pour lesquelles le domaine d'une variable de décision à changé. Ainsi, lorsque le domaine d'une variable est réduit, après un branchement ou une filtrage, une propagation est faite pour analyser si le changement de domaine peut entraîner d'autres changements de domaines. Ces nouveaux changements peuvent être faits sur la même variable ou sur n'importe quelle variable du problème. Si un domaine change à nouveau, une autre propagation est faite jusqu'à l'obtention d'une *cohérence locale des domaines*, c'est-à-dire que les algorithmes de filtrage ne peuvent plus réduire les domaines des variables. L'exemple 5 développe un cas à 3 contraintes où l'effet de la propagation lance plusieurs fois les mêmes algorithmes de filtrage jusqu'à l'obtention d'une cohérence locale des domaines.

Exemple 5. Considérons les variables x, y et z sujettes aux contraintes $C_1 := x = z \cdot y$, $C_2 := y < x$ et $C_3 := x > z$ avec les domaines $\text{dom}(x) = \{1, 2, 3, 4, 5, 6\}$ et $\text{dom}(y) = \{1, 2, 3, 4, 5\}$. Admettons que le solveur vient de brancher z à la valeur 2, alors par propagation, les algorithmes de filtrage des contraintes C_1 et C_3 doivent être exécutés.

Après l'algorithme de filtrage de C_1, 1, 3 et 5 sont retirés de $\text{dom}(x)$ puisque x doit forcément être un multiple de 2. Lors de l'exécution du même algorithme, 4 et 5 sont retirés du domaine de y, car ces valeurs deviennent incohérentes avec le domaine de x. Comme x et y font parti de toutes les contraintes, le solveur doit exécuter l'algorithme de filtrage de C_2 en plus de celui de C_3 déjà prévu.

En appliquant l'algorithme de filtrage de C_3, on se rend compte que x ne peut pas être égal à 2 et cette valeur est retirée du domaine. Comme x appartient à la portée des contraintes C_1, C_2 et C_3, mais que le solveur a déjà prévu faire l'algorithme de filtrage de C_2, on n'ajoute que C_1 dans la liste des algorithmes de filtrage à exécuter. Le solveur continue ainsi de suite jusqu'à ce que les domaines des variables soient stables. Le tableau 1.1 donne l'état des domaines pour chaque filtrage fait jusqu'à ce que la cohérence locale des domaines soit obtenue.

Contrainte	$\text{dom}(x)$	$\text{dom}(y)$	$\text{dom}(z)$	Contraintes à filtrer
	$\{1, 2, 3, 4, 5, 6\}$	$\{1, 2, 3, 4, 5\}$	$\{2\}$	$\{C_1, C_3\}$
C_1	$\{2, 4, 6\}$	$\{1, 2, 3\}$	$\{2\}$	$\{C_3, C_2\}$
C_3	$\{4, 6\}$	$\{1, 2, 3\}$	$\{2\}$	$\{C_2, C_1\}$
C_2	$\{4, 6\}$	$\{1, 2, 3\}$	$\{2\}$	$\{C_1\}$
C_1	$\{4, 6\}$	$\{2, 3\}$	$\{2\}$	$\{C_2\}$
C_2	$\{4, 6\}$	$\{2, 3\}$	$\{2\}$	$\{\}$

TABLE 1.1 – Effets de la propagation après un branchement pour le problème de l'exemple 5.

1.1.5 Problème d'optimisation de contraintes

Il est fréquent qu'un problème de satisfaction de contraintes admette plusieurs solutions, mais une seule solution est nécessaire pour résoudre le problème. Selon le contexte, une solution pourrait être considérée meilleure qu'une autre. Dans ces situations, on utilise ce qu'on appelle une *fonction objectif*. Il s'agit d'une fonction mathématique qui prend en paramètre certaines variables du problème et attribue une valeur numérique à la solution évaluée. Lorsque l'on utilise de telles fonctions pour évaluer les solutions, l'objectif est typiquement de trouver la solution au problème qui maximise (ou minimise) le domaine de la fonction objectif. Résoudre le problème de satisfaction de contraintes devient alors un problème d'optimisation. Lorsque l'on cherche à optimiser une solution par le biais d'une fonction objectif, le problème à résoudre est un *problème d'optimisation de contraintes*, en anglais *constrained optimisation problem* (COP), au lieu d'un problème de satisfaction de contraintes. La définition 2 introduit

formellement un problème d'optimisation de contraintes.

> **Définition 2** (Problème d'optimisation de contraintes). « Un problème d'optimisation de contraintes (COP) est un CSP $\mathcal{P}$ défini sur les variables $X_1, \ldots, X_n$, avec une fonction objectif $f : \mathrm{dom}(X_1) \times \cdots \times \mathrm{dom}(X_n) \to \mathbb{Q}$ qui attribue une valeur à chaque affectation de valeurs aux variables. Une solution optimale à un COP de minimisation (maximisation) est une solution d à $\mathcal{P}$ qui minimise (maximise) la valeur de $f(d)$. La valeur de la fonction objectif est souvent représentée par une variable z, avec la « contrainte » `maximise` z ou `minimise` z pour une maximisation ou un problème de minimisation, respectivement. » [59]

La résolution des COP se fait en cherchant dans un arbre de recherche à l'aide d'une technique appelée *séparation et évaluation*, ou *Branch-and-bound* en anglais, une technique qui ressemble à la méthode utilisée pour résoudre les CSPs. La différence entre deux méthodes est le calcul d'une *borne inférieure* (supérieure) sur la fonction objectif pour les *problèmes de minimisation* (maximisation). Une borne inférieure (supérieure) est une valeur attribuée à un noeud de l'arbre qui est garantie d'être inférieure (supérieure) à la fonction objectif de toutes les solutions découlant du noeud courant. Donc, pour chaque noeud de l'arbre, une borne inférieure (supérieure) indique la valeur minimale (maximale) que peut prendre la fonction objectif si le solveur continuait à faire des branchements avec ce noeud. L'exemple 6 procède à un cas de figure où la notion de bornes inférieures est incluse. Dans cet exemple, les bornes inférieures prennent les valeurs optimales selon la fonction objectif définie, mais ces bornes pourraient prendre des valeurs plus basses et rester valides.

> **Exemple 6.** Considérons un problème où les variables x et y sont sujettes à la contrainte $x \neq y$ avec $\mathrm{dom}(x) = \{1, 2, 3\}$ et $\mathrm{dom}(y) = \{1, 2, 3\}$. Ce problème contient six solutions distinctes. Considérons le cas où la qualité de la solution est exprimée par la fonction objectif $-x^2 + 4x + y$ et dont le but est d'être minimisée. La figure 1.3 présente un arbre de recherche en indiquant les valeurs que pourraient prendre les bornes inférieures pour chaque noeud.

Les bornes inférieures sont particulièrement utiles pour élaguer des sections de l'arbre de recherche qui garantissent de ne pas offrir une solution satisfaisante. Dans l'exemple 6, si la solution $x = 1, y = 2$ est sélectionnée en premier, nous avons la connaissance d'une solution dont le résultat de la valeur de la fonction objectif est 5. Le solveur de contraintes continue quand même à chercher une meilleure solution, car il existe potentiellement un noeud de l'arbre dont la fonction objectif aurait une valeur de 4. Cette information est connue du solveur puisque le premier noeud racine a une borne inférieure égale à 4. Considérant ces informations, il ne sert à rien au solveur d'explorer le cas où $x = 2$ dans l'arbre de recherche, car cette section de l'arbre ne peut pas offrir une solution de meilleure qualité que celle trouvée. Par contre, le cas où $x = 3$ est exploré puisque la borne inférieure est plus basse que la valeur

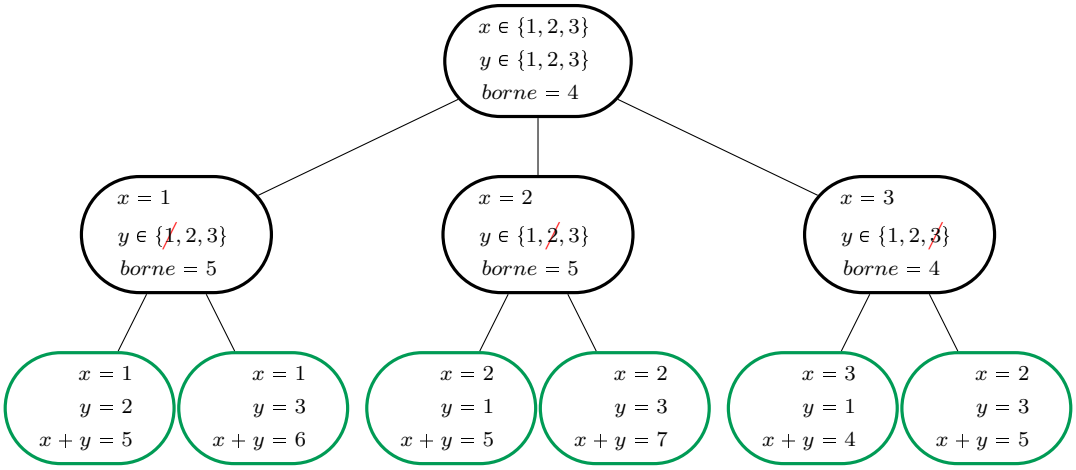

FIGURE 1.3 – Exemple d'arbre de recherche avec des bornes inférieures optimales.

de la fonction objectif de la solution trouvée.

Pour certains problèmes, évaluer une borne inférieure optimale sur la fonction objectif est en soi un problème et peut être un processus qui prend du temps. Par exemple, un des problèmes de la littérature s'appelle le problème du commis voyageur [3]. Pour ce problème, évaluer une borne inférieure qui représente optimalement la fonction objectif est particulièrement difficile à faire en un temps raisonnable [44]. Plus la borne inférieure est près de l'optimale, plus l'arbre de recherche est élagué et plus le parcourir est rapide.

1.2 Modèles SAT

Tous les modèles utilisés dans les exemples de la section 1.1 sont des problèmes en nombres entiers, c'est-à-dire des problèmes où les variables ont des domaines entiers. Un CSP peut aussi être exprimé à l'aide de variables de décision booléenne, ce qui aboutit à un *modèle de satisfaisabilité*, ou un *modèle SAT*. Nous présentons dans cette section les définitions et concepts nécessaires pour interpréter les modélisations booléennes et nous introduisons comment les modèles en nombres entiers peuvent être traduits en modèles SAT.

1.2.1 Définitions et concepts

Une variable booléenne (ou variable logique) est une variable qui ne peut que prendre la valeur *vraie* ($\top$) ou la valeur *fausse* ($\bot$). C'est ce qu'on appelle un *atome booléen*. Ce type particulier de variables est sujet à certaines opérations qui ne peuvent pas être appliquées sur des nombres entiers. Les définitions 3, 4 et 5 décrivent trois importantes opérations logiques.

> **Définition 3** (Négation logique)**.** La négation logique est une opération unaire qui produit la valeur $\top$ si et seulement si la variable logique a la valeur $\bot$. Cette opération est dénotée $\neg$ tel que $\neg p$ pour une variable logique p.

Définition 4 (Disjonction logique)**.** La disjonction logique est une opération entre deux variables logiques qui produit la valeur $\top$ si et seulement si au moins l'un des deux opérandes a la valeur $\top$. Cette opération est dénotée $\vee$ tel que $p \vee q$ pour deux variables logiques p et q.

Définition 5 (Conjonction logique)**.** La conjonction logique est une opération entre deux variables logiques qui produit la valeur $\top$ si et seulement si les deux opérandes ont la valeur $\top$. Cette opération est dénotée $\wedge$ tel que $p \wedge q$ pour deux variables logiques p et q.

Les opérations logiques peuvent être utilisées ensemble pour former une *formule de logique propositionnelle*, aussi appelée *expression booléenne*. La définition 6 décrit plus formellement une formule de logique propositionnelle.

Définition 6 (Formule de logique propositionnelle)**.** Une formule de logique propositionnelle est une expression construite à partir de variables booléennes, d'opérateurs de conjonction, de disjonction, de négation et de parenthèses. Une formule est dite *satisfiable* si elle peut être rendue vraie ($\top$) en attribuant des valeurs logiques appropriées à ses variables.

Exemple 7. $p \vee \neg q$ est une formule de logique propositionnelle satisfiable, car si p est assignée à la valeur booléenne $\top$ et q à la valeur booléenne $\bot$, la formule est évaluée à $\top$. $p \wedge \neg p$ n'est pas une formule de logique propositionnelle satisfiable, car il n'existe pas de valeur pour p qui permet d'évaluer la formule à $\top$.

Exemple 8. Une formule de logique propositionnelle peut contenir plus de deux variables et un agencement des opérateurs. La formule suivante en est un exemple : $(\neg p \vee q) \wedge (p \vee \neg q) \wedge \neg r$. Cette formule est satisfaite si p et q prennent la même valeur et que r est à $\bot$.

Un modèle de satisfaisabilité est donc la représentation d'un problème à l'aide d'une formule de logique propositionnelle. Le modèle est satisfiable si la formule est satisfiable. Sinon, il n'existe pas de solution au modèle.

D'autres opérations logiques existent, mais peuvent toujours être représentées par un mélange de négations, disjonctions et conjonctions. Nous introduisons quelques opérations logiques supplémentaires dans les définitions 7, 8 et 9 ainsi que certaines équivalences.

Définition 7 (Implication logique)**.** L'implication logique est une opération entre deux variables logiques de la forme *si p alors q* pour deux variables p et q. Ceci est équivalent à *si p est vrai alors q ne peut pas être faux*. Cette opération est dénotée $\implies$ tel que $p \implies q$ et est équivalent à $\neg q \implies \neg p$ et à $\neg p \vee q$.

Définition 8 (Disjonction exclusive)**.** La disjonction exclusive est une opération entre deux variables logiques qui produit la valeur $\top$ si et seulement si exactement l'un des deux opérandes a la valeur $\top$. Cette opération est dénotée $\oplus$ tel que $p \oplus q$ pour deux variables logiques p et q. $p \oplus q$ est équivalent à $(\neg p \vee \neg q) \wedge (p \vee q)$.

Définition 9 (Équivalence logique)**.** L'équivalence logique est une opération entre deux variables logiques qui produit la valeur $\top$ si et seulement si les deux opérandes ont les mêmes valeurs. Cette opération est dénotée $\iff$ tel que $p \iff q$ pour deux variables logiques p et q. $p \iff q$ est équivalent à $\neg(p \veebar q)$, à $(p \implies q) \wedge (q \implies p)$ et à $(\neg p \vee q) \wedge (p \vee \neg q)$.

1.2.2 Forme normale conjonctive

Une formule de logique propositionnelle peut être reformulée de plusieurs manières différentes et rester équivalente. Cette section introduit une normalisation des formules appelée *forme normale conjonctive*, ou *conjunctive normal form* (CNF) en anglais. Il existe plusieurs autres normalisations, comme la forme normale disjonctive (DNF) [30], la forme A-normale [61] ou la forme normale de négation et ses variations [27, 28].

Une formule de logique propositionnelle sous la forme CNF est exprimée à partir de *littéraux*. Un littéral est soit un atome booléen, appelé *littéral positif*, soit la négation de cet atome, appelée *littéral négatif*. Un littéral seul ou une disjonction de littéraux est appelé une *clause*. Une formule est en forme normale conjonctive si elle est représentée par une clause unique ou une conjonction de clauses.

Exemple 9. La formule $(\neg p \vee q) \wedge (p \vee \neg q) \wedge \neg r$ est en forme normale conjonctive pour trois variables logiques p, q et r. Cette formule est composée de trois clauses mises en conjonction, soient $(\neg p) \vee q$, $p \vee \neg q$ et $\neg r$. Chaque clause ne contient que des opérations de disjonction ou de négation.

Exemple 10. La formule $(p \wedge q \wedge \neg r) \vee (\neg p \wedge \neg q \wedge \neg r)$ n'est pas en forme normale conjonctive. Cette formule est composée de littéraux qui sont joints par des conjonctions plutôt que des disjonctions. Cette formule est équivalente à la formule de l'exemple 9 sans pour autant utiliser les mêmes agencements de littéraux et d'opérateurs.

Parmi les logiciels qui travaillent avec des problèmes en forme normale conjonctive, plusieurs utilisent un format de fichier proposé par DIMACS [45]. Le format CNF proposé par DIMACS a pour objectif d'encoder des problèmes pouvant être lus par différents solveurs SAT et consiste à écrire de manière simplifiée les clauses dans un fichier texte. Un fichier DIMACS commence par une ligne d'entête de la forme p cnf <variables> <clauses> où <variables> et <clauses>

sont remplacés par des nombres entiers indiquant le nombre de variables et de clauses dans la formule [42]. Chaque ligne suivant l'entête représente une clause. Une clause est écrite sous la forme d'une séquence de nombres entiers séparés par des espaces. Chaque variable logique est associée a un nombre entier positif. Si la variable apparait dans un littéral négatif, alors le nombre associé doit être inscrit avec un symbole de négation. Pour terminer l'écriture d'une clause, un 0 doit être inscrit avant de changer de ligne. L'exemple 11 démontre comment écrire un problème CNF simple sous le format DIMACS.

Exemple 11. Considérons le problème $(x \vee y \vee \neg z) \wedge (\neg x \vee \neg z) \wedge (\neg y \vee z) \wedge (x \vee z)$. Cette formule est en forme normale conjonctive avec les clauses $C_1 := x \vee y \vee \neg z$, $C_2 := \neg x \vee \neg z$, $C_3 := \neg y \vee z$ et $C_4 := x \vee z$. Une représentation sous le format DIMACS est montrée au code 1.4. Pour ce code, la variable x est représentée par le nombre 1, la variable y par le nombre 2 et la variable z par le nombre 3.

```
p cnf 3 4
1 2 -3 0
-1 -3 0
-2 3 0
1 3 0

```

Extrait de fichier 1.4 – Exemple de problème CNF en format DIMACS

1.2.3 Encodage direct

Pour représenter un CSP à l'aide d'une modélisation SAT, un travail de traduction doit être effectué. Plusieurs techniques existent pour représenter un problème en nombres entiers avec des variables binaires. Une première approche, appelée *encodage direct*, consiste à introduire une variable logique x_{ij} à chaque valeur j qui peut être affectée à la variable entière X_i [72]. L'intuition derrière cet encodage est d'assigner la variable entière X_i à j si et seulement si x_{ij} est assigné à $\top$. Pour forcer ce comportement, il faut introduire des clauses sur les variables x_{ij}.

Les premières clauses servent à s'assurer qu'il existe une variable x_{ij} qui soit assignée à $\top$ pour chaque variable X_i. Pour chaque variable X_i, il faut ajouter la clause $\bigvee_{d \in \mathrm{dom}(X_i)} x_{ij}$ [72]. Ces clauses respectent la forme normale conjonctive.

D'autres clauses doivent être ajoutées pour garantir qu'au plus une variable logique soit mise à $\top$ pour chaque variable entière X_i. Plus spécifiquement, pour toutes variables entières X_i, les clauses $\neg x_{ij} \vee \neg x_{ik}$ sont ajoutées pour chaque j et k dans l'ensemble $\mathrm{dom}(X_i)$ tels que $j \neq k$ [72].

Additionnellement, il faut ajouter des clauses encodant les contraintes du problème. Un exemple concret est démontré dans l'exemple 12.

Exemple 12. Considérons la modélisation de l'exemple 1 où trois variables X_0, X_1 et X_2 sont sujettes à la contrainte AllDifferent(X_0, X_1, X_2) avec les domaines $\text{dom}(X_0) = \text{dom}(X_1) = \text{dom}(X_2) = \{1, 2, 3\}$. Pour chaque variable entière X_i, trois variables logiques sont crées, une pour chaque valeur possible des domaines. Un total de neuf variables logiques sont utilisées. Le tableau 1.2 montre une représentation pour ces variables logiques.

