République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université A.MIRA-BEJAIA

Faculté Sciences Exactes
Département Informatique
Laboratoire d'Informatique Médicale et des Environnements Dynamiques et Intelligents(LIMED)

THÈSE

EN VUE DE L'OBTENTION DU DIPLOME DE DOCTORAT

Domaine : Mathématiques et Informatique

Filière : Informatique

Spécialité : Intelligence Artificielle et Génie Logiciel

Présentée par

AIT HATRIT Fatima

Thème

Nouveaux algorithmes pour la résolution des problèmes de satisfaction de contraintes

Soutenue le : 12/02/2026

Devant le Jury composé de :

Nom et Prénom	**Grade**		
Mme. BOUKREDERA-BOULAHROUZ Djamila	MCA	Univ. de Bejaia	Présidente
Mr. AMROUN Kamal	Professeur	Univ. de Bejaia	Rapporteur
Mme. DAHMANI - BOUARAB Farida	Professeur	Univ. de Tizi Ouzou	Examinatrice
Mme. EL BOUHISSI- BRAHAMI Houda	MCA	Univ. de Bejaia	Examinatrice
Mme. ALOUI-TIGHIDET Soraya	MCA	Univ. de Bejaia	Examinatrice

Année Universitaire : 2025/2026

Remerciements

Je remercie tout d'abord ALLAH le tout puissant de m'avoir donné la force, le courage et la patience pour mener à terme ce travail.

Je remercie mon directeur de thèse, le professeur AMROUN Kamal, pour sa confiance et ses conseils précieux tout au long de ce travail.

Je remercie les membres du jury pour l'intérêt qu'ils ont porté à notre travail. Je remercie madame BOUKREDERA-BOULAHROUZ Djamila, Maitre de conférences à l'université de Bejaia d'avoir accepté de présider le jury de cette thèse, DAHMANI - BOUARAB Farida, professeur à l'université de Tizi Ouzou, madame EL BOUHISSI Houda , Maitre de conférences à l'université de Bejaia, ainsi que madame TIGHIDET Soraya , Maitre de conférences à l'université de Bejaia de m'avoir fait l'honneur d'accepter de juger ce travail.

Je remercie ma famille pour son soutien inconditionnel et son amour, sans eux je n'y arriverais jamais.

Je remercie mes collègues et amis ainsi que les membres du laboratoire de recherche LIMED, pour leurs encouragements et leur aide précieuse.

Je remercie enfin tous ceux qui ont contribué de près ou de loin à la réalisation de ce travail.

Dédicaces

Je dédie ce modeste travail :

À la mémoire de mon cher père.

À la personne la plus chère au monde, la plus précieuse à mes yeux, ma mère. Sans ta patience, ton soutien et tes encouragements je n'en serai pas là.

À mes très chers frères et sœurs.

À mes nièces et neveux.

À mes chers amis et collègues.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des abréviations

PPC : Programmation Par Contraintes
AC : Arc Consistency
BJ : Backjumping
BT : Backtracking
BTD : Backtracking on Tree-Decomposition
CBJ : Conflict-Directed Backjumping
CNN : Convolutional Neural Networks
CSP : Constraint Satisfaction Problem)
CSOP : Constraint Satisfaction Optimization Problem
DAC : Directional Arc Consistency
DL : Deep Learning
FC : Forward Checking
GAC : Generalized Arc Consistency
GBJ : Graph-Based Backjumping
GHD : Generalized Hypertree Decomposition
GHDA : Generalized Hypertree Decomposition Algorithm
Ghw : Generalized hypertree width
GNN : Graph Neural Networks
GYO : Algorithme de réduction de Graham-Yu-Ozsoyoglu
ILS : Iterated Local Search
JAS : Join Acyclic Solving
LCV : Least Constraining Value
MAC : Maintaining Arc Consistency
MRV : Minimum Remaining Values
NC : Node Consistency
PC : Path Consistency
RL : Reinforcement Learning
SA : Simulated Annealing
SAT : Boolean SATisfiability Problem
TD : Tree Decomposition
TS : Tabu Search
VNS : Variable Neighborhood Search

Liste des Publications

Journaux internationaux

1. Fatima Ait Hatrit & Kamal Amroun . Solving non-binary constraint satisfaction problems using GHD and restart. ITEGAM-JETIA, 11(51), 2025, 72-79. https ://doi.org/10.5935/jetia.v11i51.1415

2. Fatima Ait Hatrit & Kamal Amroun . From Backtracking To Deep Learning : A Survey On Methods For Solving Constraint Satisfaction Problems. ITEGAM-JETIA, 11(51), 2025, 119-126. https ://doi.org/10.5935/jetia.v11i51.1449

Conférences internationales

1. Fatima Ait Hatrit & Kamal Amroun . Enhancing Constraint Satisfaction Problem Solving with a Restart-Nogood-Based Approach. In Proceedings of the First International Conference on Artificial Intelligence, Smart Technologies and Communications (AISTC 2025). Springer Nature, 197, 2025. p. 141.

Conférences nationales

1. Fatima Ait Hatrit & Kamal Amroun . From Search to Learning : Advancements in CSP-Solving Strategies, NCAMAI2025, Skikda, Algérie, 2025.

2. Fatima Ait Hatrit & Kamal Amroun . Exploiting Nogoods for Solving N-airy Constraint Satisfaction Problems (CSP), NCAMAI2025, Skikda, Algérie, 2025.

Introduction générale

La programmation par contraintes s'est imposée comme une approche essentielle en intelligence artificielle et en recherche opérationnelle pour la modélisation et la résolution de problèmes combinatoires complexes. Depuis son introduction dans les années 1970 et la formalisation des problèmes de satisfaction de contraintes (CSP) par Montanari en 1974 [1], ce paradigme repose sur la définition d'un ensemble de variables, de domaines de valeurs possibles pour ces variables, et d'un ensemble de contraintes qui régissent les affectations valides. La tâche centrale consiste alors à trouver une assignation de valeurs à chaque variable de manière à ce que toutes les contraintes imposées soient simultanément respectées.

L'omniprésence des CSP dans divers champs d'application témoigne de leur expressivité et de leur flexibilité. Ils ont été utilisés avec succès pour modéliser et résoudre des défis complexes dans des domaines aussi variés que l'affectation de ressources [2], la planification et la programmation d'activités [3], [4], la planification du trafic aérien [5], [6], la reconnaissance d'objets [7], [8], [9], la médecine [10], [11], ou encore la bio-informatique [12], [13], [14]. La résolution de ces problèmes s'appuie sur une panoplie d'algorithmes spécialisés, allant des méthodes de recherche systématique classiques aux heuristiques avancées, en passant par les techniques de décomposition structurelle et, plus récemment, les méthodes issues de l'apprentissage automatique. Toutefois, la nature généralement NP-complète des CSP continue de stimuler la recherche de techniques toujours plus performantes pour améliorer l'efficacité des solveurs.

Problématique

Parmi les approches prometteuses pour aborder la complexité des CSP, les méthodes de décomposition structurelle se distinguent par leur capacité à exploiter la topologie du problème pour le diviser en sous-problèmes plus gérables. La Décomposition Hypertree Généralisée (GHD) est particulièrement adaptée aux CSP impliquant des contraintes n-aires. Des algorithmes tels que FC-GHD et ses variantes améliorées FC-GHD+NG avec apprentissage de nogoods structurels et FC-GHD+NG+DR avec réordonnancement dynamique des sous-arbres [15] représentent des avancées significatives dans cette direction. L'intégration des nogoods structurels permet d'élaguer l'espace de recherche en évitant la réexploration de branches infructueuses, tandis que le mécanisme de réordonnancement dynamique vise à traiter plus rapidement les sources de conflits.
Cependant, malgré ces améliorations, la performance de ces algorithmes basés sur la GHD reste intrinsèquement liée à deux facteurs critiques. Premièrement,

la qualité de la GHD initiale est primordiale ; or, la construction d'une GHD de largeur minimale est un problème NP-difficile, et les heuristiques pratiques peuvent produire des décompositions sous-optimales. Deuxièmement, même avec une GHD de bonne qualité, l'efficacité de la résolution dépend fortement du choix du nœud racine de l'arbre de décomposition et de l'ordre de parcours initial qui en découle. Un mauvais choix peut conduire l'algorithme à s'engager dans des régions de l'espace de recherche particulièrement difficiles, et le réordonnancement dynamique de FC-GHD+NG+DR, bien qu'utile, réagit localement sans remettre fondamentalement en cause l'ordre global si aucune impasse majeure n'est rencontrée pendant longtemps. Ces sensibilités peuvent entraîner une variabilité importante des performances et un coût de résolution excessif, même pour des problèmes structurellement favorables. L'algorithme peut ainsi rester "piégé" dans une exploration coûteuse.

Objectifs de la Thèse

Face à ces constats, l'objectif principal de cette thèse est d'explorer et de proposer des améliorations significatives aux méthodes de résolution de CSP basées sur la GHD, en s'attaquant spécifiquement aux défis liés à la sensibilité à la qualité de la décomposition et à l'ordre de traitement des clusters. Nous visons à exploiter les avantages de la GHD tout en introduisant des mécanismes pour en accroître la robustesse et l'efficacité pratique. Plus précisément, nous proposons d'intégrer la technique de redémarrage (restart) comme levier pour diversifier la stratégie d'exploration structurelle du problème.

Principale Contribution

La contribution majeure de cette thèse est le développement et l'évaluation d'une approche de résolution de CSP n-aires qui combine la décomposition structurelle par GHD avec une stratégie de redémarrage dynamique et l'apprentissage continu de nogoods. Nous proposons un algorithme, baptisé $Restart-FC-GHD+NG+DR$, qui s'appuie sur les fondations de FC-GHD+NG+DR [15]. Cet algorithme hérite des capacités de filtrage par Forward Checking au niveau des nœuds feuilles et de l'enregistrement des nogoods pour améliorer la détection des conflits.
L'originalité de notre approche réside dans l'application de la technique de redémarrage : la résolution est périodiquement interrompue après un nombre défini de retours en arrière. Ce redémarrage n'est pas une simple reprise, mais une opportunité de relancer la résolution à partir d'un nouvel ordre de traitement des clusters de la GHD, induit par la sélection d'un nouveau nœud racine. Cette exploration de l'arbre de décomposition à partir de différentes racines permet de diversifier la recherche. Crucialement, les nogoods enregistrés avant un redémarrage sont préservés et exploités lors des phases de recherche ultérieures. Cette sauvegarde des nogoods évite la réexploration de combinaisons déjà identifiées comme des échecs, même après un changement de perspective structurelle. Notre approche vise ainsi à améliorer l'efficacité de la résolution des CSP en évitant l'exploration de branches infructueuses et en accélérant le processus de recherche global. Nous avons évalué notre méthode sur des instances classiques de CSP et comparé nos résultats

à ceux de l'algorithme de référence FC-GHD+NG+DR, démontrant son efficacité. L'idée du redémarrage n'est pas nouvelle en résolution de CSP et SAT [16], [17]. Elle consiste à interrompre périodiquement la recherche et à la relancer, souvent avec des paramètres modifiés ou à partir d'un point différent, pour diversifier l'exploration. Dans le contexte spécifique des algorithmes basés sur GHD, une stratégie de redémarrage s'avère particulièrement pertinente pour pallier la sensibilité à l'ordre de traitement des clusters. En permettant de considérer différents nœuds racines potentiels, elle offre une exploration de l'espace de solutions selon des perspectives structurelles variées, tout en capitalisant sur les connaissances acquises.

Organisation de la Thèse

Cette thèse est structurée en trois chapitres principaux :

- Le premier chapitre est consacré au formalisme des problèmes de satisfaction de contraintes (CSP). Il introduit les notions de base, les définitions fondamentales, les représentations graphiques et illustre ces concepts par des exemples concrets.
- Le deuxième chapitre présente une classification détaillée des différentes méthodes de résolution des CSP. Partant des approches de recherche systématique et des techniques de filtrage, il aborde ensuite les stratégies de décomposition structurelle, la recherche locale, et conclut par un aperçu des méthodes basées sur l'apprentissage automatique.
- Le troisième et dernier chapitre est dédié à la présentation de notre contribution principale. Il détaille les fondements des algorithmes basés sur la GHD, en particulier FC-GHD et ses variantes, avant d'introduire et de justifier notre approche intégrant des stratégies de redémarrage. Les algorithmes proposés sont décrits, suivis d'une évaluation expérimentale et d'une analyse des résultats obtenus sur des instances de CSP.
- Enfin, nous conclurons cette thèse en récapitulant les apports majeurs et en proposant des perspectives de recherche pour des travaux futurs dans ce domaine dynamique.

Chapitre 1

Problèmes de Satisfaction de Contraintes

1.1 Introduction

Les Problèmes de Satisfaction de Contraintes (CSP pour Constraints Satisfaction Problem) constituent un paradigme fondamental en intelligence artificielle, en recherche opérationnelle et en informatique. Ils offrent un cadre formel pour modéliser une vaste gamme de problèmes issus de domaines variés, tels que la planification d'horaires [2], l'allocation de ressources [3], le domaine de la télédétection [18] la vision par ordinateur, ou encore la bio-informatique. La puissance des CSP réside dans leur capacité à exprimer des problèmes combinatoires complexes de manière déclarative : un ensemble de variables doit se voir attribuer des valeurs issues de domaines respectifs, tout en satisfaisant un ensemble de contraintes prédéfinies.
Malgré leur apparente simplicité de formulation, la résolution efficace des CSP demeure un défi majeur. La plupart des CSP généraux sont NP-complets [3], ce qui signifie que le temps requis pour trouver une solution peut croître exponentiellement avec la taille du problème. Cette complexité inhérente a stimulé une recherche intensive au fil des décennies, menant au développement d'une multitude de techniques et d'algorithmes visant à optimiser le processus de résolution.
Ce chapitre a pour objectif de poser les fondations nécessaires à la compréhension des travaux présentés dans cette thèse. Nous commencerons par introduire formellement les concepts clés liés aux CSP, incluant leur définition, les types de contraintes et la notion de solution. Par la suite, nous présenterons une revue structurée des principales approches de résolution des CSP, en nous appuyant sur les classifications existantes et en mettant en lumière l'évolution des stratégies, des méthodes de recherche exhaustives aux approches plus modernes intégrant des techniques d'apprentissage. Cette exploration de l'état de l'art nous permettra de positionner nos contributions et de motiver les choix méthodologiques qui seront détaillés dans les chapitres suivants.

1.2 Notions et concepts

Dans cette section, nous allons introduire les définitions de base des problèmes de satisfaction de contraintes (CSP). Ces définitions sont essentielles pour comprendre

les concepts fondamentaux qui sous-tendent la résolution des CSP. Ces définitions seront utilisées tout au long de cette thèse.

Définition 1 ***(Problème de Satisfaction de Contraintes (CSP))***
Un Problème de Satisfaction de Contraintes, initialement formalisé par Montanari [1], est défini comme un triplet (X, D, C), où :

- $X = \{x_1, x_2, \ldots, x_n\}$ *est un ensemble fini de n* ***variables****. Chaque variable représente une entité à laquelle une valeur doit être assignée.*
- $D = \{D_1, D_2, \ldots, D_n\}$ *est un ensemble de n* ***domaines****, où chaque D_i est l'ensemble fini des valeurs possibles que la variable x_i peut prendre.*
- $C = \{c_1, c_2, \ldots, c_m\}$ *est un ensemble fini de m* ***contraintes****. Une contrainte est definit par une paire $(Scope(c_i), Rel(c_i))$:*
  - *$Scope(c_i)$ est un sous ensemble de variables X impliquées dans une contrainte c_i ;*
  - *$Rel(c_i)$ spécifié les combinaisons de valeurs (ou tuples) autorisées pour les variables qu'elles impliquent. L'ensemble des tuples $Rel(c_i)$ est un sous-ensemble du produit cartésien des domaines des variables impliquées dans la contrainte c_i, $Rel(c_i) \subseteq \prod_{x_k \in Scope(c_i)} D(x_k)$, où k est le nombre de variables dans $Scope(c_i)$.*

Définition 2 ***(Variables)***
Une variable X représente un élément auquel on peut attribuer une valeur issue d'un domaine spécifique. Ce domaine correspond à l'ensemble des valeurs admissibles pour la variable.
On parle de ***variable discrète*** *lorsque le domaine est fini ou dénombrable, et de* ***variable continue*** *lorsqu'il est infini et non dénombrable.*
Une variable est dite ***assignée*** *lorsqu'une valeur lui est explicitement attribuée, et* ***libre*** *dans le cas contraire. Si le domaine d'une variable se réduit à une unique valeur, elle est qualifiée de singleton.*

Définition 3 ***(Domaines)***
Le domaine d'une variable correspond à l'ensemble des valeurs que cette variable est susceptible d'adopter. La représentation d'un domaine peut s'effectuer de deux manières :

- ***En extension*** *: toutes les valeurs possibles sont explicitement énumérées. Par exemple, pour une variable x représentant un entier, son domaine pourrait être $D_x = \{1, 2, 3, 4, 5\}$.*
- ***Par intervalle*** *: seules les bornes inférieure et supérieure sont spécifiées. Cette forme de représentation, bien que plus compacte, n'est valide que lorsque le domaine est constitué d'une suite continue de valeurs sans discontinuité. Par exemple, pour une variable y représentant un réel, son domaine pourrait être $D_y = [0, 10]$, indiquant que y peut prendre n'importe quelle valeur entre 0 et 10 inclus.*

Définition 4 ***(Contraintes)***
Une contrainte est une relation qui impose des restrictions sur les valeurs que peuvent prendre certaines variables. Elle peut être exprimée de :

- ***Par intention :*** *: en définissant une relation mathématique entre les variables concernées.*

  Exemple 1 *Soit le CSP $P = (X, D, C)$ tel que :*
  $X = \{x_1, x_2, x_3, x_4\}$
  $D = \{D(x_1) = \{1, 3, 4, 7\}, D(x_2) = \{4, 7\}, D(x_3) = \{4, 7\}, D(x_4) = \{3, 4\}\}$
  $C = \{c_1 \equiv ((x_1 - x_3) \setminus x_4) = 1,\ c_2 \equiv x_4 + x_1 = x_2,\ c_2 \equiv x_2 + x_3 = x_2 + 2\}$

- ***Par extension :*** *: en énumérant les combinaisons de valeurs autorisés ou interdites.*

  Exemple 2 *Soit la la table portant sur trois variables x_1, x_2, x_3*

x_1	x_2	x_3
0	1	4
1	0	5
1	2	0

Définition 5 ***(Arité)***
L'arité d'une contrainte représente le nombre de variables sur la quelle elle est portée. les contraintes peuvent être classées selon leurs arité en différentes catégories :

- ***Unaire*** *: si l'arité est égale à un, la contrainte implique une seule variable.*
- ***Binaire*** *: si l'arité est égale à deux, la contrainte implique deux variables.*
- ***N-aire*** *: si l'arité est égale à n, la contrainte implique plus de deux variables.*

Soit un CSP $P = (X, D, C)$, l'arité de $P = max\{arite(c) | c \in C\}$

Définition 6 ***(Instantiation)***
Une instantiation I est une affectation de valeurs à certaines variables d'un CSP. Soit l'instance CSP $P = (X, D, C)$, et $W \in X$, une instanciation I de W est l'affectation d'une valeur dans $D(x_i)$ à chaque variable $x_i \in W$.
Une instanciation peut être ***partielle****, si elle ne concerne qu'une partie des variables, ou* ***complète****, si toutes les variables sont assignées.*

Définition 7 ***(Instantiation consistante)***
Une instanciation partielle est dite ***consistante*** *(ou cohérente) si elle ne viole* ***aucune*** *des contraintes dont toutes les variables de la portée sont instanciées.*
L'instantiation est dite ***inconsistante*** *si elle viole une ou plusieurs contraintes.*

Définition 8 ***(Solution)***
Une solution d'un CSP $P = (X, D, C)$ est une instanciation ***complète*** *si toutes les variables de X sont assignées et* ***cohérente*** *si toutes les contraintes de C sont satisfaites.*
Formellement, si I est une assignation $I : X \to \bigcup D_i$ telle que pour chaque $x_i \in X$, $I(x_i) \in D_i$, alors I est une solution si pour toute contrainte $C_j \in C$, l'assignation I satisfait C_j.