	$\text{dom}(X_i)$		
	1	2	3
X_0	x_{00}	x_{01}	x_{02}
X_1	x_{10}	x_{11}	x_{12}
X_2	x_{20}	x_{21}	x_{22}

Table 1.2 – Représentation des variables logiques pour l'exemple 12.

En plus des clauses de base nécessaires pour ce type d'encodage, la contrainte AllDifferent doit être représentée à l'aide de clauses indiquant que les variables entières doivent prendre des valeurs différentes. Plusieurs techniques existent pour cela [12, 36]. En programmation par contraintes, la contrainte AllDifferent peut être décomposée avec des inégalités entre chaque paire de variables entières. Le même procédé peut être reporté ici. Pour chaque paire de variables entières X_i et X_j et pour chaque valeur $k \in \text{dom}(X_i) \cap \text{dom}(X_j)$, on crée la relation $x_{ik} \implies \neg x_{jk}$, car si x_{ik} est vrai, alors $X_i = k$ et on veut éviter que $X_j = k$, ce qui est représenté par la variable x_{jk}. En forme CNF, $x_{ik} \implies \neg x_{jk}$ est équivalent à $\neg x_{ik} \vee \neg x_{jk}$.

Le code 1.5 montre la formule CNF correspondante à ce problème ainsi que l'équivalent en format DIMACS au code 1.6.

```
1   x00 ∨ x01 ∨ x02 ∧
2   x10 ∨ x11 ∨ x12 ∧
3   x20 ∨ x21 ∨ x22 ∧
4   ¬x00 ∨ ¬x01 ∧
5   ¬x00 ∨ ¬x02 ∧
6   ¬x01 ∨ ¬x02 ∧
7   ¬x10 ∨ ¬x11 ∧
8   ¬x10 ∨ ¬x12 ∧
9   ¬x11 ∨ ¬x12 ∧
10  ¬x20 ∨ ¬x21 ∧
11  ¬x20 ∨ ¬x22 ∧
12  ¬x21 ∨ ¬x22 ∧
13  ¬x00 ∨ ¬x10 ∧
14  ¬x00 ∨ ¬x20 ∧
15  ¬x10 ∨ ¬x20 ∧
16  ¬x01 ∨ ¬x11 ∧
17  ¬x01 ∨ ¬x21 ∧
18  ¬x11 ∨ ¬x21 ∧
19  ¬x02 ∨ ¬x12 ∧
20  ¬x02 ∨ ¬x22 ∧
21  ¬x12 ∨ ¬x22
22
```

Extrait de fichier 1.5 – La contrainte AllDifferent en formule CNF

⟹

```
1   p cnf 9 21
2   1 2 3 0
3   4 5 6 0
4   7 8 9 0
5   -1 -2 0
6   -1 -3 0
7   -2 -3 0
8   -4 -5 0
9   -4 -6 0
10  -5 -6 0
11  -7 -8 0
12  -7 -9 0
13  -8 -9 0
14  -1 -4 0
15  -1 -7 0
16  -4 -7 0
17  -2 -5 0
18  -2 -8 0
19  -5 -8 0
20  -3 -6 0
21  -3 -9 0
22  -6 -9 0
23
```

Extrait de fichier 1.6 – La contrainte AllDifferent en format DIMACS

1.2.4 Encodage logarithmique

Une autre manière d'encoder une variable entière avec des variables binaires est de générer un nombre logarithmique de variables logiques pour représenter les différentes valeurs possibles du domaine. Plus précisément, pour une variable entière X_i dont le domaine est $\text{dom}(X_i) = \{0, \ldots, m-1\}$, on peut représenter les différentes valeurs possibles du domaine avec $\lceil \log_2(m) \rceil$ variables logiques en utilisant une représentation binaire [58, 72]. Nous faisons ceci en créant les variables x_{ij} où $j \in \{0, \ldots, \lceil \log_2(m) \rceil - 1\}$. La séquence de variables $x_{i\lceil \log_2(m) \rceil - 1}, x_{i\lceil \log_2(m) \rceil - 2}, \ldots, x_{i0}$ est l'encodage binaire de l'entier X_i tel que représenté par l'équation 1.1.

$$X_i = \sum_{j=0}^{\lceil \log_2(m) \rceil - 1} 2^j x_{ij} \tag{1.1}$$

Pour une variable X_i, $\lceil \log_2(m) \rceil$ variables binaires sont utilisées. Ce type d'encodage permet donc d'attribuer une valeur parmi $2^{\lceil \log_2(m) \rceil}$ choix. Or, si un domaine contient m valeurs possibles dans son domaine, mais que m n'est pas une puissance de deux, il faut exclure certaines valeurs qui seraient possibles et qui ne font pas partie du domaine. Par exemple, pour une variable X_1 dont le domaine est $\{0, 1, 2\}$, nous aurions besoin de deux variables binaires. Or, ces deux variables binaires encodent 4 valeurs différentes, donc il faut retirer la valeur qui n'appartient pas au domaine à l'aide d'une clause. Dans ce cas-ci, si on veut ajouter une clause pour éviter que X_1 ne prenne pas la valeur 3, on ajoute une clause qui empêche le

représentation binaire de 3. Ainsi, la clause $\neg(x_{10} \wedge x_{11})$ doit être ajoutée, ou $\neg x_{10} \vee \neg x_{11}$ en forme normale de conjonction.

Dans le cas de l'encodage direct, pour chaque variable binaire, il est nécessaire d'ajouter des clauses qui forcent une variable à être associée à une seule valeur de son domaine. Or, dans le cas de l'encodage logarithmique, ces clauses n'existent pas, car il est impossible d'attribuer plus d'une valeur à une variable représentée par un tel encodage. N'importe qu'elle assignation valide des variables logiques est équivalente à une valeur unique du domaine, et ce même si les variables logiques sont toutes assignées à $\bot$ [72]. Un exemple concret est démontré dans l'exemple 13.

Exemple 13. Considérons la modélisation de l'exemple 1 où trois variables X_0, X_1 et X_2 sont sujettes à la contrainte AllDifferent(X_0, X_1, X_2) avec les domaines $\text{dom}(X_0) = \text{dom}(X_1) = \text{dom}(X_2) = \{1, 2, 3\}$. Pour chaque variable entière X_i, deux variables logiques sont créées. Un total de six variables logiques sont utilisées. Le tableau 1.3 montre une représentation pour ces variables logiques.

	Variables logiques	
X_0	x_{00}	x_{01}
X_1	x_{10}	x_{11}
X_2	x_{20}	x_{21}

Table 1.3 – Représentation des variables logiques pour l'exemple 13.

Comme il y a 3 valeurs dans les domaines, il faut utiliser deux variables binaires. Ces deux variables binaires peuvent donc mener à 4 valeurs différentes. Il faut donc ajouter des clauses de base pour éviter d'assigner accidentellement la quatrième valeur possible. Il faut donc ajouter les clauses $\neg x_{00} \vee \neg x_{01}$, $\neg x_{10} \vee \neg x_{11}$ et $\neg x_{20} \vee \neg x_{21}$.

Il faut ensuite ajouter les clauses qui caractérisent la contrainte AllDifferent. De la même manière qu'à l'exemple 12, pour chaque paire de variables entières X_i et X_j, et pour chaque valeur $a \in \text{dom}(X_i) \cap \text{dom}(X_j)$, on crée la relation $X_i = a \implies \neg X_j = a$ en utilisant les variables logiques appropriées. Comme un encodage logarithmique est utilisé, la relation entre les variables X_i et a n'est pas triviale.

De manière générale, nous voulons encoder $X_i = a \implies X_j \neq a$ avec $a_{\lceil \log_2(m) \rceil - 1}, \ldots, a_0$ l'encodage binaire de l'entier a. Notons que $a_k \oplus x_{ik}$ est équivalent à $\neg x_{ik}$ si a_k est 1 et équivalent à x_{ik} si a_k est 0. Dire que $X_i = a$ est équivalent à dire que $\bigwedge_{k=0}^{\lceil \log_2(m) \rceil - 1} \neg(a_k \oplus x_{ik})$, car pour chaque a_k qui compose a, $\neg(a_k \oplus x_{ik})$ force a_k et x_{ik} à prendre la même valeur, ce qui force donc l'ensemble des x_{ik} à prendre les valeurs qui encodent a. Nous avons donc

$\bigwedge_{k=0}^{\lceil \log_2(m) \rceil - 1} \neg(a_k \oplus x_{ik}) \implies \bigvee_{k=0}^{\lceil \log_2(m) \rceil - 1}(a_k \oplus x_{jk})$ ce qui se traduit en forme normale par $\bigvee_{k=0}^{\lceil \log_2(m) \rceil - 1}(a_k \oplus x_{ik}) \vee \bigvee_{k=0}^{\lceil \log_2(m) \rceil - 1}(a_k \oplus x_{jk})$. En ajoutant des conjonctions pour chaque variable X_i, on obtient l'équation 1.2. Dans l'exemple courant, les lignes 5 à 13 représentent cette équation.

$$\bigwedge_{i=0}^{|X|-1} \bigwedge_{j=i+1}^{|X|} \left(\bigvee_{k=0}^{\lceil \log_2(m) \rceil - 1} (a_k \oplus x_{ik}) \vee \bigvee_{k=0}^{\lceil \log_2(m) \rceil - 1} (a_k \oplus x_{jk}) \right) \tag{1.2}$$

Le tableau 1.4 montre une assignation valide de variables logiques. Sur ce tableau, X_1 est assigné à 0, X_2 est assigné à 2 et X_3 est assigné à 1, ce qui respecte la contrainte AllDifferent.

	Variables logiques	
X_1	$x_{00} = \bot$	$x_{01} = \bot$
X_2	$x_{10} = \bot$	$x_{11} = \top$
X_3	$x_{20} = \top$	$x_{21} = \bot$

Table 1.4 – Exemple d'assignation des variables logiques pour l'exemple 13.

Le code 1.7 montre la formule CNF correspondante à ce problème ainsi que l'équivalent en format DIMACS au code 1.8.

```
1
2     ¬x11 ∨ ¬x12 ∧
3     ¬x21 ∨ ¬x22 ∧
4     ¬x31 ∨ ¬x32 ∧
5     x11 ∨ x12 ∨ x21 ∨ x22 ∧
6     x11 ∨ x12 ∨ x31 ∨ x32 ∧
7     ¬x11 ∨ x12 ∨ ¬x21 ∨ x22 ∧
8     ¬x11 ∨ x12 ∨ ¬x31 ∨ x32 ∧
9     x11 ∨ ¬x12 ∨ x21 ∨ ¬x22 ∧
10    x11 ∨ ¬x12 ∨ x31 ∨ ¬x32 ∧
11    x21 ∨ x22 ∨ x31 ∨ x32 ∧
12    ¬x21 ∨ x22 ∨ ¬x31 ∨ x32 ∧
13    x21 ∨ ¬x22 ∨ x31 ∨ ¬x32
14
```

Extrait de fichier 1.7 – La contrainte AllDifferent en formule CNF

⟹

```
1     p cnf 6 12
2     -1 -2 0
3     -3 -4 0
4     -5 -6 0
5     1 2 3 4 0
6     1 2 5 6 0
7     -1 2 -3 4 0
8     -1 2 -5 6 0
9     1 -2 3 -4 0
10    1 -2 5 -6 0
11    3 4 5 6 0
12    -3 4 -5 6 0
13    3 -4 5 -6 0
14
```

Extrait de fichier 1.8 – La contrainte AllDifferent en format DIMACS

1.2.5 MiniSat (NaPS)

Il peut être difficile de modéliser correctement un CSP avec des variables binaires comme il faut prévoir un encodage pour chaque contrainte à utiliser. Dans une certaine mesure, il existe des outils qui peuvent prendre en entrée un modèle CSP et produire un modèle SAT. Picat [74] possède une manière de faire ce type de traduction en traduisant des fichiers FlatZinc (un type de fichier découlant de MiniZinc [53]) en fichiers CNF.

Pour certains CSPs spécifiques, une autre méthode peut être utilisée. MiniSat est un solveur SAT minimaliste développé pour aider les chercheurs et les développeurs se familiariser avec SAT [31]. MiniSat inclut un module appelé NaPS (*Nagoya pseudo-Boolean solver*), un solveur *pseudo-booléen* pour les problèmes linéaires avec des variables booléennes [32]. Si un CSP peut être représenté par une conjonction de contraintes pseudo-booléennes, NaPS a la capacité de traduire ces formules en une modélisation SAT. La définition 10 décrit formellement les contraintes pseudo-booléennes.

Définition 10 (Contrainte pseudo-booléenne)**.** « Une contrainte pseudo-booléenne est une inégalité $C_0p_0 + C_1p_1 + \ldots + C_{n-1}p_{n-1} \geqslant C_n$, où, pour tout i, p_i est un littéral et C_i un coefficient entier. » [32]

NaPS peut traduire automatiquement une modélisation faite à l'aide de contraintes pseudo-booléennes en un modèle SAT en fichier CNF qui respecte le format DIMACS tel qu'introduit à la section 1.2.2. De plus, si NaPS doit introduire des variables booléennes supplémentaires pour la traduction, il est possible de récupérer le nom de toutes les variables originales, ce qui n'est pas possible avec Picat au moment de l'écriture de ce mémoire.

Exemple 14. Considérons l'encodage direct de l'exemple 12 où neuf variables logiques sont utilisées pour respecter la contrainte AllDifferent(X_1, X_2, X_3) avec $\text{dom}(X_1) = \text{dom}(X_2) = \text{dom}(X_3) = \{1, 2, 3\}$. En contraintes pseudo-booléennes, les équations suivantes respectent le problème à encoder.

$$
\begin{aligned}
1 \cdot x_{11} + 1 \cdot x_{12} + 1 \cdot x_{13} &= 1; \\
1 \cdot x_{21} + 1 \cdot x_{22} + 1 \cdot x_{23} &= 1; \\
1 \cdot x_{31} + 1 \cdot x_{32} + 1 \cdot x_{33} &= 1; \\
-1 \cdot x_{11} - 1 \cdot x_{21} - 1 \cdot x_{31} &\geqslant -1; \\
-2 \cdot x_{12} - 2 \cdot x_{22} - 2 \cdot x_{32} &\geqslant -2; \\
-3 \cdot x_{13} - 3 \cdot x_{23} - 3 \cdot x_{33} &\geqslant -3;
\end{aligned}
$$

NaPS a la capacité de transformer ce problème en modèle SAT sous le format CNF.

1.3 Dénombrement propositionnel

1.3.1 Définition et concepts

Étant donné une formule propositionnelle telle qu'introduite à la définition 6, le principal problème à résoudre est habituellement de trouver une solution qui satisfait les clauses. Trouver une assignation qui satisfait une formule propositionnelle est un problème qui appartient à une classe de complexité appelée **NP**, c'est-à-dire les problèmes qui s'écrivent de la forme « étant

donné x, existe-t-il un y tel que $(x, y) \in Q$ » où Q est une relation mathématique qui peut être calculée en temps polynomial. Certains problèmes qui appartiennent à **NP** sont définis plus difficiles que d'autres et sont alors dits **NP**-*complet*. Cook et Levin [23, 48] ont démontré que SAT est **NP**-*complet*. 3SAT, une version du problème SAT où les formules à résoudre ne peuvent pas contenir plus de 3 variables par clause en forme CNF, est considéré comme étant le problème canonique **NP**-*complet*.

Une assignation valide de valeurs qui satisfont une formule propositionnelle s'appelle un *modèle propositionel* [70]. Une question intéressante est de se demander combien de solutions distinctes, ou de modèles propositionnels uniques, existent qui satisfont une formule considérée. Il s'agit d'un problème de *dénombrement*.

> **Définition 11** (Dénombrement propositionnel)**.** « Le dénombrement de modèles propositionnels, ou #SAT, est le problème du calcul du nombre de modèles propositionnels pour une formule propositionnelle donnée, c'est-à-dire le nombre d'assignations distinctes des variables pour lesquelles la formule est évaluée comme $\top$. Ce problème est également appelé problème de dénombrement de solutions pour SAT. » [39]

Le dénombrement de solutions pour SAT est un problème particulièrement compliqué qui appartient à la classe de complexité #**P**. Cette classe de complexité peut être décrite comme la version d'un problème dans **NP**, par exemple SAT, où on s'intéresse à compter le nombre de solutions. En fait, dénombrer le nombre de modèles propositionnels d'une formule propositionnelle est une généralisation du problème SAT en plus d'être le problème canonique #**P**-*complet* [39]. Une définition formelle de #**P** est donnée à la définition 12.

> **Définition 12** (Classe de complexité #**P**)**.** « Étant donné une relation mathématique pouvant être calculée en temps polynomial Q, le problème de dénombrement correspondant à un problème dans **NP** demande : étant donné une entrée x, combien y a-t-il de y tels que $(x, y) \in Q$? La classe de complexité #**P** comprend tous les problèmes de dénombrement associés à de telles relations mathématiques pouvant être calculées en temps polynomial. » [39]

Suivant la définition 12, si Q est la relation « y est une affectation qui satisfait la formule propositionnelle x », alors le problème de dénombrement pour Q est le problème de dénombrement du modèle propositionnel, #SAT. De plus, notons que chaque problème dans #**P** a une version connexe dans **NP**, i.e. les problèmes qui s'écrivent de la forme « étant donné x, existe-t-il un y tel que $(x, y) \in Q$ » [39] ?

Les meilleurs algorithmes connus pour trouver une assignation valide au problème SAT sont en temps exponentiel en pire cas [39]. Le problème de dénombrement consiste donc à compter le nombre de solutions d'un problème aux algorithmes exponentiels. De plus, puisque les

problèmes dans **NP** peuvent être résolus avec un espace mémoire polynomial [64], il en découle que les problèmes dans #**P** peuvent également être résolus avec un espace mémoire polynomial malgré le nombre exponentiel de sous-problèmes [39].

Plusieurs approches existent pour dénombrer SAT. Il existe le *dénombrement exact* et le *dénombrement approximatif*. Nous présentons dans les sections 1.3.2 et 1.3.3 des approches pour le dénombrement exact, dans la section 1.3.4 des approches pour le dénombrement approximatif et dans la section 1.3.5 des approches pour le dénombrement de CSP.

1.3.2 Dénombrement exact

Un dénombreur exact est un outil qui permet de retourner exactement le bon nombre de solutions pour un modèle SAT donné en entrée. Une manière triviale de calculer exactement le nombre de solutions d'un problème à n variables logiques serait de tester chacune des 2^n assignations possibles aux variables. Or, cet algorithme ne peut pas être utilisé en pratique puisque 2^n croît trop rapidement, même pour les ordinateurs modernes.

Parmi les approches ayant des implémentations pratiques, deux se démarquent. La première est l'ensemble des algorithmes qui se basent sur la fouille dans un arbre de recherche auquel on ajoute le filtrage de clauses, tel qu'introduit à la section 1.1.4. Dans le cas des arbres de recherche dont les variables ont des domaines binaires, l'algorithme consiste à choisir un littéral et lui affecter une une valeur entre $\top$ et $\bot$. Selon le littéral choisi, la formule CNF est simplifiée. Le processus se répète ensuite sur la nouvelle formule. Si la formule simplifiée est satisfiable, alors la formule originale l'est aussi. Si la formule simplifiée n'est pas satisfiable, alors le littéral est assigné à sa valeur contraire et les mêmes vérifications sont effectuées [29]. L'algorithme de dénombrement associé à cette technique consiste à déterminer quelles branches de l'espace de recherche sont satisfiables. Pour un problème à n variables, si un noeud est reconnu comme étant satisfiable avec t variables fixées, alors on compte 2^{n-h} solutions pour cette branche. Cependant, au lieu d'arrêter le déroulement lorsqu'une branche est considérée satisfiable, pour dénombrer il faut que toutes les branches de l'arbre de recherche soient explorées. Une fois l'exploration de l'arbre de recherche terminée, la somme des nombres de solutions associés aux branches donne le compte exact pour le nombre de solutions de la formule CNF. Le dénombreur exact `CDP` utilise cette technique [17].

`CDP` est un des premiers dénombreurs à utiliser une technique de fouille dans les espaces de solutions [17]. Des travaux subséquents ont été faits pour améliorer l'efficacité de l'approche. Relsat [6, 7] étend l'idée exploitée par `CDP` en représentant un CNF sous la forme d'un graphe et en analysant les composantes disjointes de la formule SAT et en calculant le nombre de solutions pour chaque composante [39]. La technique utilisée par `CDP` peut aussi être étendue par l'utilisation d'un cache, comme le montre `Cachet` [62], un dénombreur exact bâti sur le solveur SAT `zChaff` [52]. Le principe consiste à exploiter le fait que plusieurs sous-formules

peuvent être demandées à répétition lors des simplifications de clauses. En entrant dans un cache les résultats pour ces sous-formules, on peut réutiliser les résultats lorsqu'on les rencontre une deuxième fois. `Cachet` se base entre autres sur le travail de Bacchus et al. [5] pour l'exploitation du cache [39].

Un des problèmes engendrés par la création d'un cache est l'espace mémoire requis. Pour être efficace, le cache doit être placé en mémoire vive. Chaque formule placée requiert un espace non négligeable qui peut rapidement devenir trop grand pour des formules complexes. Si les éléments enregistrés en cache prenaient moins de place, il serait possible de placer davantage d'information en mémoire vive et donc améliorer la vitesse d'exécution des algorithmes. `SharpSAT` [67] est un outil qui utilise de telles techniques pour réduire l'espace requis. Par exemple, l'une des techniques consiste à ne placer en cache que les indices des variables dans une sous-formule et les indices des clauses originales qui appartiennent à la sous-formule plutôt que de placer des clauses complètes ou les conflits appris. Cette idée réduit l'espace mémoire d'une formule de manière importante, ce qui améliore grandement la vitesse des résultats pour le dénombrement [39].

Une approche différente pour le dénombrement exact consiste à convertir ou à compiler les formules CNF en une autre forme logique plus facile à dénombrer. Un exemple de ce genre de transformation est la conversion d'une formule CNF en un diagramme de décision booléen (BDD) [19]. Une fois la conversion en BDD effectuée, le nombre de solutions peut être calculé en temps polynomial par rapport à la taille du BDD qui, lui, peut être de taille exponentielle par rapport aux nombres de variables dans le problème SAT [39]. Le compilateur c2d [26] exploite ce concept en traduisant les formules CNF en formules logiques équivalentes à un BDD à une réduction linéaire près. Le nombre de solutions peut donc être rapidement calculé une fois la transformation effectuée.

1.3.3 Dénombrement exact probabiliste

Les méthodes de dénombrement exact résolvent un problème #P-complet typiquement en explorant de manière exhaustive l'espace de recherche combinatoire. Comme l'espace de recherche croît exponentiellement selon le nombre de variables, le temps d'exécution de tels algorithmes augmente drastiquement avec le nombre de variables. Certaines approches pallient ce problème en sacrifiant la précision sur le nombre de solutions au profit d'un temps d'exécution plus avantageux. La définition 13 décrit formellement le dénombrement exact probabiliste.

> **Définition 13** (Dénombrement exact probabiliste)**.** Soit un modèle F et $\delta \in (0, 1]$, un dénombrement exact probabiliste estime le nombre de solutions $\#F$ et garantit que la valeur estimée est égale à $\#F$ avec une probabilité d'au moins $1 - \delta$.

GANAK [63] est un dénombreur exact probabiliste très performant. GANAK est un outil qui exploite l'utilisation d'un cache efficace pour estimer le nombre de solutions. Le cache utilisé se base sur les mêmes principes que SharpSAT [67], mais en ajoutant une composante probabiliste pour améliorer davantage l'efficacité de l'espace mémoire requis pour les opérations.

Plus spécifiquement, GANAK utilise une famille de fonctions de hachage ϵ-universelles. Une famille de fonctions de hachage ϵ-universelles est une famille de fonctions $H(n,m)$ qui font correspondre $\{0,1\}^n$ à $\{0,1\}^m$ telles que pour deux valeurs y_1 et $y_2 \in \{0,1\}^n$ et pour une fonction $h \in H(n,m)$, nous avons $\mathrm{P}\left[h(y_1) = h(y_2)\right] \leqslant \epsilon$. GANAK utilise ces fonctions de hachage pour mettre en cache les sous-formules, ce qui permet de réduire l'espace nécessaire du stockage des formules logiques, augmentant ainsi la capacité de cache effective [63]. L'utilisation de ces fonctions a pour conséquence qu'il existe un risque, borné par une probabilité ϵ, que des collisions surviennent en cache, ce qui mènerait à un dénombrement incorrect. Ainsi, la probabilité que GANAK retourne le bon nombre de solutions est d'au moins $1 - \epsilon$.

Empiriquement, GANAK performe mieux que plusieurs dénombreurs exacts et retourne habituellement un compte exact. Dans leurs expérimentations, Sharma et al. rapportent un compte exact pour toutes les instances [63], malgré la nature probabiliste de l'algorithme.

1.3.4 Dénombreurs approximatifs

Selon l'application, il peut être considéré irréaliste d'exiger un dénombrement exact. Entre autres, si un problème s'intéresse seulement à l'ordre de grandeur du nombre de solutions, il n'est probablement pas nécessaire de connaître le compte exact. Les dénombreurs approximatifs retournent une approximation du nombre de solutions, habituellement avec des garanties. Ces algorithmes offrent moins de précision, mais sont généralement beaucoup plus rapides que ceux utilisés pour le dénombrement exact.

Plusieurs dénombreurs utilisent le même principe pour approximer le nombre de solutions, mais avec des garanties variables. Le concept consiste à diviser l'espace de solutions du problème à dénombrer en un nombre connu de sections ayant un nombre environ équivalent de solutions. Avec certaines conditions propres à chaque algorithme, ces sections sont dénombrées, puis le compte est extrapolé à l'ensemble du problème.

ApproxCount [73] est un algorithme qui utilise des chaînes de Markov Monte-Carlo [49] et exploite le fait que les échantillons sont presque uniformes, ce qui permet de calculer une bonne estimation du nombre de solutions. Cette approche offre des performances rapides et des comptes près des vraies valeurs, mais sans garantie. SampleCount [37] est un autre dénombreur basé sur le même principe que ApproxCount, mais offrant une garantie d'exactitude sur une borne inférieure sur le nombre de solutions total et en conservant la même qualité d'échantillonnage qu'ApproxCount.

Un autre dénombreur appelé `MBound` [38] utilise une technique différente pour approximer le nombre de solutions. Au lieu d'utiliser les chaînes de Markov, `MBound` ajoute des contraintes aléatoires sur le modèle à dénombrer pour réduire le nombre de solutions et ensuite extrapoler les approximations au modèle original. Si les contraintes ajoutées respectent certaines conditions précises, des garanties sur une borne inférieure et supérieure peuvent être obtenues sur le nombre de solutions.

Un dénombreur qui mérite un peu plus d'attention quant au dénombrement approximatif est `ApproxMC` [20]. Ce dénombreur offre un dénombrement très compétitif en termes de temps avec une confiance paramétrable. Pour un modèle F ayant $\#F$ solutions, des paramètres $\delta \in (0, 1]$ et $\epsilon > 0$, `ApproxMC` retourne un nombre de solutions garantit d'être dans l'intervalle $[\frac{\#F}{1+\epsilon}, (1+\epsilon)\#F]$ avec une probabilité d'au moins $1 - \delta$. `ApproxMC` fonctionne en utilisant des fonctions de hachage linéairement indépendantes pour partitionner aléatoirement l'ensemble des solutions d'une formule CNF en des cellules dites « de petite taille », c'est-à-dire des cellules contenant peu de solutions. Afin de tester si les cellules générées sont assez petites, une cellule aléatoire est choisie en vérifiant si elle est non vide et a au plus p éléments, où p est un seuil qui ne dépend que du paramètre ϵ. Si la cellule choisie contient trop de solutions, la formule CNF originale est alors partitionnée aléatoirement en un ensemble de cellules deux fois plus nombreuses qu'auparavant en choisissant une nouvelle fonction de hachage. Ce procédé est perpétué jusqu'à ce que les cellules soient assez petites, ce qui est déterminé par la valeur de p. Si la cellule est suffisamment petite, le nombre de solutions de la formule CNF est estimé en multipliant le nombre de solutions de la cellule par le nombre de cellules obtenues. `ApproxMC` recommence ensuite le processus $\lceil 35 \log_2 \frac{3}{\delta} \rceil$ fois, une valeur qui assure les garanties de performance indiquées ci-dessus. La médiane des nombres de solutions estimés est retournée comme valeur finale. `ApproxMC` a été améliorée à l'aide d'algorithmes spécifiques [21, 50, 65, 66] et la version finale d'`ApproxMC` performe mieux empiriquement que la plupart des autres dénombreurs approximatifs en plus d'avoir des garanties paramétrables.

1.3.5 Dénombrement sur modèles en nombres entiers

Les techniques et outils de dénombrement décrits précédemment s'appliquent aux formules logiques telles des équations en forme CNF. Or, des travaux récents exportent les principes utilisés pour le dénombrement de formules logiques à des modèles en nombres entiers pour la programmation par contrainte. Vavrille et al. sont parmi les premiers à s'attaquer à ce problème [71]. L'idée exploitée est d'ajouter au CSP des contraintes qui agissent en tant que fonctions de hachage. Plus spécifiquement, les auteurs ajoutent une contrainte $A\boldsymbol{x} = \boldsymbol{b} \pmod p$ où $\boldsymbol{x}$ est un vecteur de n variables entières ayant des domaines finis, A une matrice $m \times n$ dont les éléments appartiennent aléatoirement à $[p] := \{0, 1, \ldots, p-1\}$, $\boldsymbol{b}$ un vecteur de m éléments de $[p]$, et p un nombre premier [71]. Ces contraintes sont appelées des *égalités modulaires linéaires* et partitionnent uniformément les solutions avec une probabilité

$\mathrm{P}\left[A\boldsymbol{x} = \boldsymbol{b} \pmod{p}\right] = \frac{1}{p^m}$. Bien que Vavrille et al. ne font pas de dénombrement dans leur article, ils expliquent clairement que leur approche en est inspirée et peut y être étendue. Des travaux subséquents ont été effectués par Pesant et al. afin d'ajouter des contraintes de la forme $A\boldsymbol{x} \leqslant \boldsymbol{b} \pmod{p}$ et d'ajouter des garanties sur la qualité des échantillons [54]. Ces méthodes pourraient être des approches intéressantes pour faire du dénombrement puisque l'échantillonnage de solutions pourrait exploiter la puissance des solveurs de problèmes de programmation par contraintes.

1.3.6 Monotonie des contraintes

Comme les contraintes peuvent être représentées par des relations mathématiques entre des variables, il est naturel d'attribuer aux contraintes des propriétés mathématiques qui s'appliquent aux relations. L'une de ces propriétés utilise le principe de *monotonie* des fonctions, décrit par la définition 14.

Définition 14 (Fonction monotone)**.** « Une fonction réelle f est monotone sur un domaine D si elle est soit croissante sur D, donc $f(x_1) \leqslant f(x_2)$ chaque fois que $x_1 \leqslant x_2$, ou décroissante sur D, donc $f(x_1) \geqslant f(x_2)$ chaque fois que $x_1 < x_2$.

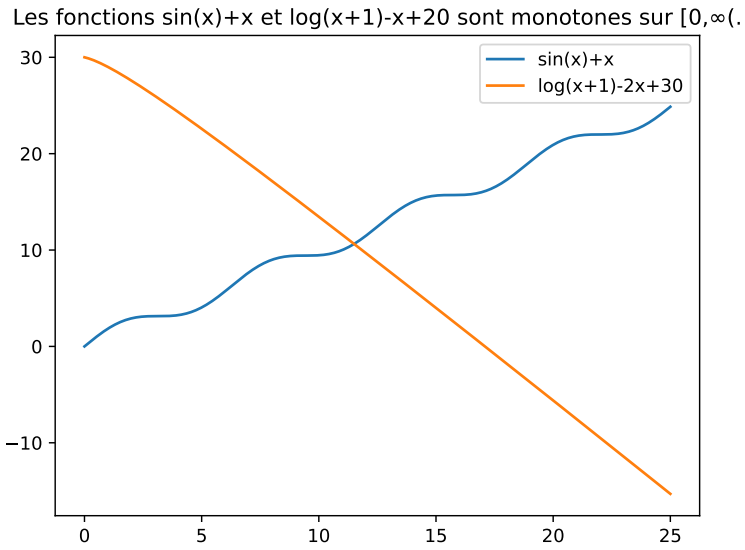

FIGURE 1.4 – Exemple de fonctions monotones.

Exemple 15. Suivant la définition 14, la fonction $\sin(x) + x$ est monotone sur le domaine $[0, \infty]$ puisque $\sin(x_1) + x_1 \leqslant \sin(x_2) + x_2$ lorsque $x_1 \leqslant x_2$. De manière équivalente, la fonction $\log_2(x + 1) - 2x + 30$ est monotone sur le domaine $[0, \infty]$

puisque $\log_2(x_1 + 1) - 2x_1 + 30 \geqslant \log_2(x_2 + 1) - 2x_2 + 30$ lorsque $x_1 < x_2$. La figure 1.4 représente visuellement les deux fonctions décrites.

Plusieurs contraintes possèdent un ensemble de paramètres qui permettent de réguler leur effet sur les variables de décision. Un exemple d'une contrainte ayant un tel paramètre est la contrainte AMONG [9].

Définition 15 (Contrainte AMONG)**.** La contrainte AMONG$([X_1, \ldots, X_n], l, u, V)$ s'assure que parmi les variables de décision $[X_1, \ldots, X_n]$, au moins l et au plus u variables de décision prennent des valeurs dans l'ensemble V.

La contrainte AMONG peut être dénombrée et les choix des paramètres l et u influencent directement le nombre de solutions. Par exemple, avec $\text{dom}(X_1) = \text{dom}(X_2) = \text{dom}(X_3) = \{0, 1\}$, AMONG$([X_1, X_2, X_3], 2, 2, \{1\})$ possède 3 solutions distinctes, alors que AMONG$([X_1, X_2, X_3], 1, 2, \{1\})$ en possède 6.

Une *fonction de probabilité* d'une contrainte prend en entrée les paramètres de la contrainte et retourne la proportion des affectations possibles qui satisfont la contrainte avec les paramètres données. Lorsque la fonction de probabilité d'une contrainte est monotone par rapport à un paramètre, il est possible de prévoir l'effet qu'aurait un changement de paramètre sur le nombre de solutions. Par exemple, Picard-Cantin et al. [56] ont déterminé que la contrainte AMONG possède une fonction de probabilité monotone en ce qui concerne les paramètres l et u. Ceci veut dire que si l est fixé, augmenter la valeur de u augmente nécessairement le nombre de solutions de la contrainte. Pareillement, si u est fixé, diminuer l augmente le nombre de solutions de la contrainte. Dans le cas de cette contrainte, la fonction de probabilité est monotone par rapport à l lorsque u est fixé.

La monotonie des contraintes est importante pour estimer une borne inférieure sur le nombre de solutions d'une contrainte lorsque les paramètres ne sont pas fixés. L'exemple 16 exprime un cas avec la contrainte AMONG où une telle borne est estimée.

Exemple 16. Considérons la contrainte AMONG$([X_1, X_2, X_3], l, u, V)$ avec $l = 1$, $u \in \{1, 2, 3\}$, $V = \{1\}$ et un domaine binaire pour chaque variable de décision. Si on veut calculer une borne inférieure sur le nombre de solutions de cette contrainte, au lieu de tester toutes les valeurs de u possibles, on peut exploiter la monotonie de la fonction de probabilité de AMONG et calculer le nombre de solutions lorsque u est à son minimum. On peut faire ceci puisque le nombre de solutions de AMONG est monotone croissant par rapport à u lorsque l est fixé [56]. Dans ce cas-ci, la borne inférieure serait donc de 3 solutions, c'est-à-dire $X_1 = 1, X_2 = 0, X_3 = 0$, $X_1 = 0, X_2 = 1, X_3 = 0$ et $X_1 = 0, X_2 = 0, X_3 = 1$. Donc, peu importe la valeur de u, on sait que la contrainte possède au moins 3

solutions. Si on prend $u = 2$, la contrainte accepte 6 solutions et elle en accepte 7 avec $u = 3$.

1.3.7 Dominance des contraintes

Dans le cas d'un contexte de dénombrement, on dit qu'une contrainte C_1 domine une contrainte C_2, noté $C_2 \prec C_1$, si C_1 accepte toutes les solutions de C_2. La dominance de contraintes peut être utile dans divers domaines, notamment l'acquisition de contraintes [11, 55], concept introduit à la section 1.4. La monotonie sur les contraintes, définie à la définition 14, a un lien direct avec la définition de la dominance de contraintes. En effet, une contrainte dont le nombre de solutions est monotone par rapport à un paramètre dominera ou sera nécessaire dominée par cette même contrainte avec une valeur de paramètre différent. L'exemple 17 donne un exemple concret d'une contrainte dont la monotonie sur le nombre de solutions mène à une dominance.

Exemple 17. Considérons la contrainte $\textsc{Among}([X_1, \ldots, X_5], l, u, V)$ avec $l = 1$, $u = 3$ et $V = \{1\}$. Diminuer la valeur du paramètre l à 0 ajoute des solutions à la contrainte sans en retirer puisque Among est monotone par rapport à l lorsque u est fixe [56]. Ainsi, la seconde contrainte dont $l = 0$ domine la première, car elle possède toutes les solutions de la première et même plus.

Une contrainte dominée (ou dominante) ne respecte pas nécessairement les relations de monotonie. L'exemple 18 est un cas concret où trois contraintes différentes présentent des relations de dominance.

Exemple 18. Considérons la contrainte $C_1 := \textsc{AllDifferent}(X_1, X_2, X_3, X_4, X_5)$, la contrainte $C_2 : X_1 \neq X_5$ et $C_3 : X_1 < X_2 < X_3 < X_4 < X_5$ avec $X_i \in \{1, \ldots, 5\}$. Aucune des contraintes ne possède de paramètres et ne peuvent donc pas être monotones par rapport à des paramètres. Pourtant, C_2 domine C_1 ($C_1 \prec C_2$) puisque toutes les solutions de C_1 sont forcément incluses dans l'espace de solution de C_2.

De plus, on a la relation de dominance $C_3 \prec C_1$, car les inégalités dans C_3 forcent à ce que les variables de décisions prennent des valeurs différentes. Il existe donc une solution unique à C_3 qui est incluse dans l'espace de solutions de C_1.

Finalement, on obtient les relations de dominances $C_3 \prec C_1 \prec C_2$.