La résolution d'un CSP peut viser différents objectifs :

1. **Vérification d'existence** : Déterminer si au moins une solution existe.
2. **Énumération de solutions** : Trouver une, plusieurs, ou toutes les solutions possibles.
3. **Optimisation** : Identifier une solution optimale, dans cas des Constraint Satisfaction Optimization Problems (CSOP), la recherche de la solution est selon une fonction de coût ou un critère de préférence.
4. **Approximation** : Trouver une solution quasi-optimale ou une assignation qui satisfait un maximum de contraintes, surtout lorsque le problème est sur-contraint ou que le temps de calcul est limité.

La tâche de déterminer l'existence d'une solution pour un CSP général est NP-complète [19], ce qui rend les techniques de résolution efficaces cruciales pour les applications pratiques.

1.3 Représentation graphique des CSP

La structure des relations entre les variables et les contraintes d'un CSP peut être visualisée à l'aide de structures graphiques. Ces représentations sont non seulement utiles pour la compréhension intuitive du problème, mais elles sont aussi fondamentales pour de nombreuses techniques de résolution, notamment celles basées sur l'analyse de la structure du problème tel que les méthodes de décomposition structurelles.
Les deux principales représentations graphiques sont le graphe de contraintes pour les CSP binaires et l'hypergraphe de contraintes pour les CSP n-aires.

Définition 9 ***(Graphe)***
Un graphe $G = (V, E)$ est composée d'un ensemble de nœuds (sommets) V et d'un ensemble d'arêtes (arcs) E, où chaque arête est un couple ordonné (u, v) de nœuds $u, v \in V$.

1.3.1 Graphe de contraintes

La représentation graphique d'un CSP dont toutes les contraintes sont binaires est sous forme d'un graphe de contraintes, la structure des interactions est représentée par un graphe $G = (V, E)$:

- Les nœuds V correspondent aux variables du CSP : $V = X$.

- Les arête E représentent la relations entre les variables. Une arête de E relie deux nœuds x_i et x_j si et seulement s'il existe une contrainte binaire $C_k \in C$ dont la portée est $\{x_i, x_j\}$.

Exemple 3 *Soit le CSP $P = (X, D, C)$ tel que :*
$X = \{x_1, x_2, x_3, x_4\}$ *et les contraintes sont* $C_1(x_1, x_2)$, $C_2(x_1, x_3)$, $C_3(x_2, x_3)$, $C_4(x_2, x_4)$, $C_5(x_3, x_4)$
Le graphe de contraintes correspondant est représenté dans la Figure 1.1 :

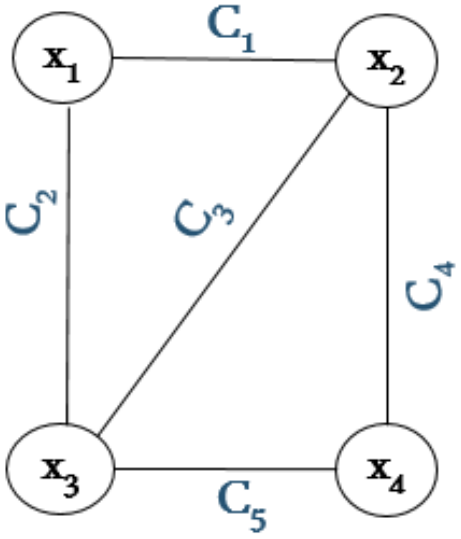

FIGURE 1.1 – Représentation du graphe de contraintes de l'Exemple 3

1.3.2 Hypergraphe de Contraintes

Lorsque le CSP contient des contraintes n-aires, le graphe de contraintes binaire n'est plus suffisant pour capturer toutes les interactions.
Plusieurs représentations existent pour les CSP n-aires, mais l'hypergraphe de contraintes est la plus générale et la plus expressive.
Dans un hypergraphe $H = (V, E_H)$:

- Les nœuds (sommets) V correspondent toujours aux variables du CSP : $V = X$.

- Les hyperarêtes E_H correspondent aux portées des contraintes.
  Chaque hyperarête $e_j \in E_H$ est un sous-ensemble de V représentant l'ensemble des variables impliquées dans la contrainte C_j. Si une contrainte est binaire, son hyperarête correspondante connectera simplement deux nœuds. Si une contrainte est ternaire (par exemple, $C(x_1, x_2, x_3)$), son hyperarête sera l'ensemble $\{x_1, x_2, x_3\}$.

Exemple 4 *Soit le CSP $P = (X, D, C)$ tel que :*
$X = \{x_1, x_2, x_3, x_4, x_5\}$ *et les contraintes sont* $C_1(x_1, x_2), C_2(x_2, x_3, x_4), C_3(x_4, x_5)$.
L'hypergraphe correspondant est représenté dans la Figure 1.2 :

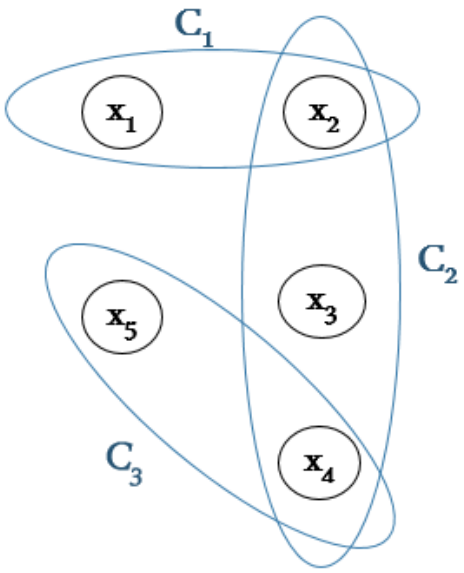

FIGURE 1.2 – Représentation de l'hypergraphe de l'exemple 4

1.4 Complexité des CSP

Le problème de décision pour un CSP est NP-complèt en général [19]. L'espace de recherche, qui correspond au produit cartésien des domaines des variables ($|D_1| \times |D_2| \times \ldots \times |D_n|$), peut être immense, rendant une exploration exhaustive impraticable.
Cependant, certaines classes de CSP peuvent être résolues en temps polynomial. La complexité dépend fortement de la structure du l'hypergraphe de contraintes et de la nature des contraintes [20].

1.5 Exemples d'instances CSP

Pour mieux comprendre la modélisation des CSP, nous allons presenter quelques exemples classiques. Ces illustrations permettent de concrétiser les notions de variables, domaines et contraintes, et de visualiser comment différents problèmes du monde réel ou des casse-têtes logiques peuvent être formulés dans ce cadre.

Exemple 5 ***(Coloriage de catres)***

Le problème du coloriage de carte est un exemple canonique de CSP. L'objectif est d'assigner une couleur à chaque région d'une carte de manière à ce qu'aucune paire de régions adjacentes n'ait la même couleur, en utilisant un nombre limité de couleurs.
Depuis le théorème des quatre couleurs [21], il est établi que toute carte peut être coloriée avec seulement quatre couleurs. Nous allons donc modéliser un exemple de problème de coloriage de carte à l'aide d'un réseau de contraintes basé sur ces quatre couleurs.
Considérons un exemple avec 8 régions à colorier et une palette de 4 couleurs

disponibles.

- ***Variables*** *: Chaque région de la carte est une variable.*
  $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$
  où chaque x_i *représente une région de la carte.*

- ***Domaines*** *: Chaque région peut être colorée avec l'une des 4 couleurs disponibles.*
  $D_i = \{Bleu, Vert, Rouge, Jaune\}$ *pour chaque région* x_i*.*

- ***Contraintes*** *: Pour chaque paire de régions adjacentes* x_i *et* x_j*, il existe une contrainte stipulant que la couleur de* x_i *doit être différente de la couleur de* x_j*.*
  Ces contraintes sont toutes binaires. Pour chaque paire de régions adjacentes, il existe une contrainte stipulant que les couleurs doivent être différentes.
  Dans l'exemple représenté dans la Figure 1.3, les contraintes seraient :

  - x_1 *est adjacente à* x_2, x_3, x_4, x_5
  - x_2 *est adjacente à* x_1, x_3
  - x_3 *est adjacente à* x_1, x_6, x_8
  - x_4 *est adjacente à* x_1, x_5
  - x_5 *est adjacente à* x_1, x_4, x_6, x_7
  - x_6 *est adjacente à* x_3, x_5, x_7, x_8
  - x_7 *est adjacente à* x_5, x_6
  - x_8 *est adjacente à* x_3, x_6

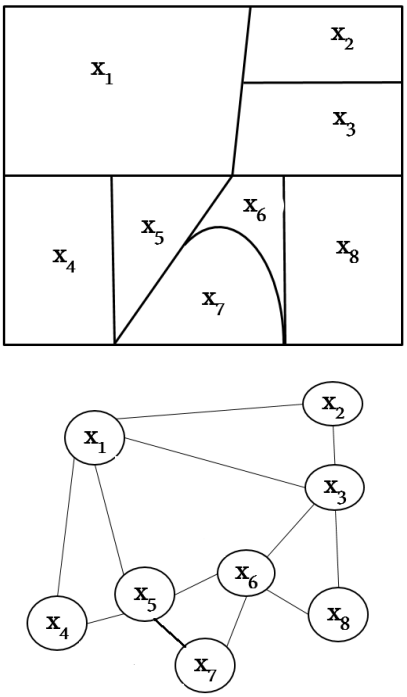

FIGURE 1.3 – Problème de coloriage de carte et son graphe des contraintes associé.

- ***Solution*** *: Une assignation de couleurs à chaque région telle que toutes les contraintes soient satisfaites.*
  Une solution partielle (Figure 1.4) pourrait être :
  $S_p = x_1 = Bleu, x_2 = Vert, x_3 = Rouge, x_4 = Jaune, x_5 = Vert.$

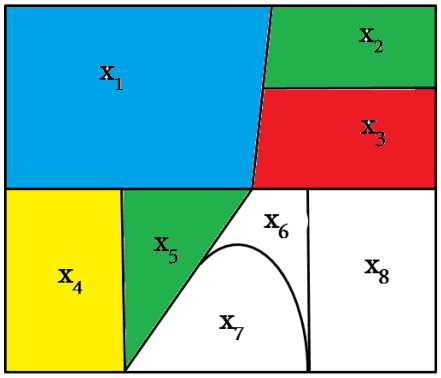

FIGURE 1.4 – Solution partielle du problème de coloriage de carte

Une des solution totale obtenue à partir de la solution partielle est (Figure1.5) :
$S = x_1 = Bleu, x_2 = Vert, x_3 = Rouge, x_4 = Jaune, x_5 = Vert, x_6 = Bleu, x_7 = Rouge, x_8 = Jaune$

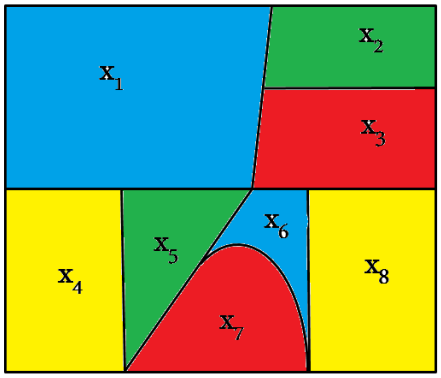

FIGURE 1.5 – Solution totale du problème de coloriage de carte

- ***Espace de recherche*** *: L'espace de recherche pour ce problème est constitué de toutes les assignations possibles des couleurs aux régions.*
  Dans cet exemple, il y a $4^8 = 65,536$ *assignations complètes possibles, illustrant la complexité exponentielle du problème.*

Exemple 6 ***(Problème des N Reines)***

Le problème des N reines est un autre autre exemple classique qui se modélise élgalement comme un CSP. L'objectif est de placer N reines sur un échiquier de $N \times N$ *cases de telle sorte qu'aucune reine ne puisse en attaquer une autre.*
Cela signifie que deux reines ne peuvent pas se trouver sur la même ligne, la même

colonne, ou la même diagonale.

Considérons l'exemple du problème des 4 reines (N=4) sur un échiquier 4×4.

- ***Variables :*** *L'approche consiste à définir une variable pour chaque reine, ou plus efficacement, une variable pour chaque colonne de l'échiquier. Si nous utilisons cette dernière approche, chaque variable* Q_i *représente la ligne sur laquelle la reine de la colonne* i *est placée.*
  Pour N=4 reines : $X = \{Q_1, Q_2, Q_3, Q_4\}$, *où* Q_i *est la ligne de la reine dans la colonne* i

- ***Domaines :*** *Pour chaque variable* Q_i, *le domaine est l'ensemble des lignes possibles sur l'échiquier.*
  Dans notre exemple, pour chaque Q_i, *le domaine est :*
  $D_i = \{1, 2, 3, 4\}$, *où* D_i *représente les lignes de l'échiquier.*

- ***Contraintes :*** *Les contraintes doivent garantir qu'aucune paire de reines* (Q_i, Q_j) *avec* $i \neq j$ *ne se menace :*

  - *Pas sur la même ligne : Puisque* Q_i *représente la ligne de la reine dans la colonne* i, *la contrainte est* $Q_i \neq Q_j$ *pour tout* $i \neq j$.
  - *Pas sur la même diagonale : Deux reines* (Q_i, Q_j) *sont sur la même diagonale si la différence absolue de leurs numéros de colonne est égale à la différence absolue de leurs numéros de ligne. C'est-à-dire,* $|i-j| = |Q_i - Q_j|$ *pour tout* $i \neq j$. *Cette contrainte peut être réécrite comme* $Q_i - Q_j \neq i - j$ *et* $Q_i - Q_j \neq -(i-j)$.
  - *Pour les 4 reines : L'ensemble des contraintes binaires est :*
    Pour toutes paires (i, j) *avec* $1 \leq i < j \leq 4$ *:* $Q_i \neq Q_j$ *Et* $|i-j| \neq |Q_i - Q_j|$

- ***Solution :*** *Une assignation d'une ligne (valeur de 1 à 4) à chacune des variables* Q_1, Q_2, Q_3, Q_4 *de telle sorte que toutes les contraintes ci-dessus soient satisfaites.*
  Une des solution possible pour le problème des 4 reines est $Q_1 = 2, Q_2 = 4, Q_3 = 1, Q_4 = 3$. *Représentée dans la Figure 1.6 :*

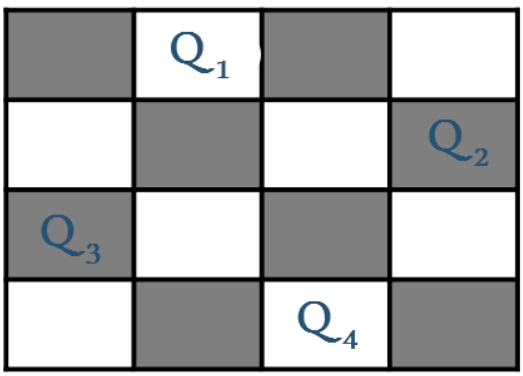

FIGURE 1.6 – Représentation d'une solution du problème des 4 reines

1.6 Méthodes de résolution des CSP

Comme souligné précédemment, la richesse expressive du formalisme CSP permet de modéliser une vaste gamme de problèmes. Conséquemment, il n'existe pas de méthode de résolution universellement optimale, l'efficacité d'une technique par rapport a une autre dépendra des caractéristiques spécifiques du problème à traiter. Au fil des décennies, une diversité d'approches a été développée, s'appuyant notamment sur des concepts fondamentaux tels que l'exploration structurée d'arbres de recherche et l'établissement de divers niveaux de consistance pour élaguer l'espace des solutions.
L'objectif principal d'un solveur CSP est de déterminer si un problème donné admet des solutions et, le cas échéant, de les exhiber. La qualité et le comportement d'un solveur peuvent être caractérisés par plusieurs propriétés essentielles :

- **Complétude :** Un solveur est dit complet s'il est toujours capable de déterminer l'existence ou non d'une solution pour tout CSP qui lui est soumis. Un solveur complet doit etre aussi capable de trouver une solution si elle existe.

- **Correction :** Un solveur est correct ou consistant s'il ne produit que des assignations qui sont des solutions valides au problème, c'est-à-dire respectant toutes les contraintes.

- **Fiabilité :** Un solveur est considéré comme fiable s'il est capable lorsqu'on le lui demande d'énumérer toutes les solutions possibles d'un problème donné.

- **Terminaison :** Un solveur est efficace s'il parvient à garantir la terminaison dans un temps raisonnable, même pour des problèmes de grande taille. L'efficacité est souvent mesurée en termes de temps d'exécution et de mémoire utilisée.

Cependant, la nature NP-complète de la plupart des CSP rend cet idéal difficile à atteindre pour les problèmes de grande taille. Pour pallier à l'exploration combinatoire de toutes les assignations possibles, la majorité des approches de résolution intègrent des techniques de filtrage et utilisent des heuristiques d'ordre de variables ou de contraintes. Ces dernière visent à réduire l'espace de recherche en supprimant des domaines des variables les valeurs qui ne peuvent manifestement pas appartenir à une solution, car elles violeraient certaines contraintes. Ce processus est crucial pour l'efficacité des solveurs.

Les méthodes de résolution des CSP peuvent être globalement classées en plusieurs catégories, bien qu'un certain nombre d'études aient déjà proposé des classifications. Dans [22], l'auteur a classé les techniques de resolution en techniques à domaine fini et techniques à domaine infini. Dans [18], ils ont classé les méthodes de résolution en deux grands groupes : les méthodes de résolution complètes et les méthodes de résolution incomplètes. Autre travaux comme [23], ont classé les méthodes de résolution de CSP en fonction des applications pratiques telles que la planification et l'ordonnancement.
Cette section nous proposons une classification structurée, inspirée de notre classification presenté dans [24] , allant des approches classiques basées sur la recherche exhaustive aux techniques modernes intégrant l'apprentissage automatique.

allant des approches de recherche systématique aux solutions basées sur l'apprentissage profond. Cette section propose une classification structurée, inspirée notamment de notre classification presenté dans [24].

A. **Méthodes de recherche systématique**
   Ces méthodes consistent à explorer l'espace des solutions de manière structurée et rigoureuse afin de trouver des attributions de valeurs qui satisfont l'ensemble des contraintes définies sur les variables. Ces approches reposent généralement sur des stratégies de recherche exhaustive, garantissant ainsi la complétude, c'est-à-dire la certitude de trouver une solution si elle existe.
   Pour améliorer leur efficacité et réduire le temps de calcul, ces méthodes intègrent souvent des optimisations telles que des heuristiques de sélection de variables, des techniques de filtrage, ou des stratégies de retour intelligent.

B. **Méthodes d'inférence et et de filtrage**
   Ces méthodes visent à améliorer l'efficacité des méthodes de recherche systématique, en réduisant l'espace de recherche en éliminant les valeurs des domaines des variables qui ne peuvent pas faire partie d'une solution.
   Elles reposent sur des techniques et stratégies de propagation de contraintes qui se divisent généralement en deux catégories : les stratégies prospectives, qui servent à guider le choix de la prochaine variable à assigner, et les stratégies rétrospectives, qui orientent le choix de la valeur à attribuer à une variable donnée.

C. **Méthodes de décomposition structurelles**
   Ces méthodes exploitent la structure du problème CSP pour diviser un problème complexe en sous-problèmes plus simples, facilitant ainsi la résolution. Elles s'appuient sur la structure du graphe de contraintes associé au CSP. En regroupant les variables et les contraintes en sous-ensembles organisés selon une structure arborescente, ces approches permettent de réduire les interdépendances entre les différentes parties du problème.
   Cette simplification structurelle facilite le traitement computationnel en isolant les interactions locales, ce qui permet d'appliquer des algorithmes plus efficaces sur des sous-problèmes indépendants ou faiblement connectés.

D. **Méthodes de recherche locale**
   Ces méthodes se concentrent sur l'exploration de l'espace des solutions en effectuant des modifications locales sur une solution partielle existante. Elles cherchent à améliorer progressivement la solution partielle en opérant des ajustements locaux plutôt que d'explorer l'ensemble de l'espace de recherche, elles se concentrent sur des voisinages de solutions existantes, ce qui les rend particulièrement efficaces dans les cas où l'espace de recherche est vaste et fortement contraint, rendant les méthodes exhaustives impraticables. Elles offrent une solution flexible et scalable, bien adaptée aux problèmes réels où la recherche d'une solution optimale absolue est moins importante que l'obtention rapide d'une solution satisfaisante.

E. **Méthodes basées sur l'apprentissage**
   Ces méthodes intègrent des techniques d'apprentissage automatique pour améliorer la résolution des CSP. Ces approches utilisent des modèles de deep learning pour prédire les valeurs des variables ou des contraintes, optimisant ainsi

le processus de recherche et améliorant la qualité et la rapidité de résolution. Elles peuvent être utilisées pour apprendre des heuristiques de recherche, des stratégies de filtrage, ou même pour générer des modèles prédictifs qui guident la recherche. Ces approches sont particulièrement prometteuses dans le contexte des CSP complexes et dynamiques, où les méthodes traditionnelles peuvent rencontrer des difficultés.

1.7 Conclusion

Dans ce chapitre, nous avons introduit les concepts fondamentaux des CSP, en définissant les variables, les domaines, les contraintes et les solutions. Nous avons également exploré la représentation graphique des CSP à travers le graphe de contraintes et l'hypergraphe de contraintes, ainsi que la complexité inhérente à ces problèmes.

Dans le chapitre suivant, nous présenterons en détail ces différentes familles de méthodes de resolution que nous avons citées, les algorithmes de résolution utilisés en mettant l'accent sur leurs principes de fonctionnement, leurs avantages et inconvénients.

Chapitre 2

Méthodes de Résolution des CSP

2.1 Introduction

La résolution efficace des CSP, dont les fondements ont été établis au chapitre précédent, constitue un défi computationnel majeur qui a stimulé une recherche algorithmique intensive et continue. Face à la complexité inhérente, souvent NP-complète, des CSP, une multitude d'approches ont été conçues, chacune avec ses propres paradigmes, avantages et limitations.
Ce chapitre propose une exploration approfondie et structurée de l'ensemble des techniques de résolution disponibles, en suivant une classification qui reflète l'évolution des stratégies, depuis les méthodes de recherche systématique jusqu'aux solutions modernes basées sur l'apprentissage profond, telle que nous l'avons proposées dans notre article [24].

L'objectif est de fournir une compréhension détaillée des mécanismes sous-jacents à chaque famille de méthodes, d'analyser leurs performances, et de mettre en lumière les contextes dans lesquels elles s'avèrent les plus pertinentes. Nous examinerons comment ces techniques abordent l'exploration de l'espace de recherche, la gestion des contraintes, et l'optimisation du processus de découverte de solutions. Cette revue exhaustive servira de fondement pour apprécier la pertinence et l'originalité des contributions relatives aux techniques de décomposition structurelle et de redémarrage qui seront présentées ultérieurement dans cette thèse.

Nous organiserons notre chapitre en cinq grandes catégories, pour chaque catégorie, nous décrirons les algorithmes clés, leurs principes de fonctionnement, en enrichissant et en détaillant les concepts esquissés dans l'article de référence :

- Méthodes de recherche systématique
- Techniques d'inférence et de filtrage
- Techniques basées sur décomposition structurelle
- Techniques basées sur la recherche locale
- Solutions basées sur l'apprentissage profond

2.2 Méthodes de Recherche Systématique

Les méthodes de recherche systématique, également qualifiées d'algorithmes énumératifs, constituent une base dans la résolution des CSP. Leur caractéristique principale est la complétude : elles explorent l'espace des solutions potentielles de manière exhaustive et structurée, garantissant ainsi de trouver une solution si elle existe, ou de prouver irréfutablement qu'aucune solution n'est possible.

Ces algorithmes procèdent typiquement en construisant une solution de manière incrémentale, variable par variable, et en vérifiant la cohérence de l'assignation partielle à chaque étape. L'exploration de l'espace de recherche peut être visualisée comme un parcours d'un arbre, où chaque branche représente un choix d'assignation pour une variable.

2.2.1 Heuristiques d'Ordre

L'efficacité des algorithmes de recherche systématique est extrêmement sensible à l'ordre dans lequel les variables sont sélectionnées pour être instanciées et à l'ordre dans lequel les valeurs sont essayées pour chaque variable. De bonnes heuristiques peuvent réduire drastiquement la taille de l'arbre de recherche exploré.

2.2.1.1 Heuristiques de Sélection des Variables

Les heuristiques de sélection des variables visent à choisir la variable à instancier en premier, en tenant compte de la structure du problème et des contraintes. Ces heuristiques peuvent être statiques donc fixées, l'ordre des variables est déterminé avant le début de la recherche, ou dynamiques, où l'ordre est ajusté au fur et à mesure de la recherche dynamiquement à chaque étape en fonction de l'état courant de la recherche.

- **Minimum Remaining Values (MRV)** : Choisir la variable non encore instanciée qui a le plus petit nombre de valeurs légales restantes dans son domaine courant [25]. L'intuition est que si une variable a peu d'options, il vaut mieux savoir rapidement si l'une d'elles mène à une solution ou à une impasse.
  Souvent, MRV est utilisé en premier, et deg est utilisé pour départager les égalités.
- **Degree Heuristic (Maxdeg)** : Choisir la variable non encore instanciée qui est impliquée dans le plus grand nombre de contraintes avec d'autres variables non encore instanciées [26]. L'idée est de s'attaquer aux variables les plus "contraignantes" en premier.
- **dom/deg [27] :** Une heuristique qui sélectionne la variable minimisant le ratio taille du domaine / degré.
- **dom+deg [28] :** Une heuristique qui sélectionne la variable qui a le domaine courant de taille minimum, et dans le cas d'égalité choisir celle qui a le degré maximum.

2.2.1.2 Heuristiques de Sélection des Valeurs

Les heuristiques de sélection des valeurs visent à choisir la valeur à affecter à la variable courante, en tenant compte de l'impact potentiel sur les autres variables et sur la structure du problème. Ces heuristiques peuvent également être statiques ou dynamiques.
L'idée générale est de choisir les valeurs qui ont le plus de chances de mener à une solution.

- **Least Constraining Value (LCV)** : Choisir la valeur qui élimine le moins de valeurs des domaines des variables voisines non encore instanciées [29]. Cela laisse un maximum de flexibilité pour les assignations futures.

- **Min-Conflicts [28]** : Choisir la valeur qui minimise le nombre de conflits avec les variables déjà assignées ou qui minimise le nombre de contraintes violées si on testait cette valeur. Cette heuristique est souvent utilisée dans les algorithmes de recherche locale, mais peut également être appliquée dans le contexte de la recherche systématique.

2.2.2 Retour-Arrière Chronologique (Backtrack)

Le *Backtracking* (BT) est considéré comme l'algorithme de base le plus répandu pour la recherche systématique [30], [31] .
Son principe est une recherche en profondeur d'abord (Depth-First Search) dans l'arbre des assignations possibles. Il repose sur une procédure récursive qui tente d'étendre progressivement une affectation partielle cohérente en attribuant des valeurs aux variables non encore instanciées. À chaque étape, une variable est choisie et une valeur de son domaine lui est assignée, à condition que cette affectation soit compatible avec les variables déjà instanciées. Si une telle compatibilité est vérifiée, l'algorithme procède à l'instanciation de la variable suivante. En revanche, si l'affectation viole une contrainte, une autre valeur est essayée. Si aucune valeur du domaine de la variable ne permet de respecter les contraintes, un *retour arrière* est effectué vers la variable précédemment instanciée afin de tester d'autres alternatives.
Ce processus continue jusqu'à ce que toutes les variables sont affectées de manière cohérente donc qu'une solution complète soit trouvée, ou jusqu'à ce qu'il soit prouvé qu'aucune solution n'est possible, toutes les combinaisons ont échoué.

Dans l'algorithme BT (Algorithme 1), les variables sont instanciées séquentiellement, selon un ordre prédéfini ou dynamiquement choisi. Pour la variable courante x_k, on lui assigne une valeur d_k de son domaine D_k. On vérifie si cette nouvelle assignation ($x_k = d_k$) est cohérente avec toutes les assignations précédentes ($x_1 = d_1, \ldots, x_{k-1} = d_{k-1}$), c'est-à-dire si toutes les contraintes impliquant uniquement des variables déjà instanciées sont satisfaites.
Si l'assignation est cohérente :

- Si toutes les variables sont instanciées ($k = n$), une solution est trouvée. L'algorithme peut s'arrêter ou continuer pour trouver d'autres solutions.

- Sinon ($k < n$), on passe à la variable suivante x_{k+1}.

Si l'assignation n'est pas cohérente, ou si toutes les valeurs du domaine de x_k ont été essayées sans succès (impasse au niveau de x_k), l'algorithme effectue un retour-arrière chronologique : il revient à la variable précédente x_{k-1} et essaie une autre valeur pour celle-ci. Si x_{k-1} n'a plus d'autres valeurs à essayer, on remonte encore, et ainsi de suite.
Si l'on revient à la première variable x_1 et que toutes ses valeurs ont été épuisées, le problème n'a pas de solution.

Algorithm 1 Algorithme Backtrack

Input : Un CSP $P = (X, D, C)$, et une affectation A. Initialement $A = \emptyset$.
Output : Une solution du CSP si le CSP est consistant.

```
1: if A est non consistante then
2:   return False
3: else
4:   if A est une instanciation totale then
5:     return True
6:   else
7:     choisir une nouvelle variable X_i de X
8:     for Chaque valeur d_i ∈ D_i do
9:       if Backtrack(P = (X, D, C), A ∪ X_i = d_i) then
10:        return True
11:      end if
12:    end for
13:  end if
14: end if
```

L'algorithme BT présente l'avantage de ne nécessiter qu'un espace mémoire linéaire, mais la complexité du BT est bornée par la taille de l'espace de recherche, qui est exponentielle en $O(m \cdot d^n)$, où n est le nombre de variables, d la taille maximale des domaines, et m le nombre de contraintes (pour le coût de vérification des contraintes à chaque nœud) [32].
Néanmoins, il s'avère plus efficace que la stratégie "generate & test", dans la mesure où il permet d'éliminer précocement des sous-arbres entiers de l'espace de recherche dès qu'une instanciation partielle est détectée comme incohérente.

Malgré cet avantage, le backtracking souffre de certaines limites, notamment le phénomène de *thrashing*, où les mêmes échecs peuvent être redécouverts à plusieurs reprises sans progrès significatif. Pour pallier à ces inconvénients, plusieurs raffinements ont été proposés. On distingue notamment les approches de type *look-back*, comme le retour arrière non chronologique (*Backjumping*)[33], qui revient directement à la variable responsable du conflit, et les approches de type *look-ahead*, comme le *Forward Checking* [25], qui anticipent les conflits en filtrant les domaines des variables non encore instanciées après chaque affectation.

2.2.3 Algorithmes avec Retour-Arrière Non Chronologique

Pour pallier les inefficacités du retour-arrière purement chronologique, des techniques de retour-arrière plus intelligentes ont été développées. Celles-ci tentent

d'identifier plus précisément la source du conflit pour effectuer des sauts plus importants dans l'arbre de recherche.

A. Backjumping (BJ)

l'algorithme Backjumping est proposé par Gaschnig [33], le BJ est une amélioration de BT qui, en cas d'échec sur la variable courante x_k, ne revient pas systématiquement à x_{k-1}. Il analyse la raison de l'échec, trouver laquelle des variables précédemment assignées entre en conflit avec toutes les valeurs de D_k et saute directement à la variable la plus récente, disons x_j $(j < k)$, responsable de l'échec. Les instanciations des variables entre x_j et x_k sont ainsi ignorées, car elles ne sont pas la cause du conflit actuel.

Le BJ est plus efficace que le BT car il évite de revisiter des parties de l'arbre de recherche qui sont vouées à l'échec pour les mêmes raisons déjà identifiées. Il est particulièrement utile dans les problèmes où les conflits sont fréquents et où les domaines des variables sont larges.vvv

B. Conflict-Directed Backjumping (CBJ)

Algorithm 2 Algorithme Conflict-Directed Backjumping

Procédure CBJ(A)

1: **if** toute les variables de A sont assignées **then**
2: **return** Solution A
3: **else**
4: choisir une variable non assignée x et définir $S := \emptyset$
5: **for** chaque valeur possible v de x **do**
6: $x := v$
7: $cs \leftarrow$ CBJ_evaluate$(A \cup \{x := v\})$
8: **if** cs est vide **then**
9: $cs \leftarrow$ CBJ$(A \cup \{x := v\})$
10: **end if**
11: **if** $x \notin cs$ **then**
12: désassigner x
13: **return** cs /* Backjumping */
14: **end if**
15: $S \leftarrow S \cup cs$
16: **end for**
17: **return** $S - \{x\}$ /* Échec pour toutes les valeurs de x */
18: **end if**

Fonction $CBJ\_evaluate(A)$

1: **if** A est consistant **then**
2: **return** $\emptyset$
3: **else**
4: choisir une contrainte c violée par A
5: **return** domaine(c)
6: **end if**

Une forme plus sophistiquée, le CBJ [34], maintient pour chaque variable x_k un ensemble de conflit "conflict set" qui contient les variables assignées précédem-

ment x_i ($i < k$) qui sont en conflit avec au moins une valeur du domaine de x_k. Lorsque le domaine de x_k devient vide, une impasse est atteinte pour x_k, l'algorithme saute en arrière à la variable la plus récente présente dans le "conflict set" de x_k. Le "conflict set" est mis à jour récursivement lors des retours-arrière.

L'algorithme *CBJ* (Algorithme 2) tente l'extension d'une affectation A initialement vide. La procédure `CBJ_evaluate(A)` retourne un ensemble de variables en conflit avec une affectation invalide, appelé *conflict set* (*cs*). Lorsqu'une valeur v est assignée à une variable x, si $A \cup \{x := v\}$ est inconsistant, `CBJ_evaluate` identifie les variables responsables de cette incohérence. Si aucun conflit n'est détecté, la procédure `CBJ` est appelée récursivement. Lorsque toutes les valeurs de x échouent, l'algorithme effectue un saut arrière vers la variable la plus récente impliquée dans les conflits. Les variables intermédiaires sont alors désassignées, et leurs ensembles de conflits réinitialisés.

Le CBJ est généralement plus efficace que le BJ simple car il identifie plus précisément les causes des conflits.

C. **Graph-Based Backjumping (GBJ)**
Cette variante de BJ exploite la structure du graphe de contraintes. En cas d'échec sur une variable x_k, elle ne revient qu'aux variables précédentes qui sont connectées à x_k dans le graphe de contraintes, ses "parents" dans une certaine structure d'ordre [35].

Ces techniques de retour-arrière non chronologique permettent d'éviter d'explorer des parties de l'arbre de recherche qui sont vouées à l'échec pour les mêmes raisons déjà identifiées. Elles améliorent ainsi l'efficacité de la recherche en réduisant le nombre de nœuds explorés et en évitant les répétitions inutiles.

2.2.4 Algorithmes avec Filtrage Avant

Pour améliorer la détection précoce des conflits, les algorithmes de recherche systématique peuvent être combinés avec des techniques de filtrage qui propagent les effets des assignations courantes aux variables futures.

2.2.4.1 Forward Checking (FC)

Le Forward Checking (FC) [25][36] c'est une procédure de recherche de solution pour les CSP. Il est considéré comme l'une des techniques de prévision d'échec "look-ahead" les plus simples. Son objectif principal est de propager l'information des variables déjà assignées vers les variables futures non encore instanciées afin d'anticiper et de prévenir les affectations qui mèneraient inévitablement à un échec plus tard dans la recherche.

L'algorithme (Algorithme 3) fonctionne de manière récursive. Lorsqu'une valeur est assignée à la variable courante x_i est instanciée avec une valeur v, FC vérifie immédiatement les conséquences de cette affectation sur les domaines des variables futures x_j non encore instanciées qui partagent une contrainte avec la variable courante x_i. Pour chaque x_j, FC supprime de son domaine D_j toutes les valeurs qui sont incompatibles avec $x_i = v$.

Si le domaine d'une variable x_j devient vide, alors la valeur v pour x_i est rejetée, et l'algorithme effectue un retour-arrière. Les valeurs supprimées doivent être restaurées lors du backtracking.

L'algorithme FC est une amélioration du BT, car il permet de détecter les conflits plus tôt et d'élaguer l'espace de recherche. Cependant, il ne garantit pas la consistance d'arc, et certaines solutions peuvent être manquées si des valeurs sont supprimées trop tôt. Malgré cela ces algorithmes fournit un cadre algorithmique robuste dans plusieurs domaines [37].

FC est généralisé aux contraintes n-aires avec l'algorithme nFC [38], il distingue les contraintes impliquant la variable courante et une variable future $C_{n,f}$ (NFC1) et celles impliquant des variables passées et une variable future $C_{p,f}$ (NFC2).

Algorithm 3 Algorithme Forward Checking

Entrée : Une instance I comprenant un ensemble de variables X, de domaines D et de contraintes C
Sortie : Une affectation cohérente ou un échec

```
1: if isFull(I) then
2:   return I comme solution
3: else
4:   Sélectionner une variable x_i ∈ X \ vars(I)
5:   for chaque valeur v_i ∈ D(x_i) do
6:     Affecter x_i ← v_i
7:     if Check-Forward(I, (x_i = v_i)) then
8:       ForwardChecking(I ∪ {(x_i = v_i)})
9:     else
10:      for chaque variable x_j ∉ vars(I) telle que ∃c_ij ∈ C do
11:        Restaurer D(x_j)
12:      end for
13:    end if
14:  end for
15: end if
```

Fonction Check-Forward

```
1: for chaque variable x_j ∉ vars(I) telle que ∃c_ij ∈ C do
2:   for chaque valeur v_j ∈ D(x_j) telle que (v_i, v_j) ∉ c_ij do
3:     Supprimer v_j de D(x_j)
4:     if D(x_j) = ∅ then
5:       return false
6:     end if
7:   end for
8: end for
9: return True
```

2.2.5 Algorithmes avec Retour-Arrière avec mémorisation

Si les sauts non chronologiques et le filtrage avant améliorent significativement le processus de recherche, ils ne préviennent pas complètement le phénomène de *thrashing*, où les mêmes raisons de conflit sont redécouvertes dans différentes parties de l'arbre de recherche. Pour adresser cette limitation fondamentale, les algorithmes de recherche systématique peuvent être dotés de capacités de mémorisation. Ces techniques, collectivement appelées apprentissage de conflits ou enregistrement de nogoods, visent à analyser les causes d'un echec pour en extraire une connaissance réutilisable, une leçon qui permettra d'éviter de répéter la même erreur.
Il est crucial de distinguer cette forme d'apprentissage, qui s'inscrit dans un cadre de recherche systématique et complet, des mécanismes de mémoire heuristique utilisés dans la recherche locale. Ici, la mémorisation ne sert pas à guider une exploration stochastique, mais à enregistrer des déductions logiques sous la forme de nouvelles contraintes implicites, garantissant ainsi la préservation de la correction et de la complétude de l'algorithme.

Définition 10 ***(Conflit)***
Un conflit est une situation où une affectation partielle d'un CSP ne peut pas être étendue en une solution complète. Un conflit est souvent identifié lorsqu'une variable ne peut pas être assignée à une valeur sans violer une contrainte.

Définition 11 ***(Good/Nogood)***
Un nogood est un ensemble d'assignations de variables qui ne peut pas être étendu en une solution complète du CSP. En d'autres termes, si un nogood est rencontré, il indique que toute tentative d'assignation des variables dans cet ensemble mènera à un échec. Inversement, un good est un ensemble d'assignations qui peut être étendu en une solution complète.

Les algorithmes de retour arrière avec mémorisation sont des techniques utilisées pour accélérer la recherche dans les CSP en évitant les répétitions inutiles, ils ne peuvent pas être utilisés seuls, mais doivent être combinés avec des techniques de filtrage et de retour arrière non chronologique pour être efficaces.
Cependant, ces algorithmes peuvent être coûteux en termes de mémoire, car ils nécessitent de stocker les nogoods. De plus, la gestion de la mémoire et la mise à jour des nogoods peuvent ajouter une complexité supplémentaire à l'algorithme. Malgré ces défis, les algorithmes de retour arrière avec mémorisation sont largement utilisés dans la pratique pour résoudre efficacement les CSP complexes.
Parmi les algorithmes de retour arrière avec mémorisation, on trouve les algorithmes de Nogood Recording [39], qui enregistrent les nogoods rencontrés lors de la recherche, l'algorithme Dynamique Backtracking [40], et autres variantes du Backjumping [41].