1.4 Acquisition de contraintes

1.4.1 Définitions et concepts

Lorsque l'on travaille avec la programmation par contraintes, la modélisation consiste en une étape cruciale à la résolution du problème puisqu'elle sert a exprimer aussi efficacement que possible le problème à résoudre. Typiquement, un expert s'occupe de faire cette modélisation, principalement due au fait que des connaissances précises sur la programmation par contraintes et sur le problème à modéliser doivent être utilisées. L'*acquisition de contraintes* est une manière d'automatiser la modélisation d'un problème en se basant sur des exemples de solutions.

> **Définition 16** (Problème d'acquisition de contraintes et de satisfaction)**.** « Un problème d'acquisition de contraintes et de satisfaction, ou *constraint acquisition and satisfaction problem* en anglais (CASP), est un problème où le solveur doit obtenir de l'information à propos des contraintes avant de résoudre le problème de satisfaction de contraintes. » [33]

L'apprentissage de contraintes est fait depuis un ensemble de contraintes qui s'appellent le *biais*. En assumant que le biais est représentatif des contraintes qui peuvent représenter le problème, un sous-ensemble du biais forme le problème.

> **Définition 17** (Solution à un CASP)**.** « Une solution à un CASP P est une solution qui satisfait le CSP obtenu après avoir acquis les connaissances pour représenter P avec des contraintes provenant d'un biais. » [33]

Dans la littérature, il est fréquent de séparer la résolution d'un CASP en deux étapes distinctes, l'apprentissage des contraintes (apprendre le CSP) et résoudre le CSP obtenu. Typiquement, l'étape qui consiste à *obtenir de l'information* se base sur plusieurs critères qui incluent l'analyse d'exemples de solutions. L'idée générale est que ces exemples satisfont le problème que l'on veut apprendre et donc qu'ils encodent une partie d'information qui serait bénéfique au solveur. Deux types d'exemples existent, les *exemples positifs* et les *exemples négatifs*. Un exemple positif est un exemple de solution au CSP à apprendre tandis qu'un exemple négatif est un exemple de solution qui ne satisfait pas le CSP à apprendre.

Plusieurs méthodes existent pour analyser ces exemples de solutions, dont l'*acquisition passive* et l'*acquisition active*. Ces deux approches sont discutées respectivement aux sections 1.4.2 et 1.4.3.

1.4.2 Acquisition passive

Une première manière générale de procéder à l'acquisition de contraintes consiste à utiliser le principe d'*acquisition passive*.

Définition 18 (Acquisition passive)**.** « L'acquisition passive est une manière d'apprendre de l'information où le solveur qui fait l'acquisition reçoit et analyse de l'information sans avoir l'opportunité d'interagir avec la source de cette information. » [14, 51]

Plusieurs outils exploitent cette technique, comme le Model Seeker [11] qui analyse des exemples de solutions positifs et négatifs et cherche parmi un catalogue de contraintes globales [8] les contraintes qui représentent le mieux les exemples fournis. Ce catalogue représente le biais du solveur et la fouille est effectuée grâce au Constraint Seeker. Plus spécifiquement, une fois les exemples reçus, Model Seeker transforme les exemples pour générer des motifs qui représentent les contraintes possibles. Le Constraint Seeker cherche ensuite dans le catalogue et retourne une liste, potentiellement volumineuse, de candidats qui respectent les exemples. La liste retournée par Constraint Seeker est un classement des candidats où les candidats en haut de la liste sont considérés comme étant plus pertinents pour l'apprentissage du CSP. Le classement retourné est en fonction d'une combinaison de propriétés structurelles comme la présence d'arguments monotones, la présence d'implications entre des contraintes, la densité de solution estimée et la popularité estimée de la contrainte [10, 11]. Finalement, le classement est traité pour retirer les contraintes redondantes ou inutiles, puis les contraintes les plus hautes dans le classement sont suggérées pour représenter les exemples donnés. Model Seeker est efficace pour apprendre des conjonctions de contraintes populaires, mais nécessite des contraintes bien documentées incluses dans le catalogue et il est impossible de spécifier au solveur des contraintes qui seraient déjà connues.

Une deuxième approche passive a été proposée par Picard-Cantin et al. [55, 56] où les auteurs utilisent la technique de *séparation et évaluation* pour trouver les bons paramètres aux contraintes Sequence et SubsetFocus considérant une liste d'exemples positifs. Dans le cadre de cet article, au lieu d'apprendre quelles contraintes expliquent des exemples, les auteurs se demandent comment trouver efficacement les valeurs des paramètres si la contrainte est connue. Par exemple, il peut être connu qu'une contrainte linéaire ($a+b\cdot X \leqslant c$) appartiennent à un modèle, mais sans que l'on sache la valeur des coefficients a, b et c. Pour choisir les valeurs à associer aux paramètres, Picard-Cantin et al. proposent d'utiliser le nombre de solutions de la contrainte comme métrique de même que la probabilité qu'une affectation aléatoire satisfasse la contrainte, puis de choisir l'assignation de paramètres la plus probable qui respecte les exemples de solutions donnés. La probabilité d'une combinaison de paramètres est évaluée telle que la contrainte ayant le moins de solutions est considérée plus probable. L'idée est que si une contrainte `C1` possède un espace de solutions plus restrictif qu'une contrainte `C2` et qu'elles respectent tous les exemples de solutions, alors il est moins probable que `C1` respectent les exemples de solutions par hasard. Ceci est équivalent à dire qu'il est plus probable que `C1` explique les exemples de solutions. Picard-Cantin et al. utilisent des chaînes de Markov dans un algorithme de programmation dynamique pour calculer le nombre de solutions des

contraintes de même que la probabilité d'observer les différentes valeurs dans les exemples de solutions.

Conacq est un algorithme d'acquisition de contraintes qui existe en version active et passive [13]. La version passive, aussi appelée Conacq.1, est basée sur l'idée qu'il est possible de représenter l'espace d'apprentissage à l'aide de formules clausales, tout comme les problèmes de satisfaisabilités. Les techniques SAT peuvent donc être appliquées sur ses représentations pour traiter les problèmes de cohérence, ou de convergence. Pour fonctionner, Conacq.1 utilise en entrée un vocabulaire préfixé, un biais et une liste d'exemples (positifs ou négatifs) de solutions. Le vocabulaire et le biais forment les différents CSPs possibles. L'ensemble des CSPs qui satisfont tous les exemples donnés s'appelle *l'espace des versions*. Conacq.1 représente l'espace des versions à l'aide de clauses où la structure permet de vérifier la satisfaisabilité des modèles suggérés en temps polynomial. Plus spécifiquement, Conacq.1 représente l'espace des versions avec une *théorie duale de Horn*, c'est-à-dire par une conjonction de clauses qui contiennent au plus un littéral négatif. Bessiere et al. explorent plusieurs avantages de cette représentation exploitant les propriétés des théories duales de Horn et en s'inspirant des mêmes techniques qui sont utilisées pour résoudre les problèmes de satisfaisabilité. L'algorithme retourne une théorie duale de Horn, qui représente potentiellement plusieurs CSPs. Si le retour de Conacq.1 ne représente qu'un seul CSP, il est dit avoir convergé.

1.4.3 Acquisition active

Une autre technique pour procéder à l'acquisition de contraintes consiste à utiliser le principe d'*acquisition active*.

> **Définition 19** (Acquisition active)**.** « L'acquisition active est une manière d'apprendre de l'information qui consiste à interagir, par exemple à l'aide de questions, avec une source d'information et d'analyser rétroactivement le retour d'information obtenu durant ces interactions. » [14, 51]

Plusieurs outils approchent l'acquisition de contraintes de manière active. Cette technique permet d'interagir régulièrement avec un utilisateur pour obtenir une rétroaction sur les choix faits durant l'exécution de l'algorithme. Cet utilisateur peut être un expert qui analyse les décisions prises, un ordinateur qui fournit des exemples de solutions périodiquement ou n'importe quelle autre source d'information qui peut changer le procédé après le début de l'acquisition de contraintes.

Conacq offre une version alternée de l'algorithme décrit plus haut qui agit de manière active en posant des requêtes à un utilisateur [13]. La version active de Conacq s'appelle Conacq.2. Au fur et à mesure que Conacq.2 analyse l'espace des versions, une *question d'appartenance* est posée à l'utilisateur. Il s'agit de requêtes où Conacq.2 présente à l'utilisateur un exemple de solution et l'utilisateur doit indiquer si cet exemple est positif

ou négatif. Pour que l'approche active soit bénéfique, il faut que les requêtes permettent d'apprendre des informations qui ne sont pas présentes dans les exemples de solutions. Ainsi, pour chaque requête, CONACQ.2 tente de faire en sorte que la question d'appartenance ne soit pas redondante en information, peu importe si la réponse est positive ou négative. Un des problèmes soulevés par les auteurs est que trouver de telles demandes est un problème qui est NP-complet. Une requête dite *optimale* peut réduire l'espace des versions de moitié, indépendamment si la réponse à la requête est positive ou négative, donc il peut être d'intérêt pour cet algorithme de passer du temps à trouver une requête presque optimale. Par contre, le nombre de requêtes nécessaires peut être particulièrement élevé. Même de manière optimale, il faudrait plus de 1000 requêtes pour trouver le CSP qui représente un sudoku de dimensions 4×4.

CONACQ a servi de base pour plusieurs autres algorithmes, notamment QUACQ [14]. De nombreux auteurs se sont inspirés de l'algorithme QUACQ en espérant améliorer les performances d'acquisition de contraintes d'actives [1, 2, 4, 15, 16, 25, 47, 68, 69]. Ayant un fonctionnement relativement similaire, ces algorithmes sont expliqués dans cette section de manière générale. Tout comme CONACQ, QUACQ et ses variantes utilisent en entrée un vocabulaire préfixé, un biais et une liste d'exemples (positifs ou négatifs) de solutions et procèdent en interagissant avec un utilisateur en posant des questions d'appartenance. QUACQ permet cependant de poser des *questions d'appartenance partielles*, c'est-à-dire des questions où l'algorithme présente à l'utilisateur un exemple incomplet et l'utilisateur doit indiquer si cet exemple partiel viole ou non le problème à modéliser. Ces algorithmes exploitent diverses stratégies pour déterminer les questions d'appartenance qui donnent le plus d'informations possibles, ce qui a un effet direct sur le nombre de questions à poser. QUACQ et ses variantes sont des algorithmes qui ont débuté en 2013 et sont donc encore en pleine expansion. Pour donner un ordre de grandeur sur le nombre de questions posées par ces algorithmes, le problème du Sudoku 9×9 est utilisé comme problème test pour plusieurs de ces algorithmes. Pour ce problème, QUACQ pose 9053 questions qui sont calculées en 2810 secondes [2]. T-QUACQ est une variante qui limite la qualité des questions d'appartenance avec une limite temporelle. Cette variante pose environ 9500 questions, mais avec moins d'une minute de calcul [2]. P_α-QUACQ est une autre version qui se base sur T-QUACQ en faisant varier les limites de temps pour les questions selon des critères de convergence précis. Cet algorithme requiert 5769 questions d'appartenance générées en seulement 12 secondes [1]. Enfin, d'autres versions ont priorisé d'améliorer la polyvalence plutôt que le temps ou le nombre de questions comme MULTIACQ qui vise à apprendre un maximum de contraintes violées par les exemples négatifs, ce qui nécessite 3821 questions d'appartenance.

1.4.4 Problèmes dans les méthodes courantes

Toutes les méthodes d'acquisition de contraintes ont leurs forces et leurs faiblesses. Cette section vise à mettre en lumière certains des problèmes associés aux méthodes décrites dans les sections précédentes.

Premièrement, quelques-unes des approches, comme celle développée par Picard-Cantin et al., sont limitées à explorer une seule contrainte. Cette limitation fait en sorte qu'il est impossible d'apprendre des modèles représentés par une combinaison (conjonction, disjonction ou autre) de contraintes différentes, du moins en tenant compte des interactions entre toutes les contraintes. Certaines autres techniques comme QuAcq peuvent apprendre plusieurs contraintes, mais ne peuvent pas tenir compte d'une *modélisation partielle*, c'est-à-dire une modélisation où certaines contraintes sont connues d'avance. Encore une fois, cette limitation prévient l'obtention d'un modèle qui tient compte de toutes les interactions.

Une troisième faiblesse de certaines des méthodes passives actuelles est l'inclusion d'une supposition d'indépendance entre les variables de décision. Cette supposition peut prendre plusieurs formes, mais est généralement utilisée dans les analyses statistiques des solutions. Dans le cas de Picard-Cantin et al. [56], une probabilité est calculée pour chaque contrainte considérée et le calcul tient compte de la fréquence d'observation des variables de décision qui sont dans la portée de la contrainte. Lorsqu'une variable de décision est dans la portée de plusieurs contraintes, les fréquences d'observation ne sont plus indépendantes et les calculs qui en découlent sont faussés. Cette manière de calculer les probabilités devient donc une approximation. Malgré tout, l'analyse statistique de Picard-Cantin et al. a montré que des résultats statistiques peuvent être exacts, performant même mieux que d'autres techniques reconnues comme le Model Seeker.

Une quatrième faiblesse des méthodes d'acquisition passives est l'ensemble limité de contraintes qui peuvent être apprises. Toutes les méthodes actuelles utilisent un ensemble de contraintes parmi lesquelles le modèle peut être appris, c'est-à-dire le biais. Certaines méthodes ont un biais plus restrictif, ce qui peut être problématique. Par exemple, la technique de Picard-Cantin et al. [56] nécessite le développement d'un algorithme de dénombrement pour chaque contrainte étudiée.

Les méthodes d'acquisition actives comme QuAcq [14] ont comme faiblesse principale le nombre élevé de questions d'appartenance. Plus spécifiquement, même les techniques les plus développées requièrent de poser des milliers de questions à un utilisateur. Dans un contexte d'application réelle, un utilisateur humain ne peut pas répondre à autant de requêtes en un temps raisonnable. De plus, il est peu probable qu'un humain puisse répondre à autant de questions avec une parfaite exactitude. Il est possible d'utiliser un programme pour répondre aux questions d'appartenance, mais cette technique suppose un programme qui possède des connaissances sur le modèle à acquérir, ce qui n'est pas généralement le cas dans les contextes

non académiques.

C'est pour pallier à ces cinq faiblesses que nous proposons, au chapitre suivant, la méthode CABSC. Il s'agit d'une méthode passive capable d'apprendre n'importe quelle contrainte déjà intégrée à un solveur. Cette méthode ne fait aucune supposition sur l'indépendance entre les variables de décision et considère l'ensemble des interactions entre les contraintes durant le processus d'acquisition de contraintes. CABSC peut apprendre de multiples contraintes différentes en plus de pouvoir intégrer dans son analyse un modèle partiel.

Chapitre 2

Constraint Acquisition Based on Solution Counting

Détails de l'article

Titre :	*Constraint Acquisition Based on Solution Counting.*
Auteurs :	Christopher Coulombe et Claude-Guy Quimper.
Publication :	*Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming* (CP 2022), pages 15 :1-15 :16, 2022 [24].

2.1 Résumé

Nous proposons CABSC, un système qui performe de l'acquisition de contraintes basée sur le dénombrement. Pour apprendre un modèle, l'utilisateur fournit des exemples positifs de solutions et un Meta-CSP, c'est-à-dire une modélisation d'un problème combinatoire dont la solution est un CSP. Ce Meta-CSP permet de lister les contraintes potentielles qui peuvent faire partie du CSP que l'utilisateur veut apprendre. Il permet également de décrire les paramètres des contraintes, tels que les coefficients d'une équation linéaire, et d'imposer des contraintes sur ces paramètres. CABSC lit le Meta-CSP en utilisant une version modifiée de MiniZinc et retourne le CSP qui accepte le moins de solutions parmi les CSPs qui acceptent tous les exemples de solutions fournis. La résolution du Meta-CSP se fait par séparation et évaluation où le calcul des bornes utilise des dénombreurs. Nos expérimentations montrent que CABSC réussit à apprendre des contraintes et leurs paramètres tout en nécessitant peu d'exemples positifs.

2.2 Abstract

We propose CABSC, a system that performs Constraint Acquisition Based on Solution Counting. In order to learn a Constraint Satisfaction Problem (CSP), the user provides positive examples and a Meta-CSP, i.e. a model of a combinatorial problem whose solution is a CSP. This Meta-CSP allows listing the potential constraints that can be part of the CSP the user wants to learn. It also allows stating the parameters of the constraints, such as the coefficients of a linear equation, and imposing constraints over these parameters. The CABSC reads the Meta-CSP using an augmented version of the language MiniZinc and returns the CSP that accepts the fewest solutions among the CSPs accepting all positive examples. This is done using a branch and bound where the bounding mechanism makes use of a model counter. Experiments show that CABSC is successful at learning constraints and their parameters from positive examples.

2.3 Introduction

Constraint solvers are used to solve complex combinatorial problems. They require an expert to model the problem using the constraints available in the solver. The model creation is a crucial step, but is often time-consuming. One way to save time to the expert is to suggest a model based on sample solutions. For instance, a hospital that wants to automatize the creation of their work schedules for its staff might provide to the experts previous schedules. Assisted with software, the expert wants to discover what constraint generated the examples. While some of these constraints are already known and even written on legal documents, there are as important constraints that are not written but are part of the work culture. These are the constraints for which a constraint acquisition software becomes handy.

When two constraints are candidates for a model, the one that was the most likely used to generate the sample solutions is the most restrictive one [14]. Different approaches exist to decide which constraint is the most restrictive. There are mainly statistical approaches [13, 14] and approaches based on a ranking system [6] (that includes many other criteria). Current methods analyze the constraint in isolation. However, adding to a model a constraint that accepts many solutions can reduce more the solution space than adding a constraint that accepts few solutions. It all depends on the interaction between the constraints in the model. We propose the first approach that takes into account this interaction. It uses a model counter to make sure that the constraints suggested to the expert are those that are the most likely to explain the observed sample solutions given the constraints that were already identified to be part of the model.

In this paper, we propose CABSC, an algorithm for Constraint Acquisition Based on Solution Counting. CABSC uses examples of solutions to evaluate which constraints to keep from a chosen set of candidates. The selection process is based on solutions counting using model

counters, an approach which differs from the current methods detailed in Section 2.4. The definitions for our approach are given in the Section 2.5, followed by a practical explanation in Section 2.6. Experiments are explained in Section 2.7 and discussed in Section 2.8.

2.4 General Background

Constraint acquisition is an intricate problem that can be solved in a few ways. A first idea called passive learning requires examples of solutions and/or non-solutions. A system chooses which constraint represents best the examples from a preselection of constraints. The preselected pool of constraints from which the model is built is called a bias. Other methods use active learning and generate examples of solution and ask an expert to classify the examples given. From a bias, the system choses the best set of constraints according to the answer provided.

Passive learning systems exploit the idea that the underlying structure of the given examples gives information about the model to learn. Beldiceanu and Simonis [6] created a Model Seeker that learns constraints from a catalog given positive and negative examples. The constraints of the catalog that accepts the positive examples and reject the negative examples are sorted with the more likely constraints having a higher rank. The sorting system is based on a ranking value that is a function of multiple parameters, including the number of solutions satisfying the constraint [5]. A constraint accepting fewer solutions is more likely to be the constraint that generated the examples as there is a lesser chance that the examples are a product of a coincidence. To work, this method needs to make the hypothesis that the constraints learned are independent of each other. That hypothesis is not what transpires in real applications and may result in errors. Two constraints with a small but near identical set of solutions would be picked over two constraints accepting more solutions if picked individually but very few solutions when combined. This is counterproductive as the idea is often to complete an already existing model or to learn multiple constraints at the same time.