2.3 Techniques d'Inférence et de Filtrage

Dans cette section, nous présenterons la notion de filtrage des CSPs, qui joue un rôle fondamental et important dans le processus de résolution de CSPs.
Dans les techniques d'Inférence et de filtrage l'objectif n'est pas de trouver une

solution, mais de réduire l'espace de recherche de solution enéliminant les valeurs qui ne figurent pas dans la solution, afin d'éviter l'exploitation des régions de l'espace de recherche qui ne contiennent pas de solution, accélérer le temps de recherche et réduire le temps de résolution, cella en utilisant des techniques de consistance locale ou globale et de propagation de contraintes [42] [43] qui est atteinte quand la valeur d'un domaine ou un tuple d'une relation ne pourra pas etre supprimé. Ces techniques de filtrage sont souvent appliquées avant ou pendant la recherche systématique pour réduire les domaines des variables, ce qui peut rendre la recherche plus efficace. Elles peuvent être classées en deux catégories principales : les techniques de consistance locale et les techniques de consistance globale.

Définition 12 ***(Consistance de Nœud (NC))***
Une instance d'un CSP est dite consistante en nœud (Node Consistency NC) si, pour chaque variable X_i, il existe au moins une valeur $d \in D_i$ telle que l'affectation $X_i = d$ satisfait toutes les contraintes unaires qui s'appliquent à cette variable.

Le principe de la consistance de nœud consiste à éliminer de chaque domaine D_i les valeurs d qui violent une contrainte unaire associée à X_i. Ainsi, seules les valeurs compatibles avec les contraintes unaires sont conservées dans le domaine de chaque variable.

2.3.1 Consistance de Arc

Définition 13 ***(Consistance d'Arc (AC))***
Une instance d'un CSP est dite consistante en arc (Arc Consistency (AC)) si, pour chaque contrainte binaire $C(X_i, X_j)$, pour chaque valeur $d_i \in D_i$ de la variable X_i, il existe au moins une valeur $d_j \in D_j$ de la variable X_j telle que l'affectation $(X_i = d_i, X_j = d_j)$ satisfait la contrainte $C(X_i, X_j)$.

Le principe de la consistance d'arc consiste à éliminer de chaque domaine D_i les valeurs d_i qui ne sont pas compatibles avec au moins une valeur d_j du domaine D_j de la variable X_j associée à la contrainte binaire. Ainsi, seules les valeurs compatibles avec les contraintes binaires sont conservées dans le domaine de chaque variable.

La consistance d'arc a été implémentée en algorithme, le premier algorithme est $AC1$ (Algorithme 4) proposé par Mackworth [44]. L'algorithme consiste à éliminer toutes les valeurs des domaines des variables qui ne satisfais pas la propriété d'arc consistance.

Bien que l'algorithme $AC1$ garantie la consistance d'arc, mais il se trouve qu'il a des inconvénients qui sont que toutes les contraintes sont réviser même si la valeur supprimer n'influence pas sur ces contraintes.

Algorithm 4 Algorithme AC1

```
repeat
    modification ← false
    for chaque contrainte C_k do
        modification ← Reviser(C_k) ∨ modification
    end for
until not modification
```

Fonction $Reviser(C_k)$

```
C_k = (X_i, X_j)
for tout d_i ∈ D_i do do
    if ∄d_j ∈ D_j|(X_i, X_j) ∈ R_k then
        supprimer d_i de D_i
        modification ← True
    end if
end for
return modification
```

Une amélioration à $AC1$ qui est $AC3$ (Algorithme 5) a été proposée par Mackworth [44], qui fait que seules les contraintes touchées par la suppression seront re-réviser par la fonction $Reviser()$, ces dernières sont mémorisées dans une file L.

Algorithm 5 Algorithme AC3

```
L ← (X_i, X_j), i ≠ j
while L ≠ ∅ do
    choisir et supprimer de L un couple (X_i, X_j)
    if Reviser((X_i, X_j)) then
        L ← L ∪ {(X_k, X_j)}/∃ une contrainte liant X_k et X_i
    end if
end while
```

Nous avons aussi les successeurs de $AC3$, $AC4$ [45], $AC5$ [46], $AC6$ [47], $AC2000$ et $AC2001$ [48] et $AC2001/3.1$ [49] développés par Bessière et Régin , $AC2001-OP$ [50].

2.3.2 Consistance d'arc directionnelle

La consistance d'arc directionnelle (Directional Arc Consistency DAC) affaiblit la consistance d'arc en révisant l'arc dans une seule direction à condition que les variables soient dans l'ordre $(X_1, \cdots, X_n)$.

Définition 14 ***(Consistance d'Arc Directionnelle (DAC))***

Une instance d'un CSP est dite consistante en arc directionnelle (DAC) si, pour chaque contrainte binaire $C(X_i, X_j)$, pour chaque valeur $d_i \in D_i$ de la variable X_i, il existe au moins une valeur $d_j \in D_j$ de la variable X_j telle que l'affectation $(X_i = d_i, X_j = d_j)$ satisfait la contrainte $C(X_i, X_j)$ et que l'affectation $(X_j = d_j, X_i = d_i)$ ne satisfait pas la contrainte $C(X_i, X_j)$.

Le principe de la consistance d'arc directionnelle consiste à éliminer de chaque domaine D_i les valeurs d_i qui ne sont pas compatibles avec au moins une valeur d_j du domaine D_j de la variable X_j associée à la contrainte binaire, et qui ne satisfont pas la contrainte dans l'autre sens. Ainsi, seules les valeurs compatibles avec les contraintes binaires dans un sens sont conservées dans le domaine de chaque variable.

2.3.3 Consistance de Chemin (Path Consistency - PC)

Définition 15 ***(Consistance de Chemin (PC))***
Une instance d'un CSP est dite consistante en chemin (PC) si, pour chaque contrainte binaire $C(X_i, X_j)$, pour chaque valeur $d_i \in D_i$ de la variable X_i, il existe au moins une valeur $d_j \in D_j$ de la variable X_j telle que l'affectation $(X_i = d_i, X_j = d_j)$ satisfait la contrainte $C(X_i, X_j)$ et que pour chaque variable X_k qui est reliée à X_i par une contrainte binaire, il existe au moins une valeur $d_k \in D_k$ de la variable X_k telle que l'affectation $(X_k = d_k, X_j = d_j)$ satisfait la contrainte entre X_k et X_j.

Le principe de la consistance de chemin consiste à éliminer de chaque domaine D_i les valeurs d_i qui ne sont pas compatibles avec au moins une valeur d_j du domaine D_j de la variable X_j associée à la contrainte binaire, et qui ne satisfont pas la contrainte entre X_k et X_j. Ainsi, seules les valeurs compatibles avec les contraintes binaires dans un sens et dans l'autre sens sont conservées dans le domaine de chaque variable.

Le PC peut rendre explicites des contraintes implicites. Il est plus rarement utilisé en pratique que l'AC en raison de son coût, sauf pour des problèmes spécifiques ou des graphes denses. Parmi les algorithmes de consistance de chemin proposé on trouve : $PC-1$ et $PC-2$ [44], $PC-3$ [45], $PC-4$ [51], etc.

2.3.4 K-consistance

Définition 16 ***(k-consistance (K-consistency))***

Une instance d'un CSP est dite k-consistante si, pour chaque contrainte k-aire $C(X_1, X_2, \ldots, X_k)$, pour chaque valeur $d_i \in D_i$ de la variable X_i, il existe au moins une valeur $d_j \in D_j$ de la variable X_j telle que l'affectation $(X_1 = d_1, X_2 = d_2, \ldots, X_k = d_k)$ satisfait la contrainte $C(X_1, X_2, \ldots, X_k)$.

Le principe de la k-consistance consiste à éliminer de chaque domaine D_i les valeurs d_i qui ne sont pas compatibles avec au moins une valeur d_j du domaine D_j de la variable X_j associée à la contrainte k-aire. Ainsi, seules les valeurs compatibles avec les contraintes k-aires sont conservées dans le domaine de chaque variable.

La k-consistance [52] est une généralisation de la consistance d'arc [45], de la consistance de chemin [51] et de la consistance d'arc directionnelle. En effet, la consistance d'arc est une 1-consistance, la consistance de chemin est une 2-consistance et la consistance d'arc directionnelle est une 2-consistance.

Définition 17 ***(forte k-consistance)***
Un CSP est fortement k-consistant (Strong k-Consistency) [20] s'il est j-consistant pour tout $j \leq k$.

Si un problème à n variables est fortement *n-consistant*, une solution peut être trouvée sans backtracking. Cependant, atteindre ce niveau de consistance est généralement aussi difficile que de résoudre le problème original.

2.3.5 Consistance d'Arc Généralisée (GAC)

Initialement, le processes de propagation de contraintes était définit pour des CSP binaires, car la définition classique de l'arc-consistance est intrinsèquement liée aux contraintes binaires. Or, de nombreux problèmes du monde réel sont modélisés à l'aide de contraintes n-aires.
Pour appliquer un niveau de filtrage similaire et aussi puissant à ces problèmes plus généraux, il est nécessaire d'étendre le concept de l'arc-consistance à l'arc consistance generalisée.

Définition 18 ***(Support)***
Un support d'une valeur d_i *(de la variable* X_i*) pour la contrainte* C *est une affectation complète aux autres variables* $X_{j\,j\neq i}$ *concernées par* C*, telle que l'ensemble de valeurs* $(d_1, ..., d_k)$ *satisfait la contrainte* C *avec* $X_i = d_i$.

Définition 19 ***(Consistance d'Arc Généralisée (GAC))***

Une instance d'un CSP est dite consistante en arc généralisée (GAC) si, pour chaque contrainte C *portant sur un ensemble de variables* $X_1, ..., X_k$*, et pour chaque variable* X_i *impliquée dans* C*, chaque valeur* $d_i \in D_i$ *possède au moins un support dans la contrainte* C.

Le principe de la consistance d'arc généralisée consiste donc à supprimer des domaines toute valeur qui ne participe à aucune solution valide d'une contrainte, quelle que soit son arité. Ainsi, seuls les valeurs conservée si et seulement si il existe au moins un support pour les autres variables telle que la contrainte soit satisfaite.

2.3.6 Algorithme GAC 2001

GAC 2001 (Algorithme 6) [49] est une généralisation de $AC2001$ pour le filtrage des CSPs n-aires en se basant sur la consistance d'arc généralisée définie précédemment. GAC est exécuté sur les CSPs dont les tuples des relations sont supposés être ordonnés tel que la variable $Last((X_i, a), C_j)$ initialisée à Nil va contenir le dernier support de chaque couple (X_i, a) pour une contrainte C_j, ces valeurs seront vérifiées premièrement lors de la recherche du support d'affectation si elle existe toujours. Dans ce cas, on peut dire que l'affectation possède un support, sinon la recherche est effectuée à partir de la valeur suivante qui est retournée par la fonction $succ()$.

Algorithm 6 Algorithme GAC 2001

for chaque variable X_i **do**
  for chaque valeur a de D_i **do**
    for chaque contrainte C_j **do**
      $Last((X_i, a), C_j) = NIL$
    end for
  end for
end for
$Q = \{(X_i, C_j) | C_j \in C \text{ et } X_i \in Scope(C_j)\}$
while Q non vide **do**
  choisir (X_i, C_j) de Q
  $Q = Q - \{X_i, C_j\}$
  if $REVISE2001(X_i, C_j)$ **then**
    $Q = Q \cup \{(X_k, C_m) | C_m \in C, X_i, X_k \in Scope(C_m) \text{ et } k \neq i \text{ et } j \neq m\}$
  end if
end while

Procédure $REVISE2001(X_i, C_j)$

supprime = False
for chaque $a_i \in D_i$ **do**
  $\tau = Last((X_i, a), C_j)$
  if $\exists k | \tau[X_{jk}][D_{jk}]$ **then**
    $\tau = succ(\tau, D_{X_i=a}^{Scope(C_j)})$
    while $\tau \neq NIL$ and $\tau rel(C_j)$ **do**
      $\tau = succ(\tau, D_{X_i=a}^{Scope(C_j)})$
    end while
    if $\tau \neq NIL$ **then**
      $Last((X_i, a), C_j) = \tau$
    else
      supprimer a de D_i
      supprime = True
    end if
  end if
end for
return *supprime*

Exemple 7 *Considérons le CSP suivant :* $P =< X, D, C >$ *une instance CSP définit comme suit :*
$X = \{A, B, C\}$,
$D = \{D(A), D(B), D(C)\}$ *tel que :* $D(A) = D(B) = D(C) = \{1, 2, 3, 4\}$,
$C = \{\{A < B\}, \{B < C\}\}$
Déroulment de l'algorithme $GAC(X, D, C)$ *:*
$Q = \{(A, A < B), (B, A < B), (B, B < C), (C, B < C)\}$ *et* $NotQ = \{\}$

1. *considérons* $(A, A < B)$ *:*
   $D_A = D(A) = \{1, 2, 3, 4\}$, $D_B = D(B) = \{1, 2, 3, 4\}$, $D_c = D(C) = \{1, 2, 3, 4\}$
   $Q = \{(B, A < B), (B, B < C), (C, B < C)\}$ *et* $NotQ = \{(A, A < B)\}$
   $1 \in D_A \Rightarrow 1 < 2$✓ ,$2 < 3, 4$✓, $3 < 4$✓, $4 <$?
   $A = 4$ *ne passe pas le test.*

$D_A = \{1, 2, 3\}$, $D_B = \{1, 2, 3, 4\}$, $D_c = \{1, 2, 3, 4\}$

2. *considérons* $(B, A < B)$:
   $Q = \{(B, B < C), (C, B < C)\}$ *et* $NotQ = \{(A, A < B), (B, A < B)\}$
   $1 \in D_B$: $1 >?$ *n'est pas vérifié d'ou* $B = 1$ *ne passe pas le test.*
   $D_A = \{1, 2, 3\}$, $D_B = \{2, 3, 4\}$, $D_c = \{1, 2, 3, 4\}$

3. *considérons* $(B, B < C)$
   $Q = \{(C, B < C)\}$ *et* $NotQ = \{(A, A < B), (B, A < B), (B, B < C)\}$
   $2 < 3$✓, $3 < 4$✓, $4 <?$ *n'est pas vérifieé d'ou* $B = 4$ *ne passe pas le test.*
   $D_A = \{1, 2, 3\}$, $D_B = \{2, 3\}$, $D_c = \{1, 2, 3, 4\}$
   Dans ce cas on a fait des modifications dans B *ce qui engendre la révision du domaine de* A *puisque* A *et* B *sont relatif.*
   D'ou on ajoute la contrainte $(A, A < B)$ *dans* Q.
   $Q = \{(C, B < C), (A, A < B)\}$

4. *considérons* $(C, B < C)$
   $D_A = \{1, 2, 3\}$, $D_B = \{2, 3\}$, $D_c = \{1, 2, 3, 4\}$
   $Q = \{(A, A < B)\}$, $NotQ = \{(B, A < B), (B, B < C), (C, B < C)\}$
   $1 \in D_C, 1 >?$ *et* $2 \in D_C, 2 >?$ *ne sont pas vérifié d'ou* $D_C = \{3, 4\}$
   $D_A = \{1, 2, 3\}$, $D_B = \{2, 3\}$, $D_c = \{3, 4\}$

5. *considérons* $(A, A < B)$
   $Q = \{\}$ *et* $NotQ = \{(A, A < B), (B, A < B), (B, B < C)\}$
   $1 > 2, 3$✓, $2 < 3$✓, $3 <?$ *n'est pas vérifié d'ou* $A = 3$ *ne passe pas le test.*
   $D_A = \{1, 2\}$, $D_B = \{2, 3\}$, $D_c = \{3, 4\}$

les solutions sont :
(A, B, C) *:* $(1, 2, 3)$, $(1, 2, 4)$, $(1, 3, 4)$, $(2, 3, 4) \Rightarrow$ *quatre solutions.*

2.3.7 Algorithme Maintaining Arc Consistency (MAC)

L'algorithme MAC (Algorithme 7) [53] est un autre algorithme de recherche qui se base sur la consistance, mais ce dernier maintien la consistance d'arc tous le long de la recherche, il vérifie cette propriété sur toutes les variables instanciées et non instanciées pour garantir le maintien de la consistance. l'algorithne MAC utilise pour le filtrage un algorithme d'arc consistance $AC - x$, et la version de MAC suit la version de AC utilisée : $MAC - 3$ si il utilise la l'algorithme $AC - 3$, $MAC - 2001$ si $AC - 2001$, $MAC - 6$ et $Mac - 7$ [54],etc.
Une autre version de MAC est l'algorithme $MAC3_be$ [55] qui économise les vérifications de contraintes sans maintenir de structures de données supplémentaires à chaque nœud de l'arbre de recherche.

L'algorithme MAC et l'ensemble des algorithmes de consistance (de noeud, arc, chemin, etc.) ne cherche pas la solution du CSP, mais ils sont souvent utilisés en combinaison avec des algorithmes de recherche pour améliorer l'efficacité de la

résolution des CSPs. . Ils réduisent l'espace de recherche en éliminant les valeurs inconsistantes, soit avant le démarrage de la recherche, ou le maintien de la consistance pendant la recherche. Dans $MAC - CBJ$ [56], l'algorithme CBJ est utilisé pour la recherche et MAC est utilisé pour le filtrage

Algorithm 7 Algorithme MAC

Input :le quadruplet (A, NA, D, C), avec A est une affectation, initialement $A = \{\}$, NA est l'ensemble des variables non encore instanciées, initialement $NA = X$, D est l'ensemble des domaines des variables et C est l'ensemble des contraintes.
Output :*true* si le CSP est arc consistent, $false$ sinon.

```
1: if NA = {} then
2:   return A                    /* A est une solution*/
3: else
4:   choisir X_i de NA
5:   repeat
6:     choisir une valeur v de D_i
7:     D_i = D_i − {v}
8:     if A ∪ {X_i, v} est consistante then
9:       D' = Revise(NA − {X_i}, D, C, ⟨X_i, v⟩)      /*Revise est similaire à celle
         de FC*/
10:      AC − x(NA − X_i, D', C)   /*D' est filtré à l'aide d'un algorithme AC*/
11:      if aucun domaine de D' n'est vide then
12:        result = MAC(A{⟨X_i, v⟩}, NA − {X_i}, D', C)
13:        if result ≠ NIL then
14:          return result
15:        end if
16:      end if
17:    end if
18:  until D_i = {}
19:  return NIL                  /* A n'est pas une solution*/)
20: end if
```

2.4 Techniques Basées sur les Décompositions Structurelles

Face à la complexité exponentielle inhérente à la résolution des Problèmes de Satisfaction de Contraintes, les méthodes de recherche systématique et de filtrage, bien qu'améliorées par de nombreuses heuristiques, peuvent s'avérer insuffisantes pour des instances de grande taille ou à la structure de contraintes particulièrement intriquée. Les stratégies de décomposition structurelle émergent alors comme une alternative prometteuse, voire indispensable. L'idée maîtresse de ces approches est d'analyser et d'exploiter la structure topologique du problème, la manière dont les variables sont interconnectées par les contraintes afin de le décomposer en sous-problèmes plus petits, plus localisés, et donc potentiellement plus faciles à résoudre. Une fois ces sous-problèmes résolus, leurs solutions partielles sont combinées pour former une solution globale au problème original [32], [57].

Ces techniques s'appuient sur des représentations graphiques du CSP, telles que le graphe de contraintes ou l'hypergraphe de contraintes, pour identifier des propriétés structurelles comme l'acyclicité ou une faible largeur qui garantissent une résolution efficace.
La complexité de ces méthodes est souvent paramétrée par une mesure de cette largeur structurelle : si cette largeur est bornée par une petite constante, le problème, bien que NP-difficile en général, peut devenir traitable en temps polynomial par rapport à la taille de l'instance, tout en restant exponentiel en la largeur.
Cette section explore les fondements théoriques et les principales méthodes de décomposition structurelle.

2.4.1 Structures Graphiques et Concepts Associés

Avant d'aborder les techniques de décomposition elles-mêmes, il est essentiel de rappeler et de préciser les structures graphiques qui sous-tendent l'analyse structurelle des CSP.

Définition 20 ***(Graphe)***
Un graphe simple G est défini comme une paire (V, E), où V est un ensemble fini de sommets (ou nœuds) et E est un ensemble de paires de sommets distincts, appelées arêtes.
Dans le contexte des CSP binaires, les variables sont associées aux sommets et une arête existe entre deux variables si une contrainte binaire les relie.