Picard-Cantin et al. [13] approached the problem with a statistical approach with the idea that the constraint that best explains the examples is the most improbable one. Equation (2.1) was therefore used by Picard-Cantin et al. [14] to calculate the probability of the constraints where $G_{\mathrm{C}}(P)$ is the probability that a random assignment satisfies the constraint C with the parameters P. The parameters can be, for instance, the coefficients of a linear equation. $S_{\mathrm{C}}(P)$ is the solution set that satisfies the constraint C with the parameters P. The probability is calculated for a constraint over n variables. $prob(e)$ is the probability to observe an assignment e of n variables and $prob(e_i)$ is the probability to observe an assignment of a single variable.

$$G_{\mathrm{C}}(P) = \sum_{\boldsymbol{e} \in S_{\mathrm{C}}(P)} \mathrm{Prob}(\boldsymbol{e}) = \sum_{\boldsymbol{e} \in S_{\mathrm{C}}(P)} \prod_{i=1}^{n} \mathrm{Prob}(e_i) \tag{2.1}$$

A hypothesis of independence between the variables of the constraints is applied in the

equation. Whenever a variable is in the scopes of multiple constraints, the hypothesis of independence between the variables becomes an approximation. In all cases, the preferred constraints are the ones with a small number of solutions but the independence hypothesis can lead to an erroneous ranking of the constraints. Moreover, this system was not designed for learning multiple constraints and requires solution counting algorithms specialized for each constraint.

Another approach was suggested by Bessiere et al. [7] which consists of creating a model from partial queries, a form of active learning, with an algorithm called QuAcq. The system creates an example and asks an expert whether the presented values are valid. The system adapts the learned constraints depending on the provided answer. Recently, QuAcq was improved with a new version called QuAcq2 [8]. In some cases, QuAcq and QuAcq2 can require a number of queries too high to be efficiently answered by a person. The number of queries can go as high as $n^2 \log(n)$ where n is the number of variables of the problem [8]. Multiple authors tackled this problem such as Daoudi et al. [10], Addi et al. [2], Addi et al. [1], Arcangioli and Lazaar [3], Tsouros et al. [20] and Tsouros et al. [19], but up to thousands of queries can still be needed.

2.5 The CABSC approach

The CABSC approach (Constraint Acquisition Based on Solutions Counting) we introduce fulfills three goals:

1. To lift the hypothesis of independence between variables;
2. To allow learning multiple constraints;
3. To work with any set of constraints for which filtering algorithms exist, rather than solution counting algorithms.

CABSC models the process of learning constraints as a Meta-CSP. As will be described in Section 2.5.1, a Meta-CSP is a combinatorial problem whose solution is a CSP. In our case, the solution is the CSP we learn from the examples. When modeling the Meta-CSP, we list the mandatory constraints, i.e. the constraints that we know belong to the model, and also the possible constraints, those that could belong to the model. The variables of the Meta-CSP encode the possible activation of a constraint and also the parameters of the constraints, such as the coefficients of a linear constraint. Solving the Meta-CSP provides the learned model. To do so, we use a branch and bound to decide which constraint to keep and identify the values of the parameters. Our approach uses constraint programming to model a Meta-CSP and to define a family of CSPs from which we can learn. We therefore do not aim to learn any CSP but the optimal CSP among a set programmed through constraint programming. This approach is inspired from regression where one defines a family of functions (e.g. linear functions) and aims at finding the function from this family that best fits the data.

Here, we aim at finding the CSP from a family of CSPs defined by the Meta-CSP that best explains the examples.

As there are multiple candidate constraints that could belong to the learned model, we follow Beldiceanu and Simonis [6] and Picard-Cantin et al. [13] by selecting the constraints that minimize the number of solutions. However, instead of analyzing the constraints individually like Beldiceanu and Simonis [6] and Picard-Cantin et al. [13], our system reasons globally on all constraints which allows us to consider multiple different constraints at once.

In order to lift the hypothesis that variables and constraints in a CSP are independent, we directly count the solutions of a model using a model counter. The solution to our Meta-CSP is therefore a CSP whose constraints are satisfied by all observed examples and is as restrictive as possible, i.e. it minimizes the number of solutions.

Our approach has two main differences from existing methods. The first difference is that constraint programming, through the declaration of a Meta-CSP, is used to define a family of CSPs from which we can learn. A second difference from most existing methods is that we use a criterion with a global view on the model to learn by considering the constraints to learn as a whole instead of individually.

2.5.1 Definition of a Meta-CSP

Following [16], a CSP $\mathcal{P}$ is a triple $\mathcal{P} = \langle X, \text{dom}, C \rangle$ where X is a n-tuple of variables $X = \langle X_1, X_2, \ldots, X_n \rangle$, dom is a function that maps a variable in $X_i \in X$ to a set of values, called domain, that can be assigned to the variable X_i, C is a t-tuple of constraints $C = \langle C_1, C_2, \ldots, C_t \rangle$. A constraint C_j is a pair $\langle R_j, S_j \rangle$ where $S_j \subseteq X$ is the scope of the constraint and R_j is a relation on the variables in S_j. In other words, R_j is a subset of the Cartesian product of the domains of the variables in S_j. A solution to the CSP $\mathcal{P}$ is an assignment to the variables $X = v_1, \ldots, X_n = v_n$ such that $v_j \in \text{dom}(X_j)$ $\forall 1 \leqslant j \leqslant n$ and each C_j is satisfied in that the tuple $\langle v_1, \ldots, v_n \rangle$ projected onto S_j is a tuple in R_j.

We extend the definition of a CSP to a Meta-CSP. The solution of a Meta-CSP is a CSP. In our case, it is the CSP we want to learn. A Meta-CSP is a tuple $M = \langle X, P, \alpha, \text{dom}, E, C \rangle$ where $X = \langle X_1, \ldots, X_n \rangle$ are the decision variables, $P = \langle P_1, \ldots, P_q \rangle$ are the parameter variables, $\alpha = \langle \alpha_1, \alpha_2, \ldots \rangle$ are the activation variables, dom is a function that maps a variable in $X \cup P \cup \alpha$ to a set of values that can be assigned to the variable, E is the example matrix of dimensions $m \times n$, and $C = \{C_1, \ldots, C_t\}$ is a set of constraints. A row $e_i = \langle e_{i,1}, \ldots, e_{i,n} \rangle$ of matrix E satisfies $e_{i,j} \in \text{dom}(X_j)$ and is a solution to the CSP we want to learn. The examples of the matrix must satisfy the constraints that we want to learn.

A constraint C_j is a quadruple $\langle R_j, S_j, P_j, \alpha_j \rangle$ where $S_j \subseteq X$ is the scope of the constraint, $P_j \subseteq P \cup \alpha$ its parameters set and $\alpha_j \in \alpha$ its activation variable. For instance, for a linear

constraint, the parameters P_j are the coefficients that need to be learned. To each constraint C_j is associated the activation variable α_j with domain $\text{dom}(\alpha_j) \subseteq \{\bot, \top\}$. Deciding whether α_j is true ($\top$) is equivalent to deciding whether the constraint appears in the learned model. One can force a constraint to appear in the learned model by setting $\text{dom}(\alpha_j) = \{\top\}$ in the definition of the Meta-CSP. The relation R_j is a set of the assignments accepted by the constraint along with the parameters given to the constraint: $R_j \subseteq \bigtimes_{X \in S_j} x \times \bigtimes_{p \in P_j}$.

A solution to the Meta-CSP is an assignment to the parameter variables $P_1 = p_1, \ldots, P_q = p_q$ and an assignment to the activation variables $\alpha_1 = r_1, \ldots, \alpha_t = r_t$ such that $r_j \in \text{dom}(\alpha_j)$ for all constraints C_j, $p_k \in \text{dom}(P_k)$ for all $1 \leqslant k \leqslant q$. Finally, the examples must satisfy the activated constraint, i.e. $\forall 1 \leqslant j \leqslant t,\ \alpha_j \implies \forall i\ \langle e_{i,1}, \ldots, e_{i,n}, p_1, \ldots, p_q \rangle \in R_j$.

2.6 Framework

2.6.1 The Language

We augmented the MiniZinc language [12] to model a Meta-CSP. The declaration of constraints in a Meta-CSP differs from the one in a CSP in two ways. First, the constraints had to be rewritten in MiniZinc to include the Boolean activation variable. This avoids writing explicitly, for each constraint, the underlying constraints needed for such variables. Second, when declaring the scope of a constraint, the indices of the decision variables in X need to be stored in the constraint. Indeed, the constraint's filtering algorithm needs a map of the decision variables in its scope to the columns of the matrix of examples E. Therefore, constraints used for the Meta-CSP have different specifications from what is possible within MiniZinc, which is why the language had to be augmented. The MiniZinc language was also modified to better communicate with the solver we developed, i.e. imports and heuristics were adapted to give a better control. Even though the modifications to MiniZinc do not change its fundamental structure, the way to write a Meta-CSP is made significantly easier.

Listing 2.1 provides a code snippet written in the augmented MiniZinc language. A set of two-dimensional points are given as solutions of an unknown CSP problem. We know that the x and y coordinates of these points are nonnegative. We do not know whether these points are subject to a linear inequality or an elliptic inequality. This Meta-CSP will tell us.

The decision variables x and y are declared on lines 2 and 3. As their values are known for each example, they are not declared as variables using the keyword *var* but rather as constants corresponding to the column numbers in the example matrix E.

Line 11 declares the first constraint of the problem. It is interpreted as follows: It is a linear constraint whose scope is the decision variable x, whose coefficient vector is [1], whose comparison operator is $\geqslant$, and whose right-hand side is 0. It can be interpreted as $[1]^T x \geqslant 0$. The activation variable is set to *true*, which means that this constraint is known to belong to

```
1  set: domain = 1..10;
2  array: x = [1]; %Points are (x,y)
3  array: y = [2];
4  array: x_y = [1..2];
5  var domain: a;
6  var domain: b;
7  var domain: c;
8  var 0..1: activation1;
9  var 0..1: activation2;
10
11 constraint Linear(x, [1], ">=", 0, true); % x >= 0
12 constraint Linear(y, [1], ">=", 0, true); % y >= 0
13 constraint Linear(x_y, [a,b], "<=", c, activation1);  % a*x + b*y <= c
14 constraint Ellipse(x_y, [a,b], "<=", c, activation2); % a*x^2 + b*y^2 <= c
15 constraint Xor(activation1, activation2, true);
```

Extrait de fichier 2.1 – Code snippet of the augmented MiniZinc

the CSP. Line 12 imposes $y \geqslant 0$ with a similar constraint. Line 13 encodes the first constraint that we want to learn. It is a linear constraint over the variables x and y whose coefficients and right-hand side are unknown and are represented by the parameter variables a, b, and c. Finally, it is unknown whether this constraint belongs to the CSP. The activation variable `activation1` will be set to 1 if it belongs and 0 otherwise. Line 14 encodes the second constraint that we want to learn. It is an elliptic constraint centered at the origin where parameter variables a, b, and c are reused. The activation variable `activation2` is used for this constraint. Line 15 shows an example of a constraint over two activation variables meaning that exactly one constraint among the linear and the elliptic constraint can be activated. This is an example of how one can define the bias (i.e. the family of CSPs from which the CSP is learned) and exploit the full richness of CP to model the learning process.

A constraint can be satisfied by all examples even if the solver chooses not to learn it by setting its activation variable to $\bot$, unlike a reified constraint which would be set to $\bot$ only if the examples are not satisfied.

Figure 2.1 is a graphical representation of the problem encoded in Listing 2.1. The curves represent both candidate constraints: the linear candidate and the elliptic candidate. The dots are the sample solutions that are provided.

We are looking for the CSP that is the most likely the one that generated the points provided in the example matrix E, i.e. the CSP that accepts the fewest solutions among all CSPs that accepts all solutions in E. We see in Figure 2.1 that the Elliptic constraint accepts 9 solutions while the linear constraint accepts 10 solutions. Therefore, our approach learns that an elliptic inequality fits best the examples with parameters $a = 1, b = 2, c = 10$, `activation1` $= \bot$ and `activation2` $= \top$, which confirms the visual intuition.

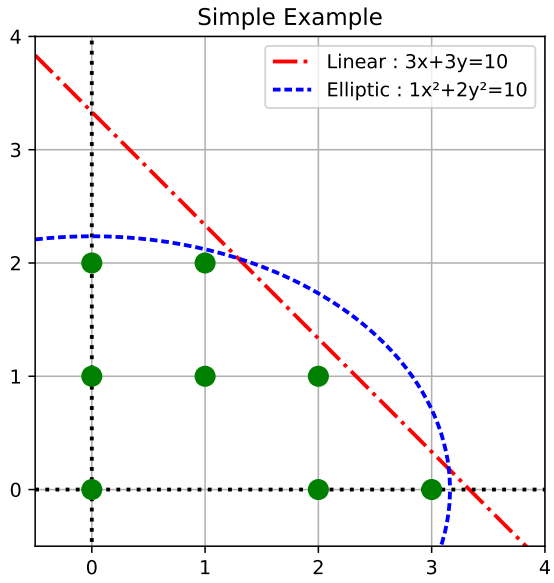

Figure 2.1 – Simple example

2.6.2 The Solver

We created a custom solver called CabscSolver that reads the Meta-CSP written in the augmented MiniZinc language and the example matrix E. This solver finds the CSP that accepts the fewest solutions among all CSPs that accept all examples. CabscSolver uses a branch and bound to solve the problem. The branching variables are the activation and parameter variables $\alpha \cup P$. After branching, constraint propagation is triggered. Let $C(\vec{x}, \vec{p}, \alpha)$ be a constraint where $\vec{x}$ is the vector of decision variables, $\vec{p}$ is the vector of parameter variables, and α is the activation variable. Only the domains of $\vec{p}$ and α need to be filtered as the values of the decision variables are provided by the examples. To filter the constraint, one needs to filter the expression $\alpha \implies \bigwedge_{i=1}^{m} C(e_i|\vec{x}, \vec{p}, \top)$ where $e_i|\vec{x}$ is the projection of the i^{th} example over the decision variables in the scope of the constraint. The filtering can take place only when the value of the activation variable α is known. Indeed, if α is false ($\bot$), the constraint is satisfied and no filtering is required. If α is true ($\top$), a conjunction of constraints needs to be filtered. Each component of the conjunction can be filtered independently, but a more sophisticated algorithm might process the examples in batch to gain in efficiency. The choice is specific to each constraint. In the example of Listing 2.1, if variable `activation1` is set to $\top$ during the search process, the linear constraint filters values 1 and 2 from the domain of c as the point $(x, y) = (3, 0)$ prevents the linear constraint to be satisfied when $c \leqslant 2$.

In order to make the branch and bound effective at minimizing the number of solutions accepted by the CSP we want to learn, one needs to compute a lower bound on this number

of solutions. This computation is carried in two phases. In the first phase, we detect if a situation occurs where it is possible to deduce which CSP accepts the fewest solutions, regardless whether this CSP accepts the examples or not. If such a CSP can be deduced, the second phase launches a model counter to compute the number of solutions for this CSP.

Some constraints have monotonic parameters with respect to the number of solutions they accept [14]. For instance, consider the linear constraint $c^T x \leqslant b$ where the parameters c and b are a vector of nonnegative coefficients and a nonnegative right-hand side. The vector x contains the decision variables. It is clear that the number of solutions accepted by this constraint decreases as the values in c increases and b decreases. In order to obtain the most restrictive constraint, one needs to fix the parameters c to their greatest values in their domains and b to its smallest value. If all parameter variables with more than one value in their domain are monotonic and all constraints agree to set these variables to the same values (either largest or smallest) in order to minimize the number of solutions, then we can proceed to the second phase and compute a lower bound on the number of solutions. Otherwise, we use the number of examples as the trivial lower bound as this is the minimum number of solutions the CSP can accept. Since parameter variables can be subject to constraints, it is possible that fixing the value of the parameter variables leads to inconsistencies. In such a case, the CSP used to calculate the lower bound has no solution. Even if that CSP has no solution, multiple CSPs can exist further in the search tree. We therefore still use the number of examples as a lower bound on those nodes.

In the second phase, the parameter variables are set to their most restrictive value and activation variables that are not set to *false* are forced to be *true* in order to have the maximum number of activated constraints. This results in a CSP $\mathcal{A}$ for which the number of solutions needs to be determined. There exists a few model counters in the literature such as the exact probabilistic model counter `GANAK` [17] or the approximate model counter `ApproxMC4` [9, 18]. Both of these counters can only approximate the number of solutions of a model written as a CNF file. CabscSolver encodes the constraints of $\mathcal{A}$ into a pseudo-Boolean language that is translated to a CNF using the MiniSat+ module NaPS [11]. This CNF is given to the model counter which calculates the number of solutions of the model. This number is used as a lower bound on the number of solutions of the learned CSP for the current node of the branch and bound.

Executing the model counter is the most time-consuming operation in the whole search process. Since the parameter variables are often fixed to the same values (due to their monotonicity), it is worth implementing a cache system. Therefore, before calling the model counter, the system checks whether the generated model was previously counted, and if so, returns the number of solutions previously found.

The resulting algorithm is summarized in Figure 2.2. The next branching is defined by the

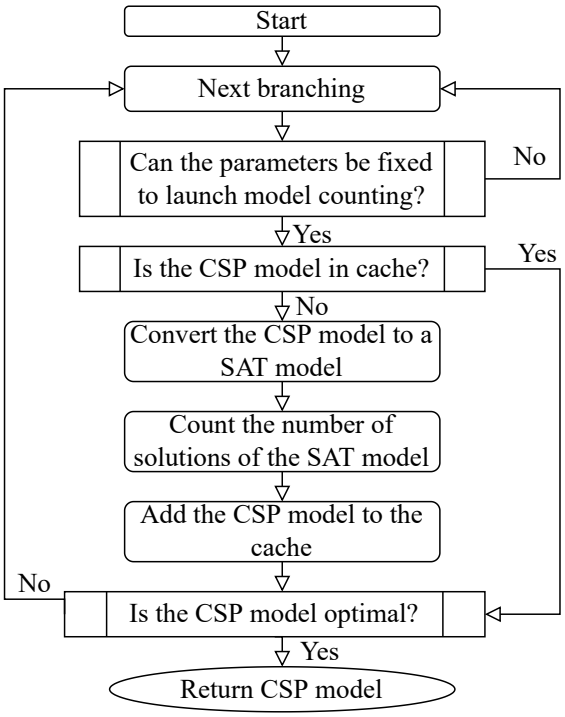

Figure 2.2 – Flow chart for the CABSC approach

best-first-search heuristics, i.e. the open node with the smallest lower bound is expanded. When the lower bound of a node is greater than the number of solutions of the incumbent CSP, this node is closed.

2.7 Experiments

2.7.1 Implementation

We implemented CabscSolver in Python [1]. While this interpreted language leads to a slow execution, in practice, most of the computation time is spent in the model counters. We use `GANAK` [17] and `ApproxMC4` [9, 18] as model counters that are both efficiently implemented in C/C++.

`GANAK` is a probabilistic exact model counter [17]. Using the parameter δ, `GANAK` guarantees with a probability of at least $1-\delta$ that the value provided is an exact count. The approximate model counter `ApproxMC4` [9, 18] was also integrated to our solver to count the number of solutions since some calculations are much faster with this counter. Let F be the real number of solutions of a model. `ApproxMC4` gives an approximation of F with a configurable confidence. More specifically, it returns a count that is guaranteed to be within $[\frac{F}{(1+\epsilon)}, F \cdot (1+\epsilon)]$ with a

1. The code and the benchmarks will be available on the authors' web sites.

probability of at least $1-\delta$, where ϵ and δ are the configurable parameters. The chosen values for the parameters ϵ and δ are discussed in Section 2.7.3.