Définition 21 ***(Connexité)***
Un graphe est dit connexe si, pour toute paire de sommets distincts (V_i, V_j), il existe un chemin (une séquence d'arêtes) les reliant.
Une composante connexe d'un graphe est un sous-graphe connexe maximal, c'est-à-dire qu'il est connexe et n'est contenu dans aucun autre sous-graphe connexe plus grand. Si un CSP n'est pas connexe, il peut être décomposé en plusieurs CSP indépendants correspondant à ses composantes connexes, chacun pouvant être résolu séparément.

Définition 22 ***(k-connexité)***
Un graphe est k-connexe si la suppression de moins de k sommets ne le déconnecte pas.

Définition 23 ***(Hypergraphe)***
Un hypergraphe est une généralisation d'un graphe, où les arêtes peuvent relier plus de deux sommets. Formellement, un hypergraphe H est défini par une paire (V, E_H), où V est un ensemble de sommets et E_H est un ensemble d'hyperarêtes. Chaque hyperarête est un sous-ensemble non vide de V, pouvant contenir plus de deux sommets. Pour un CSP n-aire, les variables correspondent aux sommets V, et chaque contrainte C_j portant sur un ensemble de variables S_j est représentée par une hyperarête $e_j = S_j$.

Un hypergraphe peut être associé à deux types de graphes : le graphe primal et le graphe dual. Ces graphes sont des représentations graphiques qui facilitent l'analyse des relations entre les variables et les contraintes dans un CSP n-aire.

Définition 24 ***(Graphe Primal)***
Le ***graphe primal*** *d'un hypergraphe H est un graphe simple dont les sommets sont ceux de H, et où une arête relie deux sommets s'ils apparaissent ensemble dans au moins une hyperarête de H.*

Définition 25 ***(Graphe Dual)***
Le ***graphe dual*** *d'un hypergraphe H est un hypergraphe H^* dont les sommets sont les hyperarêtes de H, et où une arête relie deux sommets si les hyperarêtes correspondantes dans H ont une intersection non vide.*

Définition 26 ***(Chaîne)***
Une chaîne dans un hypergraphe est une séquence alternée de sommets et d'hyperarêtes $(x_1, e_1, x_2, e_2, \ldots, x_q, e_q)$ telle que $x_k, x_{k+1} \in e_k$. Un hyperarbre est un hypergraphe qui ne contient aucun cycle (au sens des chaînes).

Définition 27 ***(Séparateur)***
Un séparateur d'un graphe $G = (V, E)$ est un sous-ensemble de sommets dont la suppression déconnecte le graphe. Plus formellement, un ensemble de sommets $S \subseteq V$ est un séparateur de G si la suppression de S de G augmente le nombre de composantes connexes du graphe résultant. Les séparateurs sont souvent utilisés pour diviser un problème complexe en sous-problèmes plus simples à résoudre.

Ces structures graphiques et les concepts associés sont les outils mathématiques sur lesquels reposent l'analyse de la tractabilité structurelle des CSP.

2.4.2 Tractabilité et Acyclicité

La notion d'acyclicité est intimement liée à la tractabilité des CSP. Si la structure d'un CSP est acyclique ou proche de l'acyclicité, des algorithmes efficaces existent. La simple acyclicité du graphe primal n'est pas suffisante pour garantir la tractabilité des CSP n-aires. Des notions plus fortes d'acyclicité pour les hypergraphes ont été définies. L'une des plus importantes est l'α-acyclicité.

Définition 28 ***(Acyclicité d'un Hypergraphe)***
Un hypergraphe est $\alpha - acyclique$ s'il peut être réduit à l'ensemble vide par l'application répétée de deux opérations.
L'α-acyclicité est une propriété qui garantit que les hypergraphes peuvent être traités efficacement.

L'algorithme de Graham [58], souvent appelé GYO reduction (Algorithme 8), permet de réduire un hypergraphe en appliquant des opérations de suppression de sommets et d'hyperarêtes. Ces opérations sont définies comme suit :

- **Suppression d'un sommet** : Si un sommet v n'appartient qu'à une seule hyperarête, on peut le supprimer.
- **Suppression d'une hyperarête** : Si une hyperarête est contenue dans une autre hyperarête, elle peut être supprimée.
  Il a été prouvé que les requêtes conjonctives (équivalentes aux CSP) dont l'hypergraphe est $\alpha - acyclique$ peuvent être évaluées en temps polynomial [59].

Dans le cas ou l'hypergraphe est acyclique, l'algorithme retoure un ensemble vide, sinon il retourne un ensemble de sommets et d'hyperarêtes qui ne peuvent pas être réduits davantage.

Algorithm 8 Algorithme GYO

Input : un hypergraphe $H = (V, E_H)$ où E est l'ensemble des hyperarêtes
Output : H est acyclique ou non

while qu'il existe des hyperarêtes dans H **do**
  Supprimer tout sommet appartenant à une seule hyperarête
  Supprimer toute hyperarête vide ou incluse dans une autre hyperarête
  Ajouter les hyperarêtes supprimées à un ordre quelconque
end while
if toutes les hyperarêtes ont été supprimées **then**
  return H est acyclique
else
  return H est cyclique
end if

Un hypergraphe est $\alpha - acyclique$ si et seulement s'il admet un arbre de jonction [60].

Définition 29 ***(Arbre de Jonction)***

Un arbre de jonction T pour un hypergraphe $H = (V, E_H)$ est un arbre dont les nœuds sont les hyperarêtes de H ($N(T) = E_H$), et qui satisfait la propriété de connexité : pour tout sommet $x \in V$, le sous-ensemble des nœuds de T qui contiennent x forme un sous-arbre connexe de T.

Cette propriété garantit que les informations concernant une variable peuvent être propagées de manière cohérente à travers l'arbre.

Définition 30 ***(Hypergraphe Conforme et Graphe Triangulé)***
Un hypergraphe est dit conforme si toute clique de son graphe primal est contenue dans au moins une hyperarête.
Un graphe est triangulé s'il ne contient aucun cycle induit de longueur supérieure ou égale à quatre . C'est un graphe dans lequel chaque cycle de longueur supérieure ou égale à quatre possède au moins une diagonale, c'est-à-dire une arête qui relie deux sommets non adjacents du cycle.

Un hypergraphe est α-acyclique si et seulement si son graphe primal est triangulé et qu'il est conforme [61]. La triangulation d'un graphe est une étape clé dans de nombreuses méthodes de décomposition.

Ces notions d'acyclicité et les structures arborescentes associées sont fondamentales car elles identifient des classes de CSP qui peuvent être résolues efficacement. Pour les CSP qui ne sont pas nativement acycliques, les méthodes de décomposition visent à les transformer en une structure équivalente qui est, elle, acyclique ou dont la cyclicité est bornée.

2.4.3 Méthodes Structurelles de resolution

Cette section introduit les principales méthodes de décomposition structurelle proposées dans la littérature, en commençant par celles qui se basent principalement sur le graphe de contraintes, plus adaptées aux CSP binaires, pour ensuite aborder celles conçues pour les hypergraphes nécessaires pour les CSP n-aires.
L'objectif commun à toutes ces méthodes est de transformer une instance CSP, potentiellement complexe et cyclique, en une autre instance, équivalente en termes de solutions, mais dont la structure souvent un arbre de clusters de variables ou de contraintes permet une résolution plus efficace.
Chaque méthode définit sa propre notion de "cluster" et sa propre mesure de "largeur". Cette largeur quantifie la complexité résiduelle de la structure décomposée. Si la largeur est petite, la résolution du problème décomposé est généralement rapide, polynomiale en la taille du problème, exponentielle en la largeur.

2.4.3.1 Méthode TCLUSTER (Tree Clustering)

La méthode du Tree Clustering (TCLUSTER), proposée par Dechter et Pearl [29], est une technique influente qui vise à transformer un CSP général en un CSP acyclique équivalent, dont le graphe de contraintes est un arbre de jonction. Le nouveau CSP obtenue a les meme solutions que le CSP de d'origine.
L'algorithme de Tree Clustering presente dans l'Algorithme 9 est basé sur le principe suivant :
Si le graphe G n'est pas déjà triangulé, on y ajoute des arêtes de manière à éliminer tous les cycles induits de longueur supérieure ou égale à quatre. Cette étape est cruciale car les graphes triangulés possèdent des propriétés structurelles qui facilitent leur décomposition. Trouver une triangulation optimale est NP-difficile, donc des heuristiques sont souvent utilisées, comme Maximum Cardinality Search [61]. Puis on identifie les cliques maximales du graphe triangulé (une clique est un sous-graphe complet). Les cliques maximales du graphe triangulé deviennent les nœuds (clusters) de la structure arborescente. Chaque cluster C_i induit un sous-problème du CSP. Puis ces cliques maximales sont organisées en une structure arborescente (arbre de jonction)T telle que pour toute paire de cliques C_i, C_j dans T, tous les nœuds sur le chemin unique entre C_i et C_j dans T doivent contenir l'intersection $C_i \cap C_j$. Cette propriété, dite de l'intersection courante ou de la consistance, est fondamentale. Un tel arbre existe toujours pour l'ensemble des cliques maximales d'un graphe triangulé.
Pour chaque clique C_i, on calcule toutes les solutions au sous-CSP défini par les variables de C_i et les contraintes dont la portée est incluse dans C_i. Ces solutions sont stockées comme des relations. Les solutions partielles sont propagées le long de l'arbre de jonction pour assurer une consistance globale. Cela se fait typiquement par une phase ascendante des feuilles vers une racine, et une phase descendante. Lors de la phase ascendante, un nœud C_i envoie un "message" à son parent C_j, ce message étant la projection de la relation de C_i sur l'intersection $C_i \cap C_j$. Le parent C_j met à jour sa propre relation en la joignant avec ce message. Une fois l'arbre rendu consistant, les solutions globales peuvent être extraites sans retour-arrière en parcourant l'arbre.

Algorithm 9 Algorithme TCLUSTER

Input : Un CSP P et son graphe primal G
Output : Une solution de P si elle existe

1: Trianguler le graphe primal G
2: Identifier les cliques maximales $\{C_1, \ldots, C_t\}$ dans le graphe triangulé, indexées par le rang des nœuds les plus élevés
3: Construire le graphe dual à partir des clusters, et identifier un *join tree* en connectant chaque C_i à un cluster C_j (avec $j < i$) partageant un maximum de variables avec C_i
4: Résoudre localement les sous-problèmes définis par chaque cluster C_1 à C_t
5: Réduire le CSP à un problème où chaque cluster est traité comme une variable singleton, et :

1. Appliquer la consistance d'arc directionnelle (DAC) sur le *join tree*
2. Résoudre le *join tree* sans retour arrière

Exemple 8 *Considérons un CSP défini par les variables* $X = \{A, B, C, D, E\}$ *et les contraintes suivantes :*
$C_1 = (A, B)$, $C_2 = (B, C)$, $C_3 = (C, D)$, $C_4 = (D, E)$, $C_5 = (A, E)$

Étape 1 : Graphe primal : *Le graphe primal* G *relie deux variables si elles apparaissent dans une même contrainte :* $G = (X, E)$, $E = \{(A, B), (B, C), (C, D), (D, E), (E, A)\}$
Ce graphe est un cycle (Figure 2.1).

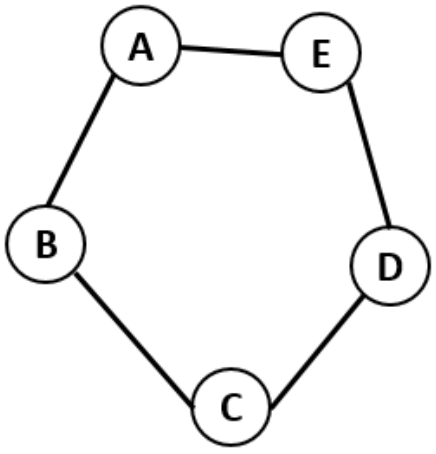

FIGURE 2.1 – Graphe primal du CSP de l'Exemple 8

Étape 2 : Triangulation du graphe *Pour le rendre triangulé (sans cycles de longueur > 3 sans diagonale), on ajoute l'arête* (B, D) *:*

$$E' = E \cup \{(B, D)\}$$

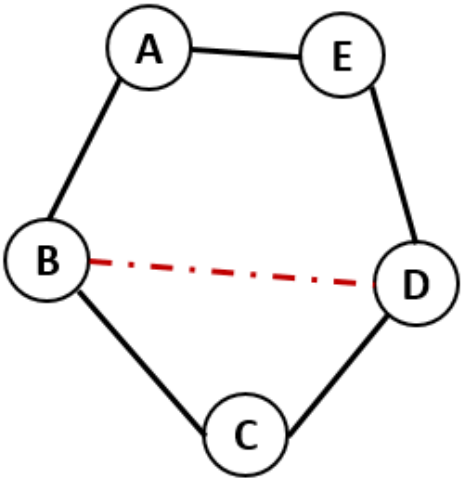

FIGURE 2.2 – Graphe triangulé du CSP de l'Exemple 8

Étape 3 : Identification des cliques maximales *À partir du graphe triangulé, on identifie les cliques maximales suivantes :*

$$\begin{aligned} &\textit{Clique 1} = \{A, B, E\} \\ &\textit{Clique 2} = \{B, C, D\} \\ &\textit{Clique 3} = \{D, E\} \end{aligned}$$

Étape 4 : Construction de l'arbre de jonction *Les cliques sont reliées en fonction des variables partagées (Figure 2.3) : :*

Clique 1 $\leftrightarrow$ *Clique 2* (*variable partagée B*) *Clique 2* $\leftrightarrow$ *Clique 3* (*variable partagée D*)

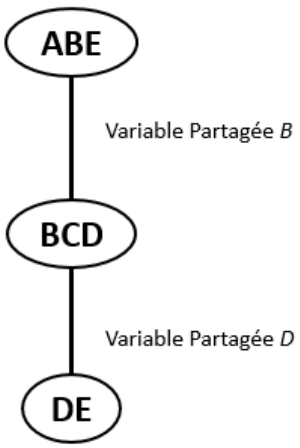

FIGURE 2.3 – Arbre de jonction du CSP de l'Exemple 8

Étape 5 : Résolution

- *On applique la* ***consistance d'arc directionnelle (DAC)*** *sur l'arbre.*

- *Ensuite, on résout l'arbre de jonction **sans retour arrière**, en propageant les solutions compatibles de proche en proche.*

Ce processus garantit une résolution efficace du CSP si sa structure permet une telle décomposition.

La complexité du Tree Clustering est dominée par la taille de la plus grande clique dans le graphe triangulé. Si $w*$ est le nombre de variables dans la plus grande clique (souvent appelé la largeur d'arbre induite), la complexité temporelle est en $O(n \cdot d \cdot w*)$, où n est le nombre de variables et d est la taille maximale des domaines. La complexité spatiale est en $O(s \cdot d^s)$, où s est la taille du plus grand séparateur.
Beaucoup de travaux ont été réalisés pour améliorer l'efficacité de cette méthode, [62], ont proposé une décomposition arborescente basée sur la notion de *bag-connected tree decomposition* qui améliore la consistance des clusters en assurant que chaque cluster est connexe, ce qui est crucial pour la propagation des solutions. [63] ont proposé d'exploiter la suvgarde des nogoods et les techniques de redémarrages pour améliorer l'algorithme BTD.

2.4.3.2 Méthode de la Décomposition en Arbre (TD)

La Décomposition Arborescente (Tree Decomposition - TD), introduite par Robertson et Seymour [64], est un concept fondamental en théorie des graphes qui a trouvé des applications majeures dans la résolution de problèmes algorithmiques, y compris les CSP, principalement ceux à contraintes binaires. Elle formalise l'idée qu'un graphe peut être décomposé en une structure arborescente de "morceaux" plus simples.

Définition 31 ***(Décomposition Arborescente TD)***
Une décomposition arborescente d'un graphe $G = (X, E)$ où X représente l'ensemble des variables du CSP et E l'ensemble des contraintes binaires les reliant, est une paire $(\{B_p\}_{p \in P}, T)$, où :

1. *$\{B_p\}_{p \in P}$ est une collection de sous-ensembles de X, appelés sacs ou clusters, indexés par les nœuds d'un arbre T.*

2. *$T = (P, F)$ est un arbre, où P est l'ensemble des nœuds ou chaque noeud correspondant à un sac et F l'ensemble de ses arêtes.*
   Ces éléments doivent satisfaire les trois conditions suivantes :

   - ***Couverture des variables :** Chaque variable $x \in X$ doit appartenir à au moins un sac B_p. Formellement, $(\bigcup_{p \in P} B_p = X)$.*
   - ***Couverture des contraintes :** Pour chaque contrainte (arête) $\{x, y\} \in E$ du CSP, il doit exister au moins un sac B_p qui contient à la fois x et y.*
   - ***Condition de connexité (ou d'interpolation) :** Pour chaque variable $x \in X$, l'ensemble des nœuds dont les sacs contiennent x : $\{p \in P | x \in B_p\}$ doit induire un sous-arbre connexe de T.*

Cette propriété est cruciale car elle garantit que les informations concernant une variable peuvent être propagées de manière cohérente à travers la structure arborescente.

Définition 32 ***(Largeur d'Arbre (Treewidth))***
La largeur d'une décomposition arborescente donnée est définie comme $\max_{p \in P}(|B_p| - 1)$.
La largeur d'arbre (treewidth) d'un graphe G, notée tw(G), est la largeur minimale sur toutes ses décompositions arborescentes possibles. Intuitivement, la largeur d'arbre mesure à quel point un graphe s'écarte structurellement d'un arbre.

Trouver la largeur d'arbre d'un graphe général et une décomposition optimale est un problème NP-difficile [65], bien que des algorithmes efficaces existent pour des largeurs d'arbre bornées fixées ou des heuristiques pour l'approximer.

La principale utilité de la décomposition arborescente pour les CSP est que si un CSP binaire a un graphe de contraintes de largeur d'arbre bornée par k, il peut être résolu en un temps qui est exponentiel en k mais polynomial en la taille du problème [66].
Un algorithme notable exploitant cette propriété est le Backtracking on Tree-Decomposition (BTD), tel que décrit par Jégou et Terrioux [67].

2.4.3.3 Méthode Backtracking on Tree-Decomposition (BTD)

Le Backtracking on Tree-Decomposition (BTD) [67] est une approche de résolution de CSP qui combine l'efficacité pratique du backtracking avec les garanties théoriques offertes par la décomposition arborescente. Il est particulièrement efficace pour les CSP dont le graphe de contraintes a une largeur d'arbre bornée, ce qui permet de résoudre le problème en temps polynomial par rapport à la taille du problème, mais exponentiel en la largeur d'arbre.

L'algorithme fonctionne en guidant une recherche de type backtracking par un ordre partiel induit par la structure de la décomposition arborescente. Typiquement, les variables au sein d'un cluster sont traitées ensemble ou dans un ordre spécifique, et la recherche progresse d'un cluster à l'autre en suivant les arêtes de l'arbre de décomposition.
Une caractéristique clé du BTD est l'exploitation des séparateurs (intersections entre sacs adjacents) pour introduire les notions de structural goods et structural nogoods.

Définition 33 ***(Structural Nogoods et Structural Goods)***
Un structural nogood est une assignation des variables d'un séparateur qui ne peut être étendue de manière cohérente dans le sous-arbre de décomposition "en dessous" de ce séparateur. Inversement, un structural good est une assignation d'un séparateur qui garantit une extension cohérente. En enregistrant et en réutilisant ces informations, BTD peut élaguer significativement l'arbre de recherche.[67].

L'idée principale de l'algorithme BTD (Algorithme 10) est de guider la recherche de type backtracking en suivant un ordre partiel imposé par la décomposition arborescente. Plutôt que d'instancier les variables une par une dans un ordre linéaire, BTD traite les variables au sein des clusters de la décomposition et progresse d'un cluster à l'autre via les séparateurs.

Algorithm 10 Algorithm BTD

Input : Affectation courante A (initialement vide), cluster courant E_i (initialement E_1), ensemble V_{E_i} des variables de E_i non encore instanciées
Output : True si A a été étendue à toutes les variables de la descendance de E_i, **False** sinon

1: **if** $V_{E_i} = \emptyset$ **then**
2: $Consistency \leftarrow$ **True**
3: $F \leftarrow$ Sons(E_i) /* les sous-clusters de E_i*/
4: **while** $F \neq \emptyset$ **and** $Consistency$ **do**
5: Choisir $E_j \in F$
6: $F \leftarrow F \setminus \{E_j\}$
7: **if** $A[E_i \cap E_j]$ est un *good* **then**
8: $Consistency \leftarrow$ **True**
9: **else**
10: **if** $A[E_i \cap E_j]$ est un *nogood* **then**
11: $Consistency \leftarrow$ **False**
12: **else**
13: $Consistency \leftarrow$ **BTD**(A,E_j, $E_j \setminus E_i \cap E_j$)
14: **if** $Consistency$ **then**
15: Enregistrer le *good* $A[E_i \cap E_j]$
16: **else**
17: Enregistrer le *nogood* $A[E_i \cap E_j]$
18: **end if**
19: **end if**
20: **end if**
21: **end while**
22: **return** $Consistency$
23: **else**
24: Choisir $x \in V_{E_i}$
25: $d_x \leftarrow D_x$
26: $Consistency \leftarrow$ **False**
27: **while** $d_x \neq \emptyset$ **and not** $Consistency$ **do**
28: Choisir $v \in d_x$
29: $d_x \leftarrow d_x \setminus \{v\}$
30: **if** toutes les contraintes $C_i \in C$ impliquant x sont satisfaites par $A \cup \{x \leftarrow v\}$ **then**
31: $Consistency \leftarrow$ **BTD**($A \cup \{x \leftarrow v\}$, E_i, $V_{E_i \setminus \{x\}}$)
32: **end if**
33: **end while**
34: **return** $Consistency$
35: **end if**

À chaque étape, il explore récursivement les clusters de l'arbre de décomposition. Lorsqu'il se trouve sur un cluster E_i, BTD commence par instancier ses variables non affectées. Pour chaque variable, une valeur est choisie et testée. Si l'affectation partielle obtenue est inconsistante, une autre valeur est essayée. Si aucune valeur ne mène à une affectation consistante, un retour arrière est déclenché. Une fois que toutes les variables du cluster courant E_i ont été instanciées de manière consistante,

l'algorithme tente d'étendre l'affectation actuelle aux clusters fils E_j de E_i. Pour chaque sous-cluster, BTD examine la projection de l'affectation actuelle sur le séparateur $E_i \cap E_j$. Si cette projection a déjà été rencontrée dans une solution valide appelée *good*, l'algorithme passe directement au cluster suivant sans exploration. Si au contraire elle est connue pour ne mener à un *nogood*, la recherche revient en arrière sans tenter d'explorer ce sous-arbre. Si la projection est inconnue, BTD effectue un appel récursif sur E_j. Si celui-ci réussit, la projection est mémorisée comme un *good* ; sinon, elle est enregistrée comme un *nogood*.
Ce mécanisme permet d'éviter de recalculer plusieurs fois les mêmes sous-problèmes. L'algorithme retourne finalement `vrai` si l'affectation courante a pu être étendue de manière consistante à tous les clusters descendants, sinon, il retourne `faux`.

La complexité temporelle de BTD est exponentielle en la largeur arborescente w de la décomposition, la complexité est bornée par $O(n * d^{w+1}$, où n est le nombre de variables, d la taille maximale du domaine, plutôt qu'exponentielle en n. Si w est petit, BTD peut être beaucoup plus efficace que le backtracking standard.

2.4.3.4 Décomposition Hypertree Généralisée (GHD)

Les techniques de décomposition arborescente et par arbre de jonction sont principalement conçues pour les graphes et donc plus directement applicables aux CSP avec contraintes binaires. Pour les CSP impliquant des contraintes n-aires, l'hypergraphe de contraintes est une représentation plus naturelle. La Décomposition Hypertree Généralisée (Generalized Hypertree Decomposition - GHD), introduite par Gottlob, Leone, et Scarcello [68], est une méthode de décomposition spécifiquement conçue pour les hypergraphes et donc bien adaptée aux CSP n-aires. Elle est au cœur des contributions de cette thèse, notamment à travers les algorithmes FC-GHD [15] et ses extensions.
la GHD est une généralisation de la décomposition arborescente basée sur une structure appelée hyperarbre, mais elle est plus flexible et puissante, permettant de traiter des hypergraphes avec des structures de contraintes plus complexes.

Définition 34 ***(Hyperarbre)***
Un hyperarbre (hypertrees) pour un hypergraphe $H = (X, E_H)$ où X sont les variables et E_H les hyperarêtes/contraintes est un triplet $(\langle T, \chi, \lambda \rangle)$ défini comme suit :

- *$T = (P, F)$ est un arbre enraciné, dont les nœuds P sont appelés les nœuds de décomposition ou clusters.*
- *$\chi : P \rightarrow 2^X$ est une fonction qui associe à chaque nœud de décomposition $p \in P$ un sous-ensemble de variables ($\chi(p) \subseteq X$), appelé le cluster de p.*
- *$\lambda : P \rightarrow 2^{E_H}$ est une fonction qui associe à chaque nœud de décomposition $p \in P$ un sous-ensemble d'hyperarêtes ($\lambda(p) \subseteq E_H$), appelé la couverture d'arêtes de p.*

Définition 35 ***(Décomposition Hypertree Généralisée (GHD))***
Une décomposition hypertree généralisée (Generalized Hypertree Decomposition (GHD)) pour un hypergraphe $H = (P, E_H)$ est un hyperarbre $(\langle T, \chi, \lambda \rangle)$ qui satisfait les conditions suivantes :

1. ***Couverture des variables des contraintes couvertes :*** *Pour chaque hyperarête $h \in E_H$, il existe un nœud $p \in P$ tel que $h \in \lambda(p)$ et $\text{var}(h) \subseteq \chi(p)$.*

2. ***Condition de connexité des variables :*** *Pour chaque variable $x \in X$, l'ensemble des nœuds $\{p \in P \mid x \in \chi(p)\}$ doit induire un sous-arbre connexe de T.*

3. ***Couverture des contraintes :*** *Pour chaque contrainte (hyperarête) $e \in E_H$, il existe au moins un nœud $p \in P$ tel que $e \in \lambda(p)$ et $\text{var}(e) \subseteq \chi(p)$.*

Définition 36 ***(Largeur d'Hyperarbre Généralisée (Ghw))***
La largeur d'une GHD est $\max_{p \in P} |\lambda(p)|$, c'est-à-dire le nombre maximal de contraintes dans la couverture d'arêtes d'un nœud. La largeur d'hyperarbre généralisée (Ghw) d'un hypergraphe H est la largeur minimale sur toutes ses GHD possibles.

Cette mesure de largeur est cruciale car elle détermine la complexité de la résolution du CSP associé. Plus la Ghw est petite, plus le problème peut être résolu efficacement. Les CSP dont l'hypergraphe de contraintes a une Ghw bornée peuvent être résolus en temps polynomial [68].

Exemple 9 *Soit un CSP défini sur les variables $X = \{A, B, C, D, E, F, G, H, I\}$ avec les contraintes suivantes :*

- $C_1 : \{A, B, C\}$
- $C_2 : \{B, D, E\}$
- $C_3 : \{C, F\}$
- $C_4 : \{E, F, G\}$
- $C_5 : \{G, H, I\}$

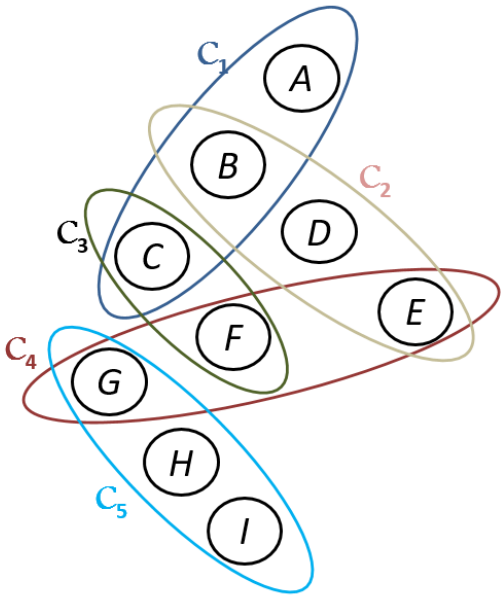

FIGURE 2.4 – Hypergraphe du CSP de l'Exemple 8

Une décomposition hypertree généralisée (GHD) de l'hypergraphe (Figure 2.4) peut être construite comme suit :

- N_1 : $\chi(N_1) = \{A, B, C\}$, $\lambda(N_1) = \{C_1\}$
- N_2 : $\chi(N_2) = \{B, D, E\}$, $\lambda(N_2) = \{C_2\}$
- N_3 : $\chi(N_3) = \{C, F\}$, $\lambda(N_3) = \{C_3\}$
- N_4 : $\chi(N_4) = \{E, F, G\}$, $\lambda(N_4) = \{C_4\}$
- N_5 : $\chi(N_5) = \{G, H, I\}$, $\lambda(N_4) = \{C_5\}$

Les connexions entre les nœuds forment l'arbre de la Figure 2.5 :

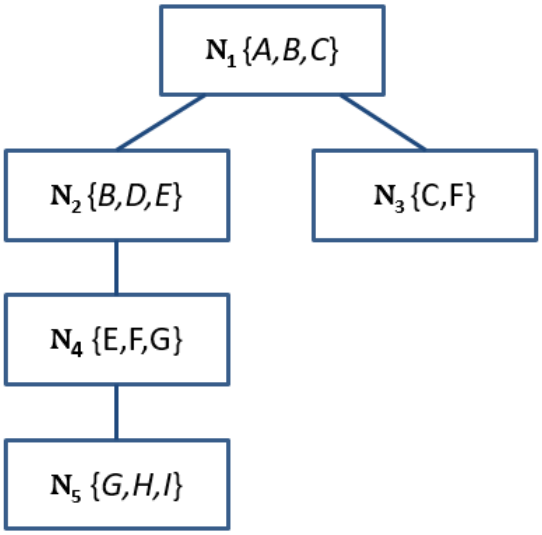

FIGURE 2.5 – Exemple de GHD pour le CSP de l'Exemple 8

Cette décomposition respecte les trois conditions fondamentales d'une GHD : couverture des variables, couverture des contraintes et connexité des occurrences de chaque variable.

Algorithmes de Calcul et de Résolution basés sur GHD

Trouver une GHD de largeur minimale est un problème NP-difficile pour une largeur $k \geq 2$ [68]. Cependant, des algorithmes d'approximation existent, ainsi que des algorithmes exacts pour des classes spécifiques de graphes.
La construction d'une GHD peut être effectuée par des algorithmes comme le "Generalized Hypertree Decomposition Algorithm" (GHDA) [68] . Des algorithmes exacts comme opt-k-decomp [69] et heuristiques comme Bucket Elimination [32], existent.

Une fois qu'une GHD obtenue, un CSP peut être résolu par des algorithmes comme le Join Acyclic Solving (JAS) [57]. JAS calcule pour chaque nœud P de la GHD la jointure des relations correspondant aux contraintes dans $\lambda(p)$, projetée sur les variables de $\chi(p)$. L'arbre de relations résultant est ensuite rendu acycliquement consistant. Cependant, JAS souffre de l'inconvénient de devoir calculer et stocker

potentiellement toutes les solutions de chaque sous-problème, ce qui peut être coûteux en mémoire et en temps.
L'algorithme FC-GHD (Forward-Checking based on a Generalised Hypertree Decomposition) [15] et ses extensions (FC-GHD+NG, FC-GHD+NG+DR), que nous avons explorez dans nos contributions, visent à surmonter ces limitations en combinant une recherche de type forward-checking avec la structure guidée par la GHD. Plutôt que de calculer toutes les solutions des sous-problèmes, FC-GHD cherche une solution à la fois, en propageant les conséquences des assignations partielles au sein des clusters et entre les clusters connectés dans la GHD. L'introduction de structural nogoods dans FC-GHD+NG et de réordonnancement dynamique des sous-arbres dans FC-GHD+NG+DR vise à améliorer encore l'efficacité. La GHD est aussi utiliser l'evaluation des requêtes conjonctives et la resolution des CSPs par la methode HSJ-Solver [70].

Toutes les méthodes de Décomposition structurelle partagent le défi de la construction initiale de la décomposition, qui peut être coûteuse. Cependant, l'investissement dans une bonne décomposition peut être amorti si elle est réutilisée ou si elle permet une résolution significativement plus rapide.
Les travaux présentés dans cette thèse s'inscrivent dans la lignée des algorithmes exploitant la GHD, en cherchant à en optimiser l'application pratique par des techniques de recherche dynamique et d'apprentissage à partir des échecs.

2.5 Techniques Basées sur la Recherche Locale

En contraste avec les méthodes de recherche systématique qui explorent l'espace des solutions de manière exhaustive et garantissent la complétude, les techniques basées sur la recherche locale adoptent une approche foncièrement différente. Au lieu de construire une solution partielle de manière incrémentale, les algorithmes de recherche locale opèrent typiquement sur des assignations complètes des variables, même si celles-ci violent initialement certaines contraintes.

L'objectif est alors d'améliorer itérativement cette assignation complète en effectuant des modifications "locales", en changeant la valeur d'une ou de quelques variables à la fois afin de réduire le nombre de contraintes violées, dans l'espoir d'atteindre une solution globale [71].

Ces méthodes sont généralement incomplètes : elles ne garantissent pas de trouver une solution même si elle existe, et ne peuvent pas prouver l'insatisfiabilité d'un problème. Cependant, leur force réside dans leur capacité à explorer de très grands espaces de recherche et à trouver rapidement des solutions de bonne qualité ou des solutions exactes pour de nombreuses instances de CSP, en particulier celles qui sont de grande taille ou difficilement abordables par des méthodes complètes en un temps raisonnable. Elles sont souvent de nature stochastique, introduisant des éléments aléatoires dans le processus de recherche pour diversifier l'exploration et éviter de rester piégé dans des régions sous-optimales de l'espace de recherche.

2.5.1 Principes de la Recherche Locale pour les CSP

Un algorithme de recherche locale typique pour les CSP peut être caractérisé par les composantes suivantes :

- L'espace de Recherche est l'ensemble de toutes les assignations complètes possibles des variables qui représente le produit cartésien des domaines.
- L'algorithme commence par une solution initiale, qui est une assignation complète, souvent générée aléatoirement ou par une heuristique simple.
- Pour une assignation courante, une structure de voisinage définit l'ensemble des assignations proches qui peuvent être atteintes par une modification locale. La modification la plus courante est le changement de la valeur d'une seule variable.
- Une fonction d'évaluation qui mesure la qualité d'une assignation complète. Pour les CSP, il s'agit typiquement du nombre de contraintes violées. L'objectif est de minimiser cette fonction, idéalement jusqu'à zéro ce qui correspond à une solution.
- Une stratégie pour choisir le prochain état ou assignation parmi les voisins de l'état courant. Cela peut être basé sur la meilleure amélioration (Hill-Climbing), un choix aléatoire (Stochastic Hill-Climbing), ou d'autres critères.
- Le critère d'Arrêt : l'algorithme s'arrête lorsqu'une solution est trouvée, un nombre maximal d'itérations est atteint, un temps limite est écoulé, ou lorsque plus aucune amélioration n'est possible pendant un certain temps.

2.5.2 Algorithmes de Recherche Locale Classiques

Les algorithmes de recherche locale pour les CSP peuvent être classés en plusieurs catégories, selon leur approche et leur mécanisme de recherche. Voici un aperçu des algorithmes classiques les plus connus :

A. Hill-Climbing (Montée de Colline) :
Le Hill-Climbing est l'un des algorithmes de recherche locale les plus simples. À chaque itération, il évalue tous les voisins de l'assignation courante et se déplace vers le voisin qui offre la meilleure amélioration de la fonction d'évaluation, c'est-à-dire celui qui réduit le plus le nombre de contraintes violées. Si aucun voisin n'améliore la solution courante, l'algorithme est bloqué dans un optimum local et s'arrête [72].
Les variantes de l'algorithm Hill-Climbing sont :

- Steepest-Ascent Hill-Climbing, il choisit le meilleur voisin.
- Stochastic Hill-Climbing il choisit aléatoirement parmi les voisins améliorants.
- Hill-Climbing avec Redémarrages Aléatoires [73] (Random-Restart Hill-Climbing), pour pallier le problème des optima locaux, l'algorithme est redémarré plusieurs fois à partir de solutions initiales différentes.

B. Recuit Simulé (Simulated Annealing - SA) :
Inspiré du processus de recuit en métallurgie, le Recuit Simulé [74], [75] est une technique probabiliste conçue pour échapper aux optima locaux. À chaque itération, un voisin est généré aléatoirement. Si ce voisin est meilleur que la solution courante, il est accepté. S'il est moins bon, il peut quand même être accepté avec une certaine probabilité, qui dépend de l'ampleur de la dégradation et d'un paramètre appelé "température" T. Cette probabilité d'acceptation d'un mauvais mouvement est généralement $e^{-\Delta E/T}$, où ΔE est l'augmentation du coût. La température est initialement élevée permettant d'explorer largement l'espace, et diminue progressivement au cours de la recherche rendant l'algorithme plus sélectif. Le SA converge théoriquement vers l'optimum global sous certaines conditions de décroissance de la température, mais en pratique, un schéma de refroidissement est utilisé.

C. Heuristique Min-Conflicts :
L'heuristique Min-Conflicts, proposée par MINTON et al. [76], s'est avérée particulièrement efficace pour les CSP. Elle fonctionne comme suit :

- Commencer avec une assignation complète initiale souvent aléatoire.
- Tant qu'il existe des contraintes violées et que le critère d'arrêt n'est pas atteint :
  a. Choisir aléatoirement une variable x_i qui participe à au moins une contrainte violée.
  b. Choisir la valeur $v \in D_i$ pour x_i qui minimise le nombre de contraintes violées (lnombre de conflits) dans la nouvelle assignation complète. En cas d'égalité, un choix aléatoire peut être fait.
  c. Modifier l'assignation de x_i avec la valeur v choisie.

Min-Conflicts est une forme de Hill-Climbing stochastique focalisée sur les variables en conflit. Sa simplicité et son efficacité, notamment pour les problèmes de planification et d'ordonnancement, elle est appliquée aux CSP flous incomplets [77] et utilisée pour l'optimisation de processus industriels [78].

D. Recherche Tabou (Tabu Search - TS) :
La Recherche Tabou, introduite par Glover [79], est une métaheuristique qui guide un algorithme de recherche locale pour explorer l'espace des solutions au-delà des optima locaux. Elle utilise une mémoire à court terme, appelée liste tabou, qui enregistre les mouvements récents ou les attributs des solutions récemment visitées et les interdit temporairement. Cela permet d'éviter de cycler en revenant immédiatement à des solutions déjà explorées.

2.5.3 Approches Avancées et Métaheuristiques

Au-delà de ces algorithmes classiques, des métaheuristiques plus élaborées ont été développées, combinant souvent plusieurs idées de recherche locale.

- **Algorithmes Génétiques et Approches Évolutionnistes :**
  Les Algorithmes Génétiques [80], [81] s'inspirent de la théorie de l'évolution naturelle. Ils maintiennent une population de solutions candidates. À chaque

génération, les solutions sont évaluées par une fonction de fitness . Des opérateurs génétiques sont appliqués pour créer une nouvelle population :

- Choisir les meilleures solutions pour la reproduction.
- Combiner deux solutions parentes pour créer une ou plusieurs solutions enfants.
- Appliquer des modifications aléatoires aux solutions enfants pour introduire de la diversité.

Bien qu'applicables aux CSP, la conception d'opérateurs de croisement et de mutation efficaces qui préservent une certaine structure de solution peut être délicate.