To read the Meta-CSP models, using the parsing toolkit Lark, we implemented, from scratch, a parser that interprets a subset of the MiniZinc language [12] to which we add the necessary augmentations. MiniZinc was not changed in any way other than the required augmentations. This allows us to efficiently communicate the Meta-CSP models to the solver.

2.7.2 Instances

We try to learn the constraints inspired from nurse scheduling problems. The problem consists of creating a schedule that respects a set of predetermined rules. In these schedules, the increments used are days, meaning that we are only preoccupied on a daily basis whether the nurses work or not. Let $\eta \in \{2,3,4\}$ and $d \in \{7,14,21,28\}$ be the number of nurses and days in a schedule (with $\eta d \leqslant 56$). All instances have a matrix of decision variables $[[X_{(1,1)},\ldots,X_{(1,d)}],\ldots,[X_{(\eta,1)},\ldots,X_{(\eta,d)}]]$. Each variable of the matrix represents a day of work for a nurse with its domain being $\{0,1,2\}$. $X_{i,j}$ takes the value 0 if the nurse i does not work on day j. If the nurse i does work during day j, $X_{i,j}$ takes the value 1 or 2, depending on whether the nurse works in room 1 or 2.

In the first benchmark, denoted **Sequence**, we want to learn one of these two constraints on the rows of the matrix.

$$\textsc{Sequence}([X_{i,1},\ldots,X_{i,d}],l,u,k,V) \qquad \forall 1 \leqslant i \leqslant \eta \tag{2.2}$$

$$\textsc{Among}(t_1,t_2,[X_{i,7w+1},\ldots,X_{i,7(w+1)}],V) \qquad \forall 1 \leqslant i \leqslant \eta,\ \forall 0 \leqslant w < \frac{d}{7} \tag{2.3}$$

Constraint (2.2) is the Sequence constraint [4] that is satisfied when at least l and at most u variables in a window $X_{i,j},\ldots,X_{i,j+k-1}$ of k consecutive variables are assigned to a value in the set V. This constraint is used to spread out the workload of the nurses over the days without underload nor overload. The parameters l, u, and k are unknown and need to be learned. Their domains are given by $\text{dom}(l) = \text{dom}(u) = \text{dom}(k) = [0,7]$ and are subject to $l \leqslant u < k$. The set V is known and fixed to $\{1,2\}$ as these are the values that represent a nurse who is working. Constraint (2.3) simply constrains the number of work days to be at least t_1 and at most t_2 every week. The parameter variables t_1 and t_2 have for domain $\text{dom}(t_1) = \text{dom}(t_2) = [0,7]$. One, and only one, constraint among (2.2) or (2.3) must be activated. We therefore constrain the activation variables of both constraints with a `Xor`, just like the line 15 of Listing 2.1. The benchmark **Sequence** is composed of 368 instances generated with distinct constraints, parameters, and examples. These instances satisfy the Sequence constraint and the parameters lie in the intervals $l,u \in [1,6]$ and $k \in [2,7]$.

The second benchmark, denoted **Complex**, inherits all the characteristics of the **Sequence** benchmark, including the constraint to learn, to which additional known constraints are added

on the decision variables. These constraints have for goal to encode a more realistic situation where constraints that we want to learn are mixed with constraints that are known. For each column $[X_{(1,d)}, \ldots, X_{(\eta,d)}]$ of the matrix that represents the schedule for the day d, we have the constraint $\textsc{Among}(b, 3, [X_{(1,d)}, \ldots, X_{(\eta,d)}], V)$ where $b = 1$ if d is a Monday, Tuesday, Wednesday, or Thursday and $b = 2$ otherwise. This constraint and its parameters are known and added to the Meta-CSP with an activation variable set to $\top$. This constraint does not need to be learned. For instances with 3 or more nurses, we also have another known constraint $X_{(\eta,j)} = 0 \vee X_{(\eta-2,j)} = 0 \quad \forall j \in \{1, \ldots, d\}$ in order to prevent nurse η from working at the same time as $\eta - 2$. When applicable, this constraint is also included in the Meta-CSP as a known constraint. The **Complex** benchmark has 247 instances that satisfy the Sequence constraint with the parameters lying in the intervals $l, u \in [1, 6]$ and $k \in [2, 7]$.

In the third benchmark denoted **Vacation**, the Meta-CSP is identical to the one of **Complex**. However, the examples E that are provided to the solver are particular: nurses can be non-working for 7 consecutive days. This represents a situation where the staff goes on leave during the vacation period. These leaves violate the Sequence constraint and force the solver to activate the Among constraint and learn its parameters t_1 and t_2. The examples were created such that nurse η never takes a vacation but other nurses do. For a problem spanning w weeks, nurses globally take no more than w weeks of vacation. We generated 272 instances for this benchmark such that the instances satisfy the Among constraint. The parameters lie in the intervals $t_1 \in [2, 3]$ and $t_2 \in [3, 7]$.

The last benchmark **Overtime** uses the same Meta-CSP as **Complex** and **Vacation**, but the examples E provided to learn the CSP differ from **Vacation** on one point: rather than leaving for vacations for 7 consecutive days, the nurses in the **Overtime** benchmark work on a stretch of 7 consecutive days. This represents a situation when the hospital is understaffed and nurses need to work overtime. This benchmark has 304 instances such that the instances satisfy the Among constraint and the parameters lie in the intervals $t_1 \in [2, 7]$ and $t_2 \in [4, 7]$ with the restriction $t_1 \leqslant t_2$.

For all benchmarks, the solver aims to learn exactly one constraint among (2.2) and (2.3). The selection depends on the known constraints added to the Meta-CSPs and the examples.

2.7.3 Experimental Setup

For each instance, the CSP we want to learn was written in the MiniZinc language [12] and used to randomly generate up to a thousand solutions. The Meta-CSP model was written in our augmented-MiniZinc language in order to learn which constraint, between the Sequence and the Among constraints, is activated and what are the parameters that were used to generate the examples.

CabscSolver supports two model counters. We first used the solver with the model counter

GANAK [17]. By setting the parameter δ to 0.05, we state that the value returned by the model counter is guaranteed to be exact with a probability of at least 0.95. Tighter guarantees can be used, but the time taken to count the number of solutions of the models increases accordingly. Using this model counter and this configuration, we nevertheless assume the given number of solutions to be exact. GANAK was used with a maximum cache size of 2000 Mb. We ran all benchmarks on the solver using this model counter.

As a second series of tests, we used a mix of ApproxMC4 [9, 18] and GANAK. Some CSP models are faster to evaluate with ApproxMC4, so we tried to make CABSC faster using both model counters. Since ApproxMC4 is not an exact model counter, we did not want to run both model counters at the same time and simply use the result returned by the fastest of the two. When using both model counters, GANAK and ApproxMC4 are simultaneously launched. If GANAK finishes first, ApproxMC4 is terminated. If ApproxMC4 finishes first, GANAK is terminated only if the returned result is conclusive. Indeed, ApproxMC4 returns a solution count that is guaranteed to be within an interval with a parametrized confidence. A solution returned by this model counter could be largely underestimated, which could lead to the wrong CSP model being learned. If F is the exact number of solutions of a CSP, the number of solutions returned by ApproxMC4 lies in $[\frac{F}{(1+\epsilon)}, F \cdot (1+\epsilon)]$ with probability $1-\delta$. When ApproxMC4 returns a number of solutions that is $(1+\epsilon)$ times greater than the number of solutions accepted by the incumbent CSP, the computation of GANAK is halted, and the node is closed, i.e. no children of this node will be explored in the search tree. Otherwise, we draw no conclusion and let GANAK terminate its computation. ApproxMC4 is rather used as a means to close nodes faster than substituting GANAK.

The same way we assumed that GANAK would return exact values, we assume that ApproxMC4 does not give a solution count that is lower than the minimum value of the interval. We used $\delta = 0.10$ and $\epsilon = 0.5$ which means that the count calculated is guaranteed to be in the range $[\frac{F}{1.5}, 1.5F]$ with a probability of at least 0.90. A lower probability is accepted from ApproxMC4 than GANAK since the main focus of using ApproxMC4 is to count CSP models faster than GANAK.

We ran the experiments on a computer with the following configuration: CentOS 7.6.1810, 32 GB ram, Processor Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz, 32 Cores. We simultaneously launch 7 instances of the solver.

From each instance, random subsets of 1, 2, 3, 5, 10, 25, and 100 examples were used. Each time, the top 3 solutions are returned by the solver, and we verify that one of these solutions is the one used to generate the examples. For the **Sequence** and **Complex** benchmarks, the expected constraint to be learned is the Sequence constraint with parameters l, u, and k. For the **Vacation** and **Overtime** benchmarks, the examples violate the Sequence constraint, and the Among constraint is expected to be learned with parameters t_1 and t_2.

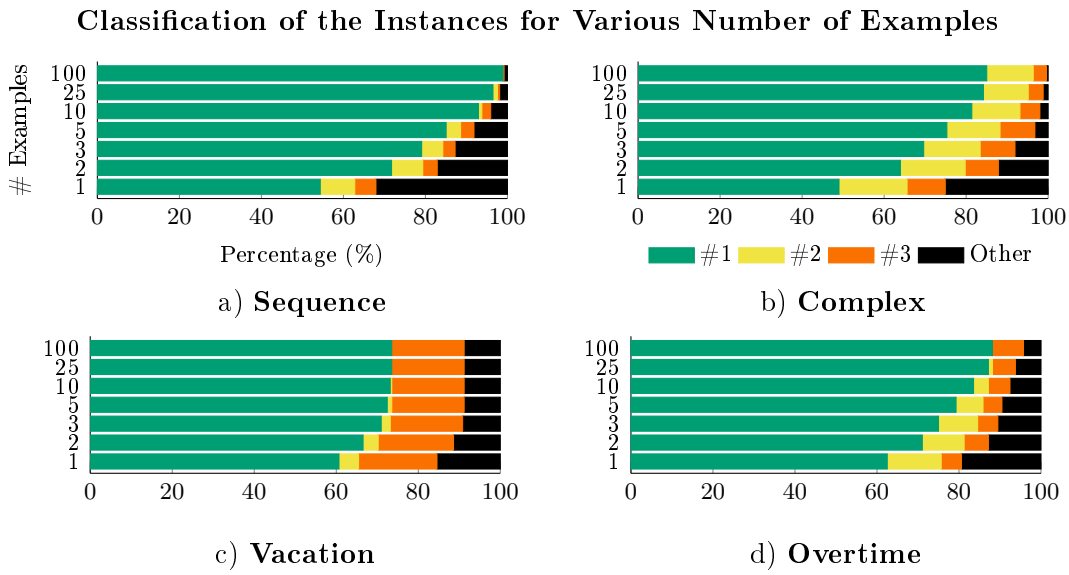

Figure 2.3 – Classification of the instances in percentage for each number of examples. CabscSolver uses `GANAK` as the only model counter.

2.8 Results and Discussion

Figure 2.3 presents the results obtained when running CabscSolver using only `GANAK` for the four benchmarks presented at Section 2.7.2. On the y-axis is the number of examples that are given to the solver. On the x-axis is the proportion of instances for which the solution is the best one returned by the solver, the second best, the third best, or whether the CSP that was used to produce the examples does not appear at all in the top-3 learned models. We recall that the solver returns the CSP that minimizes the number of solutions.

2.8.1 Accuracy

CABSC performs generally well as seen in Figure 2.3. Each benchmark presents a distinct behavior regarding the quality of the results. The first observable behavior is that CABSC succeeds in learning the CSP that was used to generate the data as seen with the benchmark **Sequence**. In this simpler case, the solver has to count the solutions of a conjunction of Sequence constraints, i.e. the constraints to learn. With few examples, our approach stays coherent with the results of Picard-Cantin et al. [13] where they reach above 70% accuracy with a single example of solution, and around 85% accuracy with 5 examples. Extending the number of examples drastically reduces the margin of incorrectly learned instances while the number of examples needed is still relatively low. With only 25 examples, 96.47% of the instances resulted in a correctly learned Sequence constraint at the first try. A few instances could not be resolved even with 100 examples. The unsolved instances occur when the solver finds a more restrictive constraint than the one that was used to generate the examples. This can happen if all the examples given are not enough to filter out parameters that would make

the constraints more restrictive. This is why we see that with more examples given, fewer instances remain unsolved. The same phenomenon happens with the **Complex** benchmark where we see an efficient progression as the number of given examples increases.

Finding a more restrictive constraint is not the only way to get an incorrect model. As Figure 2.3 c) shows, the results for the **Vacation** benchmark converge toward a point where increasing the number of examples does not affect the results while still having a non-negligible proportion (8.82%) of unsolved instances. This is caused by multiple CSPs that are tied. A tie occurs when two distinct CSPs have the same number of solutions. In an instance from **Vacation**, the constraint we want to learn restricts 2 nurses to work a minimum of 3 days and a maximum of 4 days from Monday to Sunday. Since at least one nurse is required to work each day and that a nurse can work a maximum of 4 days within the week, the only way to satisfy the requirements is by having a first nurse working 4 days and the second nurse working 3 or 4 days. It is impossible for one of the nurses to work fewer than 3 days without violating the constraints. The problem comes when setting the value for the minimum number of days a nurse can work during the week. Consider a second selection of parameters where a minimum of 2 working days is required instead of 3. The same solutions are available since this change in parameters does not add solutions. The same goes with a minimum of 1 or 0 working day. This situation leads to four distinct CSPs with the same solution space. Since the objective is to find the CSP accepting the fewest solutions, these four CSPs are equivalent and the solver returns them in an arbitrary order. Most of the unsolved instances in the benchmark **Vacation** have the correct CSP in fourth position, which would have been first if the branching heuristics broke ties differently. We did not observe in our benchmarks situations where the solution spaces differ which let us believe that these models are equivalent. If we pretend for a moment that the **Vacation** benchmark was completed using heuristics that break ties without errors, we obtain the Figure 2.4.

Figure 2.4 shows that this hypothetical heuristic allows solving perfectly the **Vacation** benchmark using as few as 10 examples. Improvements are also present with the other benchmarks. This confirms that finding equivalent CSPs is the main reason why the solver does not succeed to correctly learn some CSPs.

The unsolved instances from the **Complex** benchmark are mainly caused by constraints found more restrictive than the correct one while the unsolved instances from the **Vacation** benchmark are mostly caused by equivalent CSPs. The unsolved instances of the **Overtime** benchmark are caused by a mix of these two reasons.

The final results show that our model can accurately learn the constraints even when the schedules contain vacations, overtime, or constraints that interfere with the constraints one wants to learn. Few examples are needed to obtain good results. These figures demonstrate that CABSC can learn constraints with the right parameters in diverse situations.

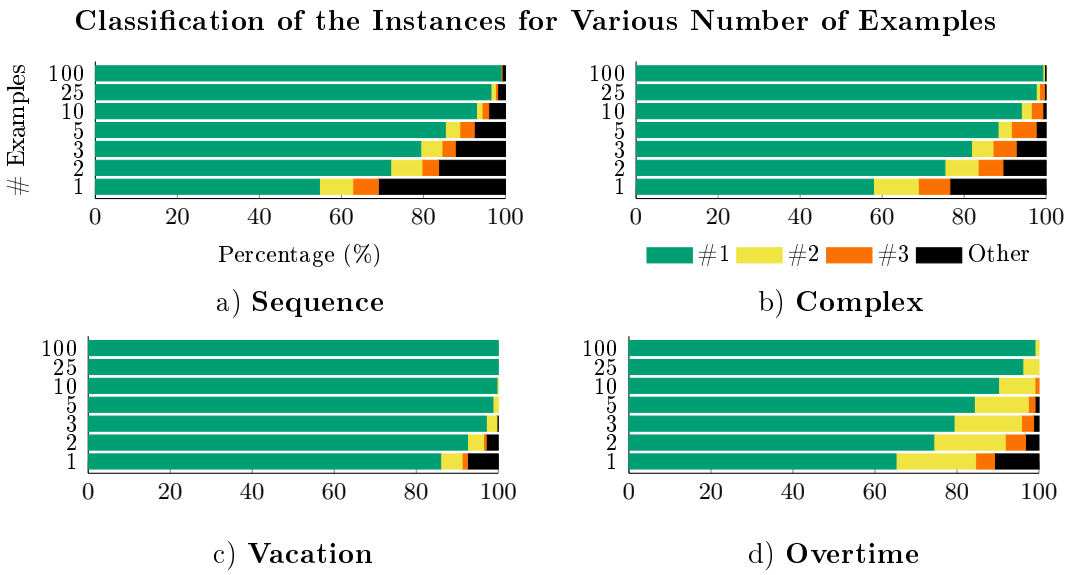

Figure 2.4 – Hypothetical best results for each benchmark

2.8.2 Execution Time

Using GANAK alone

For the **Complex** benchmark, model counting represents on average 93.6% of the time spent in the solver. Solution counting is a #P-difficult problem with few effective algorithms. Even with state-of-the-art tools, computing a lower bound on the number of solutions can take several minutes. The bound that took the longest time to compute by GANAK took 648 seconds. Figure 2.5 represents the time taken to solve all instances, i.e. the $(368 + 247 + 272 + 304) \times 7$ instances that come from the four benchmarks that were solved with 1, 2, 3, 5, 10, 25, and 100 examples using only GANAK as a model counter. Most of the instances are solved within a minute, but the solving time quickly and abruptly rises. This time limitation comes from a few main elements.

First, the size of the Meta-CSP greatly impacts the time needed for CABSC to find a solution. This size is measured in the number of parameter variables and activation variables since their number affects the depth of the search tree, thus the number of nodes explored in the branch and bound. For our instances, a few hundreds nodes could be observed on average resulting in around 30 to 60 unique calls to a model counter.

Second, the examples also impact the total runtime in two ways. With a higher number of examples, the solver is able to filter out more values from the domain of the parameter variables which directly decreases the number of potential calls to a model counter. Using a single example, the instances in the **Complex** benchmark takes on average 385.3 seconds to solve. With a hundred examples, the average time drops to 306.9 seconds, an improvement of 20.35%. The second way the solving time is impacted by the examples is with their length,

i.e. the number of decision variables. The more decision variables, the more Boolean variables in the SAT model to count. For this reason, we were not able to learn the constraints of schedules with a horizon of 56 days or more.

Lastly, all bounds do not take the same computation time. Indeed, we obtain SAT instances with various numbers of Boolean variables and clauses. The internal structure of these SAT instances can also vary. The bound that is the slowest to calculate uses a SAT instance with 672 Boolean variables and 1172 clauses and takes 648 seconds to count. The Boolean model with the greatest number of variables has 804 variables and 2052 clauses and is counted in 0.11 seconds. This demonstrates that the counting time does not only depend on the number of decision variables, but also the structure of the problem.

Using both `GANAK` and `ApproxMC4`

One method used to improve the time needed to solve a Meta-CSP is by combining a probabilistic exact model counter with an approximate model counter. This allows some CSP models to have their solutions counted quicker. The way `ApproxMC4` was added to CabscSolver was to use it to prune CSP models from the search tree when the number of solutions was reasonably far from the number of solutions of the best CSP model found so far, as explained in Section 2.7.3.