- **Iterated Local Search (ILS) et Variable Neighborhood Search (VNS) :**
  Ces métaheuristiques sont conçues pour surmonter les limitations du simple Hill-Climbing.
  - **Iterated Local Search (ILS) [82] :** alterne entre des phases de descente vers un optimum local en utilisant un algorithme de recherche locale de base et des phases de perturbation de la solution locale trouvée pour s'échapper de son bassin d'attraction et redémarrer la recherche locale à partir d'un nouveau point.
  - **Variable Neighborhood Search (VNS) [83] :** explore systématiquement différents types de voisinages. Si une recherche locale dans un voisinage simple est bloquée, VNS passe à un voisinage plus large pour tenter de trouver une amélioration, avant de revenir éventuellement à des voisinages plus simples.
- **Algorithmes Spécifiques pour SAT :**
  Le problème de satisfiabilité booléenne (SAT) est un cas particulier de CSP. Des algorithmes de recherche locale très performants [84], comme GSAT et surtout WalkSAT [85] et ses nombreuses variantes, ont été développés.
  WalkSAT, sélectionne une clause non satisfaite, puis choisit une variable de cette clause à changer sa valeur booléenne en se basant sur des heuristiques qui tentent de minimiser le nombre de clauses qui deviendraient non satisfaites ou en introduisant une part d'aléa pour la diversification.

2.5.4 Avantages et limites de la recherche locale pour les CSP

Les méthodes de recherche locale constituent une approche puissante et largement exploitée pour la résolution des CSP, en particulier lorsque les méthodes complètes deviennent impraticables face à la taille ou la complexité du problème. Leur principal atout réside dans leur *scalabilité*, elles sont souvent capables de traiter efficacement des instances de très grande taille. Leur *simplicité conceptuelle* en fait également un choix populaire, car de nombreux algorithmes de base sont faciles à comprendre et à implémenter.

De plus, elles offrent généralement une bonne *qualité de solution*, pouvant fournir rapidement des solutions de haute qualité, voire optimales dans certains cas. Leur *flexibilité* leur permet par ailleurs de s'adapter aisément à divers types de contraintes ou à des objectifs d'optimisation, comme dans les problèmes de satisfaction de contraintes avec fonction objectif (CSOP).
Toutefois, ces techniques présentent aussi certaines limites notables. Elles sont par nature *incomplètes* elles ne garantissent ni la découverte d'une solution existante, ni la preuve de l'insatisfiabilité du problème. De plus, leur efficacité est souvent *fortement dépendante des paramètres* choisis, tels que la taille de la liste tabou dans la recherche tabou, le schéma de refroidissement dans le recuit simulé, ou encore la taille de la population dans les algorithmes génétiques. Un autre inconvénient majeur est leur *vulnérabilité aux optima locaux*, qui peut bloquer certains algorithmes simples dans des solutions sous-optimales.
En somme, la recherche locale constitue un outil précieux dans l'arsenal des méthodes de résolution des CSP, particulièrement adapté aux contextes où la complétude n'est pas une exigence absolue, ou lorsque les ressources computationnelles sont limitées. Les solveurs modernes intègrent d'ailleurs fréquemment des composantes de recherche locale, souvent combinées à des approches systématiques ou à des techniques d'apprentissage automatique pour en tirer le meilleur parti.

2.6 Méthodes Basées sur l'Apprentissage Profond

L'avènement et la prolifération des techniques d'apprentissage profond (Deep Learning - DL) ont révolutionné de nombreux domaines de l'intelligence artificielle, offrant des capacités sans précédent pour l'extraction de motifs complexes à partir de grandes quantités de données. Naturellement, la communauté de la satisfaction et de l'optimisation par contraintes a commencé à explorer le potentiel de ces approches pour s'attaquer aux défis inhérents à la résolution des CSP [86], [87]. Plutôt que de se reposer uniquement sur des heuristiques conçues manuellement ou des stratégies de recherche fixes, les solutions basées sur le DL visent à apprendre comment résoudre les CSP, que ce soit en prédisant directement des solutions, en apprenant des heuristiques de recherche efficaces, ou en guidant le processus de sélection d'algorithmes.

Ces méthodes exploitent la capacité des réseaux de neurones profonds à identifier des régularités et des structures latentes dans les données d'instances de CSP pour optimiser la résolution. Elles se divisent principalement en trois paradigmes : l'apprentissage supervisé, l'apprentissage non supervisé, et l'apprentissage par renforcement.

2.6.1 Méthodes d'Apprentissage Supervisé

L'apprentissage supervisé (Supervised Learning) pour les CSP repose sur l'utilisation de données étiquetées, où chaque instance de CSP est accompagnée d'une information cible que le modèle doit apprendre à prédire. Cette information peut être la satisfiabilité de l'instance, une solution partielle ou complète, la meilleure heuristique à utiliser, ou la performance d'un algorithme donné.

A. Prédiction de Satisfiabilité et de Propriétés de Solution :
Une des premières applications a été d'entraîner des modèles à prédire si une instance CSP donnée est satisfiable ou non. Xu et al. [88] ont utilisé des Réseaux de Neurones Convolutifs (CNN) pour prédire la satisfiabilité de CSP booléens binaires, en représentant les instances sous forme de matrices. Ils ont également exploré des techniques d'adaptation de domaine et d'augmentation de données pour pallier la rareté des données étiquetées, un défi récurrent dû au coût de résolution des CSP qui sont NP-difficiles. L'intuition est que les motifs dans la structure des contraintes, capturés par les filtres convolutifs, peuvent être indicatifs de la satisfiabilité ou l'insatisfiabilité. D'autres travaux visent à prédire des caractéristiques plus fines, comme le nombre de solutions ou la dureté d'une instance.

B. Apprentissage d'Heuristiques de Recherche :
Un axe de recherche particulièrement actif consiste à utiliser l'apprentissage supervisé pour apprendre des heuristiques de recherche plus performantes que celles conçues manuellement.

- **Optimisation de l'Arbre de Recherche (Variable/Value Ordering) :** Song et al. [89] proposent un modèle supervisé pour apprendre des heuristiques d'ordonnancement de variables. L'objectif est d'entraîner un réseau à choisir, à chaque nœud de l'arbre de recherche, la variable qui mènera le plus probablement à une élagage rapide de l'arbre ou à une solution. Les données d'entraînement peuvent provenir d'exécutions de solveurs performants ou de solutions optimales connues pour certaines classes de problèmes. Des approches similaires existent pour l'apprentissage d'heuristiques d'ordonnancement de valeurs.
- **Sélection d'Algorithmes (Algorithm Selection) :** Étant donné qu'aucun algorithme de résolution n'est universellement dominant, la sélection d'algorithmes vise à choisir le meilleur solveur ou la meilleure configuration heuristique pour une instance CSP donnée. Ortiz-Bayliss et al. [90] présentent un cadre général basé sur l'apprentissage machine (supervisé) pour cette tâche, où des caractéristiques de l'instance CSP sont utilisées pour prédire quel algorithme de leur portfolio sera le plus performant. Des travaux antérieurs comme SATzilla [91] ont popularisé cette approche pour le problème SAT.

C. Apprentissage de Structures de Contraintes Complexes :
Des architectures plus récentes, comme les Transformers Récurrents [92] sont explorées pour capturer des structures de contraintes complexes et dynamiques, offrant une alternative aux Réseaux de Neurones Graphiques (GNN) ou aux modèles neuro-symboliques, notamment pour des problèmes CSP visuels où la structure des contraintes émerge de données perceptuelles.

Le principal défi de l'apprentissage supervisé pour les CSP reste l'acquisition de grandes quantités de données étiquetées de haute qualité, ainsi que la conception de représentations d'entrée (features) qui capturent efficacement les aspects pertinents du problème.

2.6.2 Méthodes d'Apprentissage Non Supervisé

L'apprentissage non supervisé (Unsupervised Learning) pour les CSP cherche à découvrir des structures ou des motifs cachés dans les instances de CSP sans nécessiter de données étiquetées, c'est-à-dire sans connaître à l'avance les solutions ou les propriétés cibles. L'objectif est souvent d'apprendre des représentations utiles des variables, des contraintes, ou des instances entières, qui peuvent ensuite être utilisées pour le clustering, la visualisation, ou comme features pour des tâches d'apprentissage ultérieures.

A. Réseaux de Neurones Profonds Agnostiques (Agnostic DNNs) :
Galassi et al. [93] ont exploré l'idée d'utiliser des Réseaux de Neurones Profonds (DNN) de manière indépendante par rapport aux contraintes explicites du problème. Ils entraînent un DNN à étendre une solution partielle en effectuant une seule assignation globalement cohérente, en se basant uniquement sur des exemples de séquences de construction de solutions faisables. L'intérêt scientifique est d'évaluer si un DNN peut apprendre la structure implicite d'un problème combinatoire sans en connaître les règles formelles.

B. Réseaux de Neurones Graphiques (Graph Neural Networks - GNNs) :
Les GNN [94] [95] sont une classe de réseaux de neurones spécifiquement conçus pour traiter des données structurées sous forme de graphes. Dans le contexte des CSP, l'instance peut être représentée comme un graphe ou hypergraphe de contraintes. Les GNN apprennent des représentations vectorielles (embeddings) pour chaque nœud (variable) en agrégeant itérativement les informations de leurs voisins. Ces embeddings peuvent ensuite être utilisés pour diverses tâches.
Xu et al. [96] ont analysé la puissance expressive des GNN, montrant qu'ils peuvent être aussi discriminants que le test d'isomorphisme de graphes de Weisfeiler-Lehman. Dans le cadre des CSP, les GNN peuvent être utilisés de manière non supervisée pour apprendre des représentations de la structure des contraintes, qui pourraient ensuite servir à définir des heuristiques globales ou à guider la décomposition structurelle.

L'apprentissage non supervisé est particulièrement attrayant pour les CSP car il évite le besoin coûteux d'étiquetage, mais le défi est de définir des objectifs d'apprentissage pertinents qui mènent à des représentations utiles pour la résolution.

2.6.3 Apprentissage par Renforcement

L'apprentissage par renforcement (Reinforcement Learning (RL)) offre un paradigme où un agent (le solveur CSP) apprend à prendre des actions, par exemple, choisir une variable à instancier, ou une valeur à assigner dans un environnement ou l'état courant du processus de résolution du CSP, afin de maximiser une récompense cumulative à long terme comme, trouver une solution rapidement, minimiser la taille de l'arbre de recherche [97]. L'agent apprend par essais et erreurs, en explorant différentes stratégies et en observant les conséquences de ses actions.

A. Apprentissage d'Heuristiques de Recherche Adaptatives :
Le RL est particulièrement bien adapté pour apprendre des politiques de re-

cherche dynamiques qui s'adaptent aux caractéristiques spécifiques de l'instance en cours de résolution.

- Xu et al. [91] ont appliqué le RL pour apprendre une fonction de valeur qui adapte les stratégies de résolution, comme le choix d'heuristiques aux caractéristiques spécifiques des instances CSP, en se basant sur l'expérience accumulée pour faciliter la résolution de nouveaux cas similaires.
- **Optimisation du Branchement (Branching Optimization) :** Chalumeau et al. [98] ont intégré un modèle de RL dans le solveur *CP SeaPearl* pour guider les décisions de branchement, choix de la variable et de la valeur. L'agent de RL apprend à partir des caractéristiques historiques des solutions pour orienter la recherche. L'état est souvent représenté par des caractéristiques du graphe de contraintes actuel, et l'action est le choix de la variable/valeur.
- **GNNs Entraînés par Gradient de Politique :** Tönshoff et al. [99] utilisent une approche combinant GNN et RL (gradient de politique) pour apprendre des heuristiques globales pour les CSP sans supervision explicite de la meilleure action à chaque étape. Le modèle reçoit un retour sur la performance globale de la recherche, lui permettant d'ajuster sa politique. Cette approche permet de gérer différents types de contraintes au sein d'un modèle unique.
  D'autres travaux comme ceux de Song et al. [89] peuvent aussi avoir des composantes de RL ou s'en inspirer pour l'apprentissage d'heuristiques dynamiques.

L'apprentissage par renforcement est prometteur car il permet d'apprendre des stratégies complexes et adaptatives. Les défis incluent la définition d'espaces d'états et d'actions appropriés, la conception de fonctions de récompense efficaces, et la gestion de l'exploration par rapport à l'exploitation pendant l'apprentissage.

2.6.4 Défis des Approches Basées sur l'Apprentissage Profond

L'application de l'apprentissage profond à la résolution des CSP est un domaine de recherche relativement jeune mais en pleine expansion. Ces approches offrent le potentiel de découvrir automatiquement des heuristiques et des stratégies de résolution qui surpassent celles conçues par des experts humains. Elles s'adapter à des distributions spécifiques d'instances de problèmes et peuvent traiter des problèmes où les contraintes sont implicites ou difficiles à formaliser symboliquement.
Cependant, plusieurs défis majeurs subsistent :

- **Représentation des CSP :** Comment représenter efficacement une instance CSP (variables, domaines, contraintes de diverses arités et types) en entrée d'un réseau de neurones ? Les matrices de contraintes, les graphes de contraintes, ou d'autres structures sont explorés, mais il n'y a pas encore de consensus.
- **Généralisation :** Les modèles appris sur un ensemble d'instances peuvent ne pas bien généraliser à des instances de tailles différentes, de structures différentes, ou issues d'autres distributions. La scalabilité est un enjeu clé.

- **Coût de l'Apprentissage :** L'entraînement de réseaux de neurones profonds peut nécessiter des ressources computationnelles importantes et de grandes quantités de données qui, pour les CSP, peuvent être coûteuses à générer si elles nécessitent des solutions ou des étiquettes de performance.
- **Interprétabilité :** Les décisions prises par les modèles d'apprentissage profond sont souvent difficiles à interpréter, ce qui peut être un frein dans des applications critiques où la confiance et la vérifiabilité sont importantes.
- **Intégration avec les Solveurs Existants :** Comment intégrer au mieux les composantes apprises au sein des architectures des solveurs CSP symboliques existants pour bénéficier du meilleur des deux mondes ?

Malgré ces défis, l'intersection de l'apprentissage profond et de la résolution de contraintes est une voie de recherche fertile, promettant des avancées significatives dans notre capacité à aborder des problèmes combinatoires complexes.

2.7 Comparaison des Méthodes de Résolution des CSPs

Dans cette section, nous allons examiner et comparer les différentes méthodes de résolution des CSP, en mettant en lumière leurs principes, avantages et limites. Le tableau 2.1 résume les principales catégories de méthodes, leurs principes de fonctionnement, ainsi que leurs avantages et inconvénients respectifs.

Catégorie	Méthode	Principe	Avantages	Limites
Méthodes de recherche systématique	Backtracking (BT)[30]	Explore l'espace des solutions de manière exhaustive avec retour arrière.	Complète et simple à implémenter.	Complexité exponentielle, inefficace sans heuristiques.
	Backjumping (BJ)[33]	Revient directement à la variable à l'origine du conflit.	Réduit l'exploration redondante.	Moins efficace pour les CSP faiblement contraints.
	Forward Checking (FC)[44]	Supprime les valeurs inconsistantes des variables futures.	Évite les impasses précocement.	Surcharge de calcul pour le filtrage.
Techniques d'inférence et de filtrage	Consistance d'arc (AC)[37]	Garantit qu'une valeur a un support compatible.	Réduction efficace des domaines.	Ne garantit pas de solution ; seulement du filtrage.
	Consistance de chemin (PC)[1]	Étend la consistance d'arc aux triplets de variables.	Réduction plus forte.	Coût plus élevé.
	Maintien de la consistance d'arc (MAC)[27]	Maintient AC dynamiquement pendant la recherche.	Réduit les retours arrière.	Complexité accrue.
Décomposition structurelle	Décomposition en arbre (TD)[64]	Arborescence des contraintes.	Réduction de complexité si faible treewidth.	Inefficace si treewidth élevé.

	Join Tree Decomposition (JTD)[67]	Regroupe en cliques hiérarchiques.	Moins de propagation inutile.	Dépend de la taille des cliques.
	Hypertree généralisé (GHD)[100]	Décompose en hypertrees.	Flexible et scalable.	Construction coûteuse.
Recherche locale	Min-Conflicts [101][77]	Change localement les valeurs pour réduire les conflits.	Rapide pour grands problèmes.	Ne garantit ni solution optimale ni complétude.
Apprentissage profond	Apprentissage supervisé [88][92][90][89]	Apprend à partir d'instances annotées.	Améliore les heuristiques.	Besoin de données étiquetées.
	Apprentissage non supervisé [93][96]	Découvre des structures cachées.	Adaptable, pas besoin d'annotation.	Moins de contrôle sur les résultats.
	Apprentissage par renforcement (RL) [98][91][99]	Apprend par essai-erreur.	S'adapte au problème.	Entraînement complexe et coûteux.

Table 2.1 – Comparatif des méthodes de résolution des CSP

L'analyse des différentes méthodes de résolution des CSP montre un compromis clair entre complétude, efficacité et exploitation de la structure. Les méthodes de recherche systématique garantissent une complétude théorique, mais souffrent d'une complexité exponentielle, rendant leur application directe limitée aux instances de taille modérée. Les techniques d'inférence et de filtrage viennent pallier cette limitation en réduisant l'espace de recherche par propagation de contraintes, améliorant ainsi l'efficacité pratique sans altérer la complétude lorsqu'elles sont combinées avec la recherche systématique. Les approches de décomposition structurelle offrent une perspective complémentaire, en exploitant les propriétés topologiques des graphes de contraintes pour traiter efficacement certaines classes de CSP, notamment celles de faible largeur arborescente, et justifient pourquoi certains problèmes structurellement favorables restent tractables malgré leur taille. En revanche, pour les CSP de grande dimension ou faiblement structurés, les techniques de recherche locale permettent une exploration heuristique rapide de l'espace des solutions, offrant des performances empiriques élevées malgré l'absence de garanties de complétude ou d'optimalité. Enfin, l'intégration de méthodes d'apprentissage profond constitue un levier moderne pour améliorer les heuristiques et guider la recherche, en tirant parti de l'expérience sur des familles d'instances, bien que ces approches restent dépendantes des données et ne fournissent pas de garanties formelles.

Dans l'ensemble, cette typologie met en évidence la nécessité d'approches hybrides, combinant recherche systématique, filtrage, exploitation structurelle et apprentissage adaptatif, afin d'équilibrer garanties théoriques et performance pratique, et de rendre les solveurs capables de traiter efficacement la diversité des CSP rencontrés dans la réalité.

2.8 Conclusion

Ce chapitre a présenté un large éventail de méthodes de résolution des CSP, chacune avec ses propres caractéristiques. Les méthodes de recherche systématique, renforcées par la propagation de contraintes, offrent la complétude mais peuvent être coûteuses. La recherche locale offre une scalabilité mais sacrifie la complétude. Les stratégies de décomposition structurelle tentent de tirer parti de la topologie du problème pour le simplifier. Enfin, l'apprentissage profond ouvre des perspectives nouvelles en apprenant des stratégies à partir de données.
Aucune méthode n'est universellement supérieure, le choix dépend fortement de la nature du CSP, sa taille, la densité des contraintes, le type de contraintes, sa structure, ses besoin de complétude, etc. Les solveurs modernes combinent souvent plusieurs de ces techniques, par exemple, recherche systématique avec MAC et apprentissage de nogoods, ou recherche locale avec des redémarrages intelligents.
Les défis persistent, notamment pour les CSP n-aires de grande taille et à structure complexe. Les techniques de décomposition, telles que GHD, sont prometteuses pour ces cas, mais leur efficacité, comme nous le verrons, peut être améliorée par des stratégies dynamiques, ce qui motivera les contributions présentées dans les chapitres suivants.

Chapitre 3

Amélioration de la Résolution de CSP N-aires via GHD

3.1 Introduction

Dans le chapitre précédent nous avons dressé un panorama des diverses méthodes de résolution des CSPs, soulignant notamment l'intérêt des techniques de décomposition structurelle pour aborder les instances complexes, en particulier celles impliquant des contraintes n-aires. Parmi ces techniques, la Décomposition Hypertree Généralisée (GHD) s'est révélée être un cadre puissant, offrant des garanties théoriques de tractabilité pour les CSP dont la largeur d'hyperarbre généralisée (ghw) est bornée.

Cependant, la simple existence d'une GHD de faible largeur ne suffit pas toujours à garantir une résolution pratique efficace. Les algorithmes qui exploitent la GHD, tels que le Join Acyclic Solving (JAS) [57], bien que théoriquement fondés, peuvent souffrir de problèmes d'explosion mémoire ou de coûts de calcul prohibitifs en devant calculer et stocker l'ensemble des solutions de chaque sous-problème défini par les nœuds de la GHD. Pour pallier ces inconvénients, des approches combinant la recherche énumérative avec la structure guidée par la GHD ont été proposées. L'algorithme Forward-Checking based on a Generalised Hypertree Decomposition (FC-GHD) et ses extensions [15] représentent une telle tentative, visant à bénéficier de l'efficacité de la recherche en profondeur tout en exploitant la structure de la GHD pour élaguer l'espace de recherche.

Ce chapitre a un double objectif. Premièrement, il vise à présenter en détail le fonctionnement de l'algorithme FC-GHD et de ses variantes améliorées par l'intégration de nogoods structurels (FC-GHD+NG) et par le réordonnancement dynamique des sous-arbres (FC-GHD+NG+DR). Cette présentation est essentielle pour comprendre le contexte de nos contributions. Deuxièmement, et c'est l'apport principal de ce chapitre, nous introduirons une nouvelle stratégie basée sur le redémarrage (restart), conçue pour améliorer la robustesse et les performances de ces algorithmes basés sur GHD, en particulier face à des choix initiaux sous-optimaux dans l'ordre de traitement des clusters de la GHD ou lors de la rencontre de régions difficiles de l'espace de recherche. Nous décrirons notre algorithmes, et présenterons une évaluation expérimentale de leur efficacité.

3.2 Forward Checking Guidé par une GHD

L'algorithme FC-GHD [15] a été proposé comme une alternative au JAS pour résoudre les CSP n-aires en exploitant une GHD. Contrairement à JAS qui calcule toutes les solutions des sous-problèmes, FC-GHD recherche une solution globale à la fois, à la manière d'un algorithme de backtracking, mais où la structure de la recherche est guidée par la GHD.

3.2.1 Définitions Préliminaires

Définition 37 ***(GHD)***
Supposons disposer d'une Décomposition Hypertree Généralisée ($\langle T, \chi, \lambda \rangle$) *de l'hypergraphe de contraintes du CSP, où* $T = (P, F)$ *est un arbre de décomposition,* $\chi(p)$ *est le sac de variables du nœud* $p \in P$*, et* $\lambda(p)$ *est l'ensemble des contraintes (hyperarêtes) associées à* p*.*

Définition 38 ***(Sous-Problème)***
À chaque nœud $p \in P$ *de la GHD est associé un sous-problème* P_p *défini par les variables de son sac* $\chi(p)$ *et les contraintes de sa couverture d'arêtes* $\lambda(p)$*. Une solution de* P_p*, notée sol*(P_p)*, est une assignation des variables de* $\chi(p)$ *qui satisfait toutes les contraintes de* $\lambda(p)$*.*

Ordre de traitement :
Les nœuds de la GHD sont typiquement traités selon un ordre de parcours en profondeur d'abord (pre-order traversal) à partir d'un nœud racine choisi.

Notations

- $prec(n_i)$: Nœud précédant n_i dans l'ordre de parcours.
- $succ(n_i)$: Nœud suivant n_i dans l'ordre de parcours.
- $Parent(n_i)$: Parent de n_i dans l'arbre de décomposition T.

3.2.1.1 L'Algorithme FC-GHD

L'algorithme FC-GHD explore les solutions en parcourant les nœuds de la GHD. Pour chaque nœud n_i de la GHD traité selon un ordre de parcours en profondeur, FC-GHD tente de trouver une solution sol(P_{n_i}) pour le sous-problème associé. Cette recherche de sol(P_{n_i}) est elle-même réalisée par un algorithme de type Forward Checking appliqué sur le dual du sous-problème où les variables duales sont les contraintes de $\lambda(n_i)$ et leurs domaines sont les tuples de leurs relations.
Le fonctionnement général est le suivant :

1. **Initialisation :** L'algorithme commence par le nœud racine n_1 de la GHD.
2. **Traitement d'un Nœud** n_i **:**
    - Résolution du Sous-Problème P_{n_i} : Chercher une assignation sol(P_{n_i}) pour les variables $\chi(n_i)$ qui satisfait les contraintes $\lambda(n_i)$ et qui est compatible avec la solution sol$(P_{\text{Parent}(n_i)})$ du parent de n_i sur les variables communes $\chi(n_i) \cap \chi(\text{Parent}(n_i))$.

- Si une solution sol(P_{n_i}) est trouvée :
  - L'algorithme appelle une procédure de Filtrage. Cette procédure vérifie la cohérence de sol(P_{n_i}) avec les contraintes situées aux nœuds descendants de n_i dans la GHD. Plus précisément, pour chaque enfant n_c de n_i, elle s'assure que sol(P_{n_i}) peut être étendue en une solution partielle pour les contraintes de $\lambda(n_c)$ qui impliquent des variables de $\chi(n_i)$.
  - Si le filtrage réussit c'est-à-dire que sol(P_{n_i}) est viable pour les descendants, la procédure Filter_child_nodes(n_i) est appelée. Elle propage les conséquences de sol(P_{n_i}) en élaguant les tuples des relations des contraintes aux nœuds enfants n_c. Les tuples de ces relations incompatibles avec sol(P_{n_i}) sont marqués comme supprimés.
  - L'algorithme passe ensuite au nœud suivant dans l'ordre de parcours (succ(n_i)).
- Si aucune solution sol(P_{n_i}) n'est trouvée ou si le filtrage échoue :
  - L'algorithme effectue un Retour-Arrière (BackTrack(n_i)). Cette procédure consiste à : Restaurer les tuples qui avaient été marqués comme supprimés par n_i ou ses descendants Restore_removed_tuples(n_i). et revenir au nœud parent $Parent(n_i)$ pour y chercher une autre solution.

3. **Terminaison :**

   Si tous les nœuds ont été traités avec succès, une solution globale est formée par la combinaison des sol(P_{n_i}). Sinon si un retour-arrière est effectué jusqu'à la racine et que celle-ci n'a plus de solutions alternatives, le CSP est insatisfiable.

3.2.1.2 Améliorations de FC-GHD

L'algorithme FC-GHD de base peut être amélioré en intégrant des mécanismes plus sophistiqués pour gérer les échecs et optimiser la recherche. Deux améliorations majeures ont été proposées par Habbas et al. [15] : l'introduction de nogoods structurels (FC-GHD+NG) et le réordonnancement dynamique des sous-arbres (FC-GHD+NG+DR).

1. **L'algorithme FC-GHD + NG**
   L'algorithme FC-GHD+NG introduit la notion de nogoods structurels pour éviter de réexplorer des branches de l'arbre de recherche qui sont vouées à l'échec. Cette approche repose sur l'enregistrement des nogoods lors des échecs et leur utilisation pour filtrer les solutions potentielles dans les nœuds parents.

   Définition 39 ***(Nogood Structurel)***
   Un nogood structurel est une assignation des variables du séparateur $\chi(Parent(n_i)) \cap \chi(n_i)$ qui a conduit à un échec lors de la tentative de résolution du sous-problème P_{n_i} ou d'un de ses descendants.
   Plus précisément, si une assignation sol($P_{Parent(n_i)}$) du parent mène à une impasse dans le sous-arbre enraciné en n_i, alors la projection de sol($P_{Parent(n_i)}$) sur le séparateur est enregistrée comme un nogood pour le nœud $Parent(n_i)$ par rapport à son enfant n_i.

Fonctionnement de $FC - GHD + NG$

Lorsqu'un retour-arrière se produit du nœud n_i vers son parent Parent(n_i), la procédure Record_nogood(n_i) est appelée pour enregistrer le nogood structurel. Lors de la résolution d'un sous-problème P_{n_i}, avant de tenter une solution sol(P_{n_i}), la procédure Filter_NG vérifie si la projection de cette solution potentielle sur le séparateur avec son parent correspond à un nogood déjà enregistré. Si c'est le cas, cette voie est immédiatement abandonnée. La fonction Nogood(sol(P_{n_i})) effectue cette vérification.

Avantage de FC-GHD + NG

L'enregistrement et l'utilisation des nogoods structurels permettent d'éviter de réexplorer des branches de l'arbre de recherche de la GHD qui sont vouées à l'échec pour des raisons déjà identifiées au niveau des séparateurs, améliorant ainsi l'élagage.

2. **l'algorithme FC-GHD + NG + DR**
   Cette version, FC-GHD+NG+DR, va plus loin en introduisant un mécanisme de Réordonnancement Dynamique (Dynamic Reordering - DR) des sous-arbres de la GHD. Lorsqu'une impasse est détectée profondément dans un sous-arbre de la GHD par exemple, au nœud n_k qui est un descendant lointain de n_j, un simple retour-arrière vers Parent(n_k) peut ne pas être suffisant si la cause réelle du conflit se situe plus haut, au niveau de n_j. L'algorithme pourrait alors explorer inutilement d'autres branches sous n_j avant de finalement revenir à n_j.
   Si une impasse est rencontrée au nœud n_i et que toutes les solutions pour son parent $n_j = Parent(n_i)$ ont été explorées sans succès pour étendre la solution au sous-arbre enraciné en n_i, FC-GHD+NG+DR ne se contente pas de revenir au parent de n_j. Il modifie dynamiquement l'ordre de parcours (σ) de la GHD. La procédure Reorder_hypertree(n_i, σ) est appelée. Elle réinsère tous les nœuds du sous-arbre enraciné en n_i qui le sous-arbre fautif, juste après le parent de n_i qui est n_j dans la liste (σ).
   L'idée est que si le sous-arbre enraciné en n_i est la source de nombreux conflits, il est préférable de le traiter le plus tôt possible après avoir fait un choix pour son parent n_j, plutôt que d'explorer d'autres branches qui pourraient être indépendantes du conflit. Cela vise à détecter les incohérences plus rapidement.

Correction de FC-GHD :

L'algorithme FC-GHD et ses améliorations sont correct puisque ils ne trouvent que des solutions valides, complet car il trouve une solution si elle existe, et termine. La correction découle du fait que les assignations locales sont vérifiées et que la compatibilité entre nœuds parents et enfants est assurée. La complétude est garantie par la nature exhaustive de la recherche de type backtracking et par le fait que la GHD couvre toutes les variables et contraintes. La terminaison est assurée car l'espace de recherche est fini.

Ces algorithmes, FC-GHD et ses variantes NG et NG+DR, constituent la base sur laquelle nous allons maintenant introduire notre principale contribution : l'intégration d'une stratégie de redémarrage pour améliorer davantage leur robustesse et leur performance.

3.3 Stratégies de Redémarrage dans la Résolution de CSP

Avant de présenter en détail nos contributions relatives à l'intégration de stratégies de redémarrage au sein des algorithmes de résolution de CSP basés sur la Décomposition Hypertree Généralisée (GHD), il est pertinent de situer cette approche dans le contexte plus large des recherches existantes. L'utilisation de redémarrages (restarts) est une technique bien établie et reconnue pour son efficacité dans l'amélioration de la performance et de la robustesse des algorithmes de recherche combinatoire, tant pour les CSP que pour les problèmes de satisfiabilité booléenne (SAT). L'intuition fondamentale derrière les redémarrages est de contrer le phénomène de "heavy-tailed distribution" des temps d'exécution, où un algorithme peut occasionnellement s'engager dans des branches de l'espace de recherche particulièrement longues et infructueuses. En interrompant périodiquement la recherche et en la relançant, potentiellement avec des modifications de paramètres ou d'heuristiques, on espère éviter ces explorations coûteuses et diversifier la recherche pour augmenter les chances de trouver une solution rapidement [16] [17]. Ce panorama des travaux existants mettra en lumière les différentes facettes et bénéfices des stratégies de redémarrage, justifiant ainsi notre démarche.

Une première direction dans l'utilisation des redémarrages concerne leur capacité à rendre les stratégies de recherche plus adaptatives. Les travaux de Guddeti [102] sur la stratégie Randomization and Dynamic Geometric Restarts (RDGR) illustrent bien ce principe. S'appuyant sur les approches antérieures comme Randomization and Rapid Restarts (RRR) et Randomization and Geometric Restart (RGR), RDGR se distingue par sa capacité à ajuster dynamiquement le seuil de redémarrage en fonction des progrès réalisés par la recherche. Plutôt que d'appliquer une séquence fixe de seuils, RDGR augmente géométriquement le budget alloué à la prochaine phase de recherche uniquement si la qualité de la solution courante a été améliorée durant la phase actuelle. Cette adaptabilité permet de consacrer davantage de ressources aux explorations qui semblent prometteuses tout en coupant court plus rapidement aux recherches stagnantes, menant ainsi à des résultats plus stables et souvent plus performants, notamment pour les instances sur-contraintes. Cette approche souligne l'intérêt de ne pas considérer le redémarrage comme une simple interruption, mais comme une opportunité de réévaluer et d'ajuster la stratégie de recherche en cours.

Un autre aspect crucial, souvent couplé aux redémarrages, est l'apprentissage à partir des échecs. L'idée est que l'information glanée lors d'une phase de recherche, même si elle se termine par un redémarrage avant la découverte d'une solution, ne doit pas être perdue. Les travaux de Lecoutre, Saïs, Tabary et Vidal [103] se concentrent sur l'enregistrement et la minimisation des nogoods extraits de la dernière branche explorée juste avant qu'un redémarrage ne soit déclenché. Ces nogoods, qui représentent des assignations partielles prouvées inconsistantes, sont ensuite conservés et exploités lors des phases de recherche ultérieures pour élaguer l'espace et éviter de répéter les mêmes erreurs. Cette combinaison de redémarrages et d'apprentissage de nogoods permet de bénéficier à la fois de la diversification de la recherche offerte par les redémarrages et de l'efficacité croissante due à l'ac-

cumulation de connaissances sur les régions infructueuses de l'espace de recherche, améliorant ainsi significativement la robustesse globale du solveur.

L'application des stratégies de redémarrage aux méthodes de résolution basées sur la décomposition structurelle est particulièrement pertinente pour les travaux présentés dans cette thèse. Jégou et Terrioux [63] ont exploré cette voie dans le contexte de l'algorithme BTD (Backtracking with Tree-Decomposition). Ils ont identifié que l'efficacité de BTD est souvent très sensible au choix initial du cluster racine de la décomposition arborescente. Un mauvais choix peut conduire à une dégradation significative des performances. Pour pallier ce problème, ils proposent d'intégrer des redémarrages qui permettent de sélectionner un nouveau cluster racine à chaque reprise de la recherche. Cette approche offre la possibilité d'explorer le problème sous différentes perspectives structurelles, réduisant ainsi la dépendance à un unique choix initial. Leurs travaux examinent également en détail comment les informations apprises, telles que les nogoods classiques et les "structural (no)goods" spécifiques à BTD, peuvent être gérées et préservées de manière valide à travers ces changements de racine, assurant que l'apprentissage reste bénéfique malgré la modification de l'orientation de la recherche induite par le redémarrage.

Enfin, il convient de mentionner une autre utilisation des redémarrages, qui est de servir de cadre pour des méta-algorithmes d'apprentissage. Les travaux de Wattez, Koriche, Lecoutre, Paparrizou et Tabary [104] [105] utilisent des séquences de redémarrages pour structurer le processus d'apprentissage de la meilleure heuristique d'ordonnancement des variables. Dans ce paradigme, chaque "run" entre deux redémarrages constitue un "essai" pour un algorithme d'apprentissage basé sur les bandits manchots. L'algorithme de bandit sélectionne une heuristique, le solveur l'exécute jusqu'au prochain seuil de redémarrage, et une récompense mesurant la performance de l'heuristique est retournée au bandit. Ici, les redémarrages ne modifient pas directement la stratégie de recherche de bas niveau, mais fournissent les points de décision et le feedback nécessaires à un processus d'apprentissage de plus haut niveau visant à optimiser le choix des composantes du solveur.

Ce panorama des travaux existants démontre la polyvalence et l'efficacité reconnue des stratégies de redémarrage dans divers contextes de la résolution de CSP. Qu'il s'agisse d'adapter dynamiquement les paramètres de recherche, d'intégrer des mécanismes d'apprentissage de conflits, de pallier la sensibilité à des choix structurels initiaux dans les méthodes de décomposition, ou de supporter des processus d'apprentissage d'heuristiques, les redémarrages offrent un levier puissant pour améliorer les solveurs. C'est fort de ces constats et de ces inspirations que nous avons entrepris d'explorer l'intégration de stratégies de redémarrage spécifiquement adaptées aux algorithmes de résolution basés sur la GHD, comme FC-GHD et ses variantes, afin d'en accroître la robustesse et l'efficacité.

3.4 Algorithmes *Restart-FC-GHD+NG+DR*

S'inspirant du succès de FC-GHD+NG+DR et des stratégies de redémarrage éprouvées, nous proposons une nouvelle classe d'algorithmes qui intègrent un mécanisme de redémarrage pour modifier dynamiquement l'ordre de traitement des

clusters de la GHD, tout en préservant et en exploitant les nogoods découverts. Nous présenterons ici le principe général de cet approche.

L'idée centrale est d'exécuter un algorithme de base FC-GHD avec gestion des nogoods pendant un certain cout de recherche, typiquement mesuré par le nombre de retours-arrière (backtracks) effectués. Si aucune solution n'est trouvée ou si le problème n'est pas prouvé insatisfiable avant que ce cout ne soit épuisé, la recherche est interrompue et un redémarrage est initié.
Lors d'un redémarrage :

- Les nogoods structurels appris lors de la phase de recherche précédente sont conservés. Ces nogoods représentent des assignations partielles des séparateurs qui ont été prouvées infructueuses et restent valides indépendamment de l'ordre de traitement des clusters. Leur préservation est cruciale pour éviter de refaire les mêmes erreurs lors des phases de recherche ultérieures.

- Un nouvel ordre de traitement des clusters de la GHD est sélectionné. Dans le contexte d'une GHD, cela revient généralement à choisir un nouveau nœud racine pour l'arbre de décomposition T. Ce nouveau choix de racine induit un nouvel ordre de parcours en profondeur et donc une stratégie d'exploration différente du CSP. L'ensemble de tous les ordres de parcours possibles correspondant à chaque nœud de la GHD pouvant servir de racine est noté $ORDERS$ dans les articles de contribution.

- Mise à jour du seuil de redémarrage qui correspond au nombre de backtracks pour le prochain redémarrage peut être ajusté, pour donner plus de temps à la recherche avec le nouvel ordre.

- La recherche est relancée avec le nouvel ordre et le nouveau seuil, en bénéficiant des nogoods précédemment appris.

Ce cycle de recherche, interruption, changement d'ordre et reprise se pour suit jusqu'à ce qu'une solution soit trouvée, que le problème soit prouvé insatisfiable, si tous les ordres possibles ont été explorés sans succès, ou si un nogood vide est dérivé, ou qu'un critère d'arrêt global temps total, nombre total de redémarrages soit atteint.

3.4.1 Description algorithmique de *Restart-FC-GHD+NG+DR*

Nous présentons ici une description de notre algorithme *Restart-FC-GHD+NG+DR* proposé dans notre articles [106]. Cet algorithmes, vise à intégrer une stratégie de redémarrage dans le cadre de la résolution de CSP n-aires en utilisant la GHD. L'algorithme est basé sur le principe de redémarrage dans l'algorithme FC-GHD+NG+DR avec conservation des nogoods.

L'algorithme *Restart-FC-GHD+NG+DR* présenté dans l'Algorithme 11 intègre explicitement le redémarrage au sein de l'algorithme FC-GHD+NG+DR. La principale différence est que le mécanisme de base pour la phase de résolution est déjà FC-GHD+NG+DR, incluant son propre réordonnancement dynamique des sous-arbres. Le redémarrage global vient alors en complément pour changer la racine de

la GHD lorsque la recherche avec la racine courante et ses réordonnancements locaux s'avère trop coûteuse toute en conservant les nogoods deja enregistrés.

Algorithm 11 Algorithme *Restart-FC-GHD+NG+DR*

Input : une GHD compète' $\langle T, \chi, \lambda \rangle$ associer a une instance CSP
Output : une solution $\mathcal{A}$ of P si elle existe

1: $\sigma \leftarrow (n_1, n_2, \ldots, n_e)$ /* σ un pre-ordre en prpfondeur d'abord T avec n_1 comme racine */
2: $n_i \longleftarrow n_1$
3: $nb\_backtracks \leftarrow 0$
4: $restart \leftarrow 1$
5: **while** $restart = 1$ **do**
6: **while** $n_i \neq \emptyset$ **do**
7: $sol(P_{n_i}) \leftarrow Solve\_subpb(P_{n_i})$
8: **if** $sol(P_{n_i}) = \