This method is a lot faster than using `GANAK` as the only model counter as demonstrated by the Figure 2.6. The worst instance with `GANAK` alone lasted 2350 seconds while the same instance lasted 1099 seconds using `ApproxMC4`. The arithmetic average solving time of the **Complex** drops from 333.0 seconds to 158.7 seconds. This represents an improvement of 52.3% in

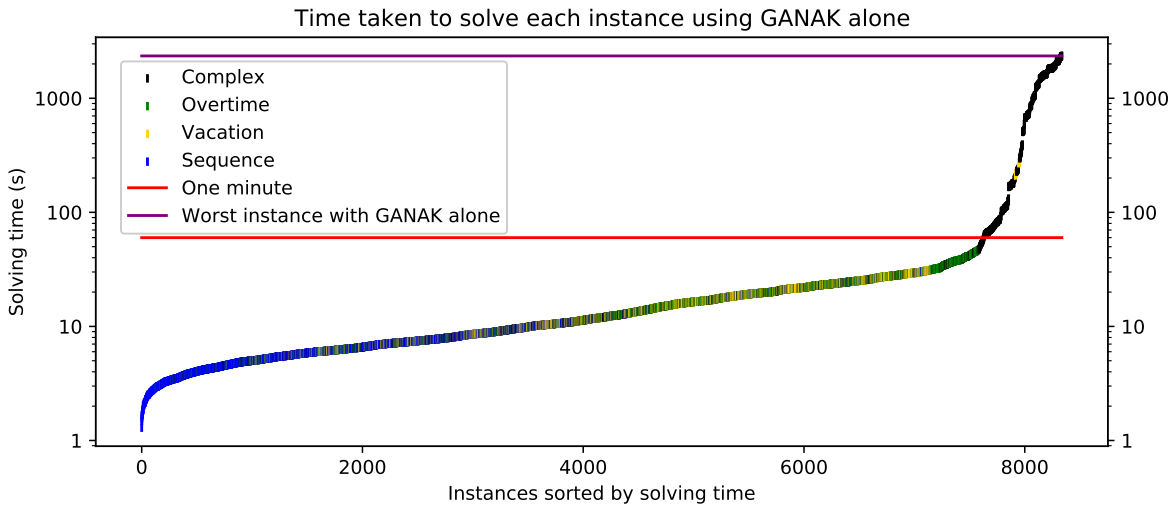

Figure 2.5 – Measures of time for all instances using `GANAK` alone

average. The geometric average drops from 54.2 seconds to 41.0 seconds, an improvement of 24.4%.

The results obtained using both `GANAK` and `ApproxMC4` have a lower accuracy by a small margin. While the accuracy of the results for the **Sequence**, **Vacation** and **Overtime** benchmarks remain unchanged, **Complex** suffers slight changes when few examples of solutions are given. Since the results have no significant differences to be seen on a graph, the changes are textually reported. With a single example of solution, the percentage of correctly learned CSP models drops from 48.99% to 48.48%. When using two examples of solutions, the percentage of correctly learned CSP models drops from 63.97% to 63.56% and with three examples, it drops from 69.64% to 68.83%. When using five examples of solutions or more, adding `ApproxMC4` do not change the results anymore. All the other accuracy results are exactly the same, whether `ApproxMC4` was used or not.

The lack of changes in the accuracy of **Sequence**, **Vacation** and **Overtime** benchmarks is mainly caused by the fact that `ApproxMC4` returns approximations that are often too close to take into account. The solver then has to ask `GANAK` to finish calculating the number of solutions of the CSP model regardless of the time needed by `ApproxMC4`. For the **Complex** benchmark, many CSP models were approximated by `ApproxMC4` a lot faster than `GANAK` could and with values that allow pruning many nodes. `ApproxMC4` sometimes overestimates the count of solutions outside the wanted interval of values. Since we used $\delta = 0.10$ for the model counters, `ApproxMC4` therefore has a probability of at most 0.10 to return values outside the wanted interval. This can cause many of the evaluations to accidentally prune correct CSP models, which can cause the Meta-CSP not to be properly solved. On the opposite side, it

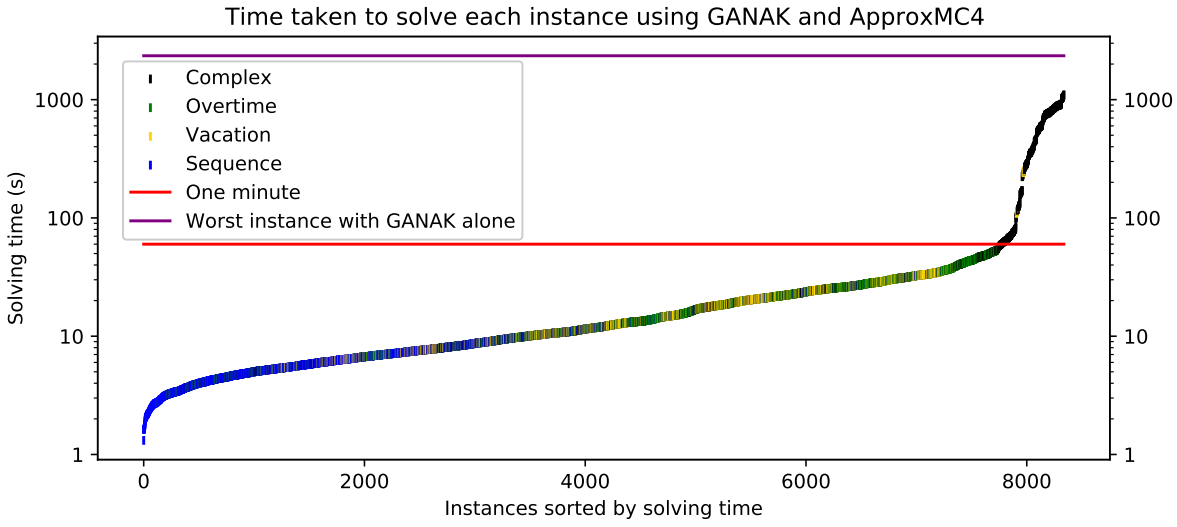

Figure 2.6 – Measures of time for all instances using `GANAK` with `ApproxMC4`

is possible to see improvements in the CSP learned due to overestimations that prune CSP models that would be learned if counted exactly. This happened on few instances from the **Complex** benchmark where the correct CSP went from being the third suggestion to the second. Since the correct CSP was not suggested as a first choice, the accuracy of correctly learned CSP models did not improve from these.

2.8.3 Potential Improvements

There exist several open source model counters that are efficient at counting SAT models, but fewer available programs to count the solutions of a CSP. Translating Sequence constraints into pseudo-Boolean constraints and then to CNF offers no guarantee in the efficiency of the model. Directly counting the solution of a CSP could be faster and would certainly prevent from translating the model.

Parallelization could also speed up the exploration of the search tree. An approach like Embarassingly Parallel Search [15] could be appropriate, but also parallelization within the model counters would be suited as it is offered by `ApproxMC3` [9, 18].

2.9 Conclusion

We introduced CABSC, a technique for Constraint Acquisition Based on Solution Counting. Our approach learns the CSP that accepts all provided examples but that minimizes the size of its solution space. This criterion has proven to return good solutions. The branch and bound uses model counters to compute a bound on the number of solutions for a given CSP. Experimental results show that CABSC successfully learns models and require few examples for our benchmarks.

Références

[1] H. A. Addi et R. Ezzahir, « P_a-quacq : Algorithm for Constraint Acquisition System », in *Smart Data and Computational Intelligence*, 2019, p. 249-256.

[2] H. Ait Addi, C. Bessiere, R. Ezzahir et N. Lazaar, « Time-Bounded Query Generator for Constraint Acquisition », in *To appear in Proceedings of the 15th International Conference on the Integration of Constraint Programming (CPAIOR 2018)*, 2018, p. 1-17.

[3] R. Arcangioli et N. Lazaar, « Multiple Constraint Acquisition », in *Proceedings of the 2015 International Conference on Constraints and Preferences for Configuration and Recommendation and Intelligent Techniques for Web Personalization*, 2015, p. 16-20.

[4] N. Beldiceanu et É. Contejean, « Introducing global constraints in CHIP », *Mathematical and Computer Modelling*, t. 20, n° 12, p. 97-123, 1994.

[5] N. Beldiceanu et H. Simonis, « A Constraint Seeker : Finding and Ranking Global Constraints from Examples », in *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*, 2011, p. 12-26.

[6] N. Beldiceanu et H. Simonis, « A Model Seeker : Extracting Global Constraint Models from Positive Examples », in *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, 2012, p. 141-157.

[7] C. Bessiere et al., « Constraint Acquisition via Partial Queries », in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, F. Rossi, éd., 2013, p. 475-481.

[8] C. Bessiere et al., « Partial Queries for Constraint Acquisition », CoRR, rapp. tech. abs/2003.06649, 2020.

[9] S. Chakraborty, K. S. Meel et M. Y. Vardi, « Algorithmic Improvements in Approximate Counting for Probabilistic Inference : From Linear to Logarithmic SAT Calls », in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016, p. 3569-3576.

[10] A. Daoudi, Y. Mechqrane, C. Bessiere, N. Lazaar et E.-H. Bouyakhf, « Constraint Acquisition with Recommendation Queries », in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016, p. 720-726.

[11] N. Eén et N. Sörensson, « Translating Pseudo-Boolean Constraints into SAT », *Journal on Satisfiability, Boolean Modeling and Computation*, t. 2, n° 1-4, p. 1-26, 2006.

[12] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck et G. Tack, « MiniZinc : Towards a Standard CP Modelling Language », in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, 2007, p. 529-543.

[13] É. Picard-Cantin, M. Bouchard, C. Quimper et J. Sweeney, « Learning Parameters for the Sequence Constraint from Solutions », in *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, 2016, p. 405-420.

[14] É. Picard-Cantin, M. Bouchard, C. Quimper et J. Sweeney, « Learning the Parameters of Global Constraints Using Branch-and-Bound », in *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*, 2017, p. 512-528.

[15] J.-C. Régin, M. Rezgui et A. Malapert, « Embarrassingly parallel search », in *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, 2013, p. 596-610.

[16] F. Rossi, P. van Beek et T. Walsh, éd., *Handbook of Constraint Programming* (Foundations of Artificial Intelligence). Elsevier, 2006, t. 2.

[17] S. Sharma, S. Roy, M. Soos et K. S. Meel, « GANAK : A Scalable Probabilistic Exact Model Counter », in *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI-19)*, 2019, p. 1169-1176.

[18] M. Soos et K. S. Meel, « BIRD : Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting », in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019, p. 1592-1599.

[19] D. C. Tsouros, K. Stergiou et C. Bessiere, « Structure-Driven Multiple Constraint Acquisition », in *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP 2019)*, 2019, p. 709-725.

[20] D. C. Tsouros, K. Stergiou et P. G. Sarigiannidis, « Efficient Methods for Constraint Acquisition », in *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, 2018, p. 373-388.

Chapitre 3

Travaux subséquents avec CABSC

Ce chapitre introduit des possibilités de travaux subséquents pour l'approche CABSC. Premièrement, nous présentons dans la section 3.1 un exemple de problème pour lequel CABSC n'est pas le mieux adapté pour performer l'acquisition de contraintes. Cet exemple permet de mettre en évidence certaines difficultés de CABSC. Par la suite, dans la section 3.2, certaines problématiques de l'approche CABSC sont discutées ainsi que des pistes de réflexion pour des solutions.

3.1 Problème de préséances

L'article *Constraint Acquisition Based on Solution Counting* reporté au chapitre 2 de ce mémoire possède plusieurs résultats dans des contextes de créations d'horaires simplifiés. Ces contextes d'horaires font chacun ressortir quelques problèmes de CABSC, mais il est d'intérêt d'examiner une autre situation qui révèle davantage les limites de l'approche. Ces problèmes sont ensuite analysés à la section 3.2. Le problème d'apprentissage de préséance expliqué à la définition 20 en est un tel exemple.

> **Définition 20** (Problème d'apprentissage de préséances)**.** Soit L une liste de n tâches qui doivent être exécutées séquentiellement. Soit $P \subseteq L \times L$ un ensemble de paires de tâches appelées préséances. Une paire $\langle a, b \rangle \in P$ indique que la tâche a doit être accomplie avant que la tâche b puisse commencer. Le problème d'apprentissage de préséances consiste à apprendre les prédécences qui existent dans P et qui ont été satisfaites lors de la génération des solutions.

Afin d'utiliser CABSC pour apprendre des préséances, on fournit au programme une liste d'exemples positifs. Chaque exemple consiste en une liste de nombres qui représentent des tâches dans l'ordre qu'elles sont exécutées dans la solution. Pour chaque instance du problème d'apprentissage de préséances, un ensemble fixé de préséances est préétabli et les exemples respectent chacune de ces préséances. Pour résoudre ce problème, CABSC requiert

la modélisation d'un Meta-CSP dont l'espace d'apprentissage donne la possibilité d'apprendre les bonnes préséances.

Le Meta-CSP doit pouvoir apprendre jusqu'à $n(n-1)$ contraintes, soit une contrainte de préséance entre chaque paire (ordonnée) de tâches. Cependant, les algorithmes de filtrage exploitent les exemples de solutions donnés pour exclure les préséances pour lesquelles il existe un contre-exemple. La solution est alors de fixer les préséances restantes à $\top$, solution qui peut être obtenue par CABSC en faisant seulement un appel au dénombreur. Cet appel au dénombreur est présent, car il est intégré dans la structure de CABSC, mais est facultatif et sert seulement à évaleur la solution objectif qui est le nombre de solutions admises par le CSP. Si on néglige cet appel facultatif, il ne faut qu'un filtrage des contraintes à partir des exemples de solutions pour découvrir le modèle optimal.

Par contre, si on exige du Meta-CSP qu'il apprenne au plus k préséances, par exemple avec la contrainte $\text{constraint } sum(\alpha) \leqslant k$, alors CABSC pourrait devoir évaluer un nombre exponentiel d'appels aux dénombreurs indépendamment de l'heuristique choisit. Le problème est que CABSC doit trouver la combinaison idéale de contraintes tout en se limitant à k préséances. Le nombre d'appels aux dénombreurs est donc de l'ordre de $\binom{|\alpha|}{k}$.

Le problème d'apprentissage de préséances met en lumière quelques problèmes. Premièrement, malgré un filtrage des contraintes de préséances, en limitant le nombre de contraintes apprises, CABSC requiert un nombre exponentiel d'appels à un dénombreur de solutions où chaque appel requiert un temps exponentiel. Le temps de calcul peut facilement exploser ce qui rend CABSC impraticable sur certains problèmes. De plus, si la solution retournée par CABSC est une liste de préséances ayant un maximum de contraintes mises à $\top$, il est probable que des préséances redondantes soient apprises comme $X_1 < X_2$, $X_2 < X_3$ et $X_1 < X_3$ où la dernière est redondante (par transitivité). Bien que ces contraintes redondantes ne seraient pas fausses, il n'est pas idéal d'apprendre des contraintes qui n'améliorent pas l'espace de solutions du CSP appris. Dans un cas où on limite le nombre de contraintes à apprendre, il faut être capable de facilement filtrer ces contraintes comme il est certain qu'elles n'ajoutent pas d'informations au modèle CSP retourné.

3.2 Travaux futurs

CABSC est un outil qui corrige plusieurs problèmes présents dans les méthodes courantes d'apprentissage de contraintes, mais introduit à son tour plusieurs difficultés. Le plus gros problème est le temps de résolution d'un Meta-CSP qui peut être largement trop élevé pour une utilisation pratique de l'approche. Deux facteurs principaux expliquent les temps élevés : le temps nécessaire pour dénombrer les modèles et la recherche d'un ensemble de paramètres dans un arbre de recherche exponentielle. La première problématique est améliorée avec l'avancement des techniques de dénombrement qui évoluent sans une intervention directe sur

l'approche CABSC, tandis que la deuxième dépend directement de la structure de CABSC. Ce problème peut être décortiqué en plusieurs sous-problèmes qui sont explorés dans cette section : le calcul de la borne inférieure qui n'est pas toujours optimal, le choix de l'heuristique qui peut être difficile et l'absence de vérification de dominance entre les contraintes considérées.

3.2.1 Calcul de la borne inférieure

Un problème récurrent dans la résolution par séparation et évaluation est qu'il s'avère souvent difficile d'obtenir une borne efficace lors de la recherche dans l'espace de solutions. Même les problèmes les mieux connus, comme le problème du commis voyageur [3], attirent encore l'attention des chercheurs quant à l'amélioration des bornes inférieures [18, 40, 41, 57]. CABSC n'y fait pas exception et souffre intensément de ce problème.

Pour que le solveur puisse dénombrer les Meta-CSP, il faut que les noeuds de l'arbre de recherche représentent des CSPs valides. Or, il est souvent difficile ou impossible d'avoir de tels noeuds avant d'être à une feuille de l'arbre. Ceci est expliqué par le fait qu'il est fréquent que le nombre de solutions des contraintes à apprendre ne soit pas monotone par rapport à leurs paramètres et donc que les interactions entre les contraintes soient difficiles à prévoir.

En fait, le seul cas où CABSC arrive à dénombrer des noeuds qui ne sont pas des feuilles est lorsque le Meta-CSP contient des contraintes dont le nombre de solutions est monotone par rapport à des paramètres qui sont fixés en dernier par le solveur. En effet, si tous les paramètres responsables de la monotonie des contraintes sont fixés avant la fin des branchements, alors il devient impossible de prévoir le nombre de solutions plus tard dans l'arbre de recherche. Ceci est expliqué par le fait que les contraintes ne peuvent pas être dénombrées tant qu'elles possèdent des paramètres non reliés à la monotonie de contraintes et qui ne sont pas associées à des valeurs. Cette particularité fait en sorte que l'arbre de recherche est souvent développé au complet avant d'en faire du dénombrement. Autrement dit, CABSC n'arrivent pas à calculer de bornes inférieures à moins qu'il ne reste que les paramètres reliés à une monotonie du nombre de solutions ou si toutes les variables d'activations sont fixées. L'exploration de l'arbre de recherche devient exhaustive et contredit l'objectif d'utiliser la séparation et l'évaluation.

Il existe peu de solutions à ce problème. Une première idée pourrait être que lorsque l'on rencontre un noeud, on lance le dénombrement sur l'ensemble des sous-noeuds. En théorie, cette méthode permettrait d'obtenir les meilleures bornes inférieures possibles, mais en pratique, elle demande de dénombrer l'entièreté de l'espace de recherche dès le noeud racine. Cette idée n'est donc pas applicable.

Une seconde approche plus réaliste pourrait être d'essayer de fixer temporairement toutes les variables d'activation à $\top$ dès que les paramètres des contraintes sont fixés. Si les heuristiques choisies privilégient de fixer ces paramètres, alors toute la partie inférieure de l'arbre de recherche pourrait exploiter le dénombrement des modèles. Dans l'éventualité où le CSP

devient insatisfiable avec plusieurs contraintes d'activées, la borne inférieure pourrait être le nombre d'exemples de solution fournis, tel que mentionné dans l'article original [24]. Malheureusement, cette méthode demande de forcer l'heuristique de recherche à faire des branchements sur les paramètres, ce qui peut être difficile, car le filtrage des domaines des paramètres ne s'effectue que lorsque la contrainte associée a sa variable d'activation de fixée à $\top$ afin d'éviter de retirer des valeurs qui n'auraient pas dû être retirées si la contrainte était désactivée. Cette solution pourrait tout de même valoir la peine d'être investiguée.

3.2.2 Choix d'heuristique

CABSC est très sensible aux choix d'heuristiques. Comme le montre l'exemple de l'apprentissage de préséances à la section 3.1, une heuristique peut faire la différence entre trouver la solution avec un nombre polynomial d'appels aux dénombreurs ou avec un nombre exponentiel d'appels. Également, comme discuté à la section 3.2.1, l'heuristique peut faire la différence entre être capable de calculer des bornes inférieures et devoir faire une recherche exhaustive de l'espace de solutions. Ce problème est commun à tous les problèmes combinatoires, mais il vaut tout de même la peine d'élaborer ce problème que CABSC subit.

Dans son état actuel, la structure de CABSC permet de faire un choix d'heuristique qui décide de la manière de fixer les différents paramètres des contraintes et un second choix d'heuristique qui permet de choisir de quelle manière la recherche dans l'arbre doit être effectuée. Ces choix d'heuristiques sont importants, car ils permettent à l'utilisateur de personnaliser la recherche. Une difficulté de CABSC est de gérer l'interaction entre ces deux types d'heuristiques. Si un utilisateur choisit une recherche en largeur pour explorer les différents noeuds de l'arbre, mais que l'heuristique choisit pour les variables privilégie de fixer les $|\alpha|$ variables d'activation, alors le solveur ouvre forcément un minimum de $2^{|\alpha|}$ noeuds sans jamais calculer une seule borne inférieure. Cette dynamique rend peu pratique l'utilisation de ces deux heuristiques sur des problèmes au-delà de quelques dizaines de variables d'activations. Au contraire, la recherche en largeur pourrait être hautement bénéfique si elle était utilisée conjointement avec les améliorations proposées à la section 3.2.1 puisqu'elle bénéficierait de davantage de valeurs de bornes inférieures.

Le problème du choix d'heuristique n'est pas un problème pour lequel il existe de solution, car les heuristiques sont des choix qui doivent être faits par les utilisateurs en connaissance du problème à apprendre. Dans le cas d'un Meta-CSP, une expertise sera probablement toujours requise pour ces choix. Cependant, si des améliorations de calcul de bornes inférieures fonctionnaient et si CabscSolver devenait un outil public, le solveur pourrait bénéficier d'une documentation sur les effets des heuristiques.

3.2.3 Dominance entre les contraintes

Soit M un modèle partiel d'un problème de satisfaction de contraintes et C_1 et C_2 deux contraintes qui peuvent être apprises et donc ajoutées à M. Si C_1 est dominée par C_2, alors $M_1 := M \wedge C_1$ est dominée par $M_2 := M \wedge C_2$ puisque l'espace de solutions de M_1 est forcément inclus dans l'espace de solutions de M_2. D'un point de vue d'apprentissage de contraintes, il serait donc d'intérêt de repérer les contraintes qui ont un lien de dominance afin de réduire le nombre d'appels qui peuvent être faits aux dénombreurs de modèles.

Deux cas principaux peuvent survenir où connaître les relations de dominance serait un atout. Premièrement, si le noeud ayant le CSP M_1 est dénombré et ne comporte pas de contradictions, alors le noeud ayant le CSP M_2 ne pourrait jamais obtenir un nombre de solutions plus petit. Le dénombrement du CSP M_2 devrait alors être évité, car il ne fournit aucune information et consomme du temps précieux. Dans un tel cas, si on sait que $C_1 \succ C_2$, il peut être d'intérêt de privilégier le dénombrement de M_1. Le deuxième cas où la connaissance des relations de dominance peut être utile est lorsque les redondances du modèle doivent être retirées. L'exemple 19 démontre un tel cas.

> **Exemple 19.** Considérons la problème d'apprentissage de préséances de la définition 20. Soit M le CSP appris à la fin de la résolution du problème. Le CSP M contient les contraintes $C_1 := X_1 < X_2$, $C_2 := X_2 < X_3$ et $C_3 := X_1 < X_3$. Dans ce cas, il existe un cas de dominance tel que $C_1 \wedge C_2 \prec C_3$ ce qui cause la contrainte C_3 à être redondante. Le modèle final pourrait ainsi bénéficier si C_3 était retirée.

Ce dernier exemple démontre qu'une contrainte peut être dominée par la conjonction de contraintes.

Quelques techniques existent pour aider avec la dominance de contraintes. Une première idée concerne d'introduire des contraintes qui agissent afin de briser les liens de dominance, tel qu'explorer par Chu et Stuckey [22]. Une seconde approche pourrait consister à détecter les relations de dominances. Une manière pourrait être d'implémenter un algorithme de vérification pour chaque contrainte, ce qui permettrait de comparer les contraintes de même type ensemble. Avec cette méthode, les redondances simples pourraient être évitées. Par contre, la redondance de l'exemple 19 serait encore cachée, à moins d'inclure dans les algorithmes une vérification pour les conjonctions de contraintes. Enfin, la vérification de relations de dominances entre contraintes différentes semble être plus difficile à trouver. De telles contraintes pourraient être par exemple $C_1 := \textsc{Among}([X_1, X_2, X_3], 0, 3, \{1, 2, 3, 4\})$, $C_2 := 0 < X_1 + X_2 + X_3 < 5$ et $\text{dom}(X_1) = \text{dom}(X_2) = \text{dom}(X_3) = \{1, 2, 3, 4\}$ tel que $C_2 \prec C_1$. La vérification de relations de dominance dans ce cas pourrait être un sujet de recherche intéressant.

Conclusion

Dans le cadre de ce mémoire, nous avons abordé des concepts reliés à l'apprentissage de contraintes. Une introduction aux concepts centraux à la programmation par contraintes a été explorée dans le chapitre 1 en discutant du paradigme de programmation par contraintes, de la modélisation booléenne, des techniques de dénombrement et de l'acquisition de contraintes.

Chaque notion du premier chapitre est une partie fondamentale du deuxième chapitre. Dans celui-ci, nous avons exploré une nouvelle idée pour performer l'acquisition de contraintes. Tandis que la plupart des méthodes actuelles se concentrent sur les interactions avec les utilisateurs ou s'appuient sur des suppositions d'indépendance entre les variables pour acquérir des contraintes, la méthode explorée dans ce chapitre approche l'acquisition de contraintes d'un nouvel angle. Cette méthode, appelée CABSC, est une nouvelle technique d'acquisition de contraintes qui exploitent le dénombrement de modèles de problèmes de satisfaction de contraintes et exploite la pleine richesse de la programmation par contraintes pour modéliser le processus d'apprentissage à l'aide d'un Meta-CSP. Cette manière de procéder permet d'être plus automatique qu'une technique qui engage un utilisateur dans le procédé d'apprentissage. De plus, comme les choix de contraintes sont faits à l'aide du dénombrement de modèles, CABSC retire l'hypothèse selon laquelle les variables sont indépendantes et offre des contraintes qui doivent forcément tenir compte des interactions dans le modèle.

L'approche établie au chapitre 2 n'est pas sans problème. L'acquisition de contraintes est un sujet complexe et les connaissances actuellement nous poussent à faire des compromis. Le troisième chapitre aborde quelques-uns des compromis faits dans CABSC, comme le calcul des bornes inférieures qui n'est pas optimal et la redondance de contraintes apprises. L'approche d'acquisition de contraintes explorée dans ce mémoire est donc une nouvelle méthode qui pourra idéalement lancer des chercheurs dans de nouvelles directions où de moins en moins de compromis seront à faire, en particulier le développement de techniques qui incluent les interactions entre les variables de décision et les paramètres.

Bibliographie

[1] H. A. Addi et R. Ezzahir, « P_a-quacq : Algorithm for Constraint Acquisition System, » in *Smart Data and Computational Intelligence*, 2019, p. 249-256.

[2] H. Ait Addi, C. Bessiere, R. Ezzahir et N. Lazaar, « Time-Bounded Query Generator for Constraint Acquisition, » in *To appear in Proceedings of the 15th International Conference on the Integration of Constraint Programming (CPAIOR 2018)*, 2018, p. 1-17.

[3] D. L. Applegate, R. E. Bixby, V. Chvatál et W. J. Cook, *The Traveling Salesman Problem : A Computational Study*. 2006, p. 606.

[4] R. Arcangioli et N. Lazaar, « Multiple Constraint Acquisition, » in *Proceedings of the 2015 International Conference on Constraints and Preferences for Configuration and Recommendation and Intelligent Techniques for Web Personalization*, 2015, p. 16-20.

[5] F. Bacchus, S. Dalmao et T. Pitassi, « Algorithms and complexity results for #SAT and Bayesian inference, » in *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science.*, 2003, p. 340-351.

[6] R. J. Bayardo et J. D. Pehoushek, « Counting Models Using Connected Components, » in *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, 2000, p. 157-162.

[7] R. J. Bayardo et R. C. Schrag, « Using CSP Look-Back Techniques to Solve Real-World SAT Instances, » in *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*, 1997, p. 203-208.

[8] N. Beldiceanu, M. Carlsson et J.-X. Rampon, « Global Constraint Catalog, 2nd Edition (revision a), » Computer Systems Laboratory, rapp. tech. 2012 :03, 2012.

[9] N. Beldiceanu et É. Contejean, « Introducing global constraints in CHIP, » *Mathematical and Computer Modelling*, t. 20, n° 12, p. 97-123, 1994.

[10] N. Beldiceanu et H. Simonis, « A Constraint Seeker : Finding and Ranking Global Constraints from Examples, » in *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*, 2011, p. 12-26.

[11] N. Beldiceanu et H. Simonis, « A Model Seeker : Extracting Global Constraint Models from Positive Examples, » in *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, 2012, p. 141-157.

[12] C. Bessiere, G. Katsirelos, N. Narodytska, C.-G. Quimper et T. Walsh, « Decompositions of all different, global cardinality and related constraints, » in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, 2009, p. 419-425.

[13] C. Bessiere, F. Koriche, N. Lazaar et B. O'Sullivan, « Constraint acquisition, » *Artificial Intelligence*, t. 244, p. 315-342, 2017.

[14] C. Bessiere et al., « Constraint Acquisition via Partial Queries, » in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, F. Rossi, éd., 2013, p. 475-481.

[15] C. Bessiere et al., « New Approaches to Constraint Acquisition, » in *Data Mining and Constraint Programming : Foundations of a Cross-Disciplinary Approach*. 2016, p. 51-76.

[16] C. Bessiere et al., « Partial Queries for Constraint Acquisition, » CoRR, rapp. tech. abs/2003.06649, 2020.

[17] E. Birnbaum et E. L. Lozinskii, « The Good Old Davis-Putnam Procedure Helps Counting Models, » *J. Artif. Int. Res.*, t. 10, n° 1, p. 457-477, 1999.

[18] H.-J. Böckenhauer, J. Hromkovič, R. Klasing, S. Seibert et W. Unger, « An Improved Lower Bound on the Approximability of Metric TSP and Approximation Algorithms for the TSP with Sharpened Triangle Inequality, » in *STACS 2000*, 2000, p. 382-394.

[19] R. E. Bryant, « Graph-Based Algorithms for Boolean Function Manipulation, » *IEEE Transactions on Computers*, t. C-35, 1986.

[20] S. Chakraborty, K. S. Meel et M. Y. Vardi, « A Scalable Approximate Model Counter, » in *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, 2013, p. 200-216.

[21] S. Chakraborty, K. S. Meel et M. Y. Vardi, « Algorithmic Improvements in Approximate Counting for Probabilistic Inference : From Linear to Logarithmic SAT Calls, » in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016, p. 3569-3576.

[22] G. Chu et P. J. Stuckey, « Dominance breaking constraints, » *Constraints*, t. 20, p. 155-182, 2015.

[23] S. A. Cook, « The Complexity of Theorem-Proving Procedures, » in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, p. 151-158.

[24] C. Coulombe et C.-G. Quimper, « Constraint Acquisition Based on Solution Counting, » in *Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, 2022, 15 :1-15 :16.

[25] A. Daoudi, Y. Mechqrane, C. Bessiere, N. Lazaar et E.-H. Bouyakhf, « Constraint Acquisition with Recommendation Queries, » in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016, p. 720-726.

[26] A. Darwiche, « New Advances in Compiling CNF to Decomposable Negation Normal Form, » in *Proceedings of the 16th European Conference on Artificial Intelligence*, 2004, p. 318-322.

[27] A. Darwiche et P. Marquis, « Decomposable Negation Normal Form, » *J. ACM*, t. 48, p. 608-647, 2001.

[28] A. Darwiche et P. Marquis, « A Knowledge Compilation Map, » *Journal of Artificial Intelligence Research*, t. 17, p. 229-264, 2002.

[29] A. Darwiche et K. Pipatsrisawat, *Handbook of Satisfiability : Second Edition* (Frontiers in Artificial Intelligence and Applications). IOS Press, 2021, chap. 3.

[30] B. A. Davey et H. A. Priestley, *Introduction to Lattices and Order*, 2^{e} éd. Cambridge University Press, 2002.

[31] N. Eén et N. Sörensson, « An Extensible SAT-solver, » in *SAT*, sér. Lecture Notes in Computer Science, Springer, 2003, p. 502-518.

[32] N. Eén et N. Sörensson, « Translating Pseudo-Boolean Constraints into SAT, » *Journal on Satisfiability, Boolean Modeling and Computation*, t. 2, n^{o} 1-4, p. 1-26, 2006.

[33] E. C. Freuder et R. J. Wallace, « Suggestion Strategies for Constraint-Based Matchmaker Agents, » in *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP 1998)*, 1998, p. 192-204.

[34] A. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández et I. Miguel, « The design of essence : A constraint language for specifying combinatorial problems, » in *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007, p. 80-87.

[35] A. M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández et I. Miguel, « Essence : A Constraint Language for Specifying Combinatorial Problems, » *Constraints*, t. 13, p. 268-306, 2008.

[36] I. P. Gent et P. Nightingale, « A new encoding of alldifferent into SAT, » in *International Workshop on Modelling and Reformulating Constraint Satisfaction*, 2004, p. 95-110.

[37] C. P. Gomes, J. Hoffmann, A. Sabharwal et B. Selman, « From Sampling to Model Counting, » in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007, p. 2293-2299.

[38] C. P. Gomes, A. Sabharwal et B. Selman, « Model Counting : A New Strategy for Obtaining Good Bounds, » in *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI 06)*, t. 1, Boston, Massachusetts, 2006, p. 54-61.

[39] C. P. Gomes, A. Sabharwal et B. Selman, *Handbook of Satisfiability : Second Edition* (Frontiers in Artificial Intelligence and Applications). IOS Press, 2021, chap. 25.

[40] I. Gorodezky, R. D. Kleinberg, D. B. Shmoys et G. Spencer, « Improved Lower Bounds for the Universal and a priori TSP, » in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, 2010, p. 178-191.

[41] J. Graczyk et N. Mihalache, « Optimal bounds for the analytical traveling salesman problem, » *Journal of Mathematical Analysis and Applications*, t. 507, n° 2, 2022.

[42] J. Harder, *DIMACS CNF - Varisat Manual*, English, version Version 0.2.x.

[43] E. Hebrard, E. O'Mahony et B. O'Sullivan, « Constraint Programming and Combinatorial Optimisation in Numberjack, » in *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2010, p. 181-185.

[44] M. Held et R. M. Karp, « The Traveling-Salesman Problem and Minimum Spanning Trees, » *Operations Research*, t. 18, p. 1138-1162, 1970.

[45] D. J. Johnson et M. A. Trick, *Cliques, Coloring, and Satisfiability : Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, 1996.

[46] J.-L. Lauriere, « A language and a program for stating and solving combinatorial problems, » *Artificial Intelligence*, t. 10, p. 29-127, 1978.

[47] N. Lazaar, « Parallel Constraint Acquisition, » in *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI-21)*, 2021, p. 3860-3867.

[48] L. A. Levin, « Universal Sequential Search Problems, » t. 9, 1973, p. 115-116.

[49] N. N. 1. Madras, *Lectures on Monte Carlo methods* (Fields Institute monographs). 2002, t. 16.

[50] K. S. Meel et S. Akshay, « Sparse Hashing for Scalable Approximate Model Counting : Theory and Practice, » in *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, 2020, p. 728-741.

[51] N. Michel, J. J. Cater III et O. Varela, « Active versus passive teaching styles : An empirical study of student learning outcomes, » *Human Resource Development Quarterly*, t. 20, p. 397-418, 2009.

[52] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang et S. Malik, « Chaff : Engineering an Efficient SAT Solver, » in *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*, 2001, p. 530-535.

[53] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck et G. Tack, « MiniZinc : Towards a Standard CP Modelling Language, » in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, 2007, p. 529-543.

[54] G. Pesant, C.-G. Quimper et H. Verhaeghe, « Practically Uniform Solution Sampling in Constraint Programming, » in *To appear in Proceedings of the 19th International Conference on the Integration of Constraint Programming (CPAIOR 2022)*, 2022.

[55] É. Picard-Cantin, M. Bouchard, C. Quimper et J. Sweeney, « Learning Parameters for the Sequence Constraint from Solutions, » in *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, 2016, p. 405-420.

[56] É. Picard-Cantin, M. Bouchard, C. Quimper et J. Sweeney, « Learning the Parameters of Global Constraints Using Branch-and-Bound, » in *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*, 2017, p. 512-528.

[57] G. Righini, « Efficient optimization of the Held–Karp lower bound, » *Open Journal of Mathematical Optimization*, t. 2, 2021.

[58] K. H. Rosen, *Discrete Mathematics and Its Applications*. McGraw-Hill Education, 2018.

[59] F. Rossi, P. van Beek et T. Walsh, éd., *Handbook of Constraint Programming* (Foundations of Artificial Intelligence). Elsevier, 2006, t. 2.

[60] S. Russell et P. Norvig, *Artificial Intelligence : A Modern Approach*, 4th. Prentice Hall Press, 2020.

[61] A. Sabry et M. Felleisen, « Reasoning about Programs in Continuation-Passing Style, » in *LISP AND SYMBOLIC COMPUTATION*, 1993, p. 288-298.

[62] T. Sang, F. Bacchus, P. Beame, H. Kautz et T. Pitassi, « Combining component caching and clause learning for effective model counting, » in *Proceedings of the 7th international conference on Theory and Applications of Satisfiability Testing (SAT'04)*, 2004.

[63] S. Sharma, S. Roy, M. Soos et K. S. Meel, « GANAK : A Scalable Probabilistic Exact Model Counter, » in *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI-19)*, 2019, p. 1169-1176.

[64] M. Sipser, *Introduction to the Theory of Computation*, Third. 2013.

[65] M. Soos, S. Gocht et K. S. Meel, « Tinted, Detached, and Lazy CNF-XOR Solving and Its Applications to Counting and Sampling, » in *Computer Aided Verification*, 2020, p. 463-484.

[66] M. Soos et K. S. Meel, « BIRD : Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting, » in *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI-19)*, 2019, p. 1592-1599.

[67] M. Thurley, « sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP., » in *Proceedings of the 9th international conference on Theory and Applications of Satisfiability Testing (SAT'06)*, 2006, p. 424-429.

[68] D. C. Tsouros, K. Stergiou et C. Bessiere, « Structure-Driven Multiple Constraint Acquisition, » in *Proceedings of the 25th International Conference on Principles and Practice of Constraint Programming (CP 2019)*, 2019, p. 709-725.

[69] D. C. Tsouros, K. Stergiou et P. G. Sarigiannidis, « Efficient Methods for Constraint Acquisition, » in *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, 2018, p. 373-388.

[70] A. Van Gelder, « Autarky Pruning in Propositional Model Elimination Reduces Failure Redundancy, » *J. Autom. Reason.*, t. 23, p. 137-193, 1999.

[71] M. Vavrille, C. Truchet et C. Prud'homme, « Solution sampling with random table constraints, » *Constraints*, p. 1-33, 2022.

[72] T. Walsh, « SAT v CSP, » in *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2020)*, 2000, p. 441-456.

[73] W. Wei et B. Selman, « A New Approach to Model Counting, » in *Theory and Applications of Satisfiability Testing*, 2005, p. 324-339.

[74] N.-F. Zhou, *Combinatorial Search With Picat*, 2014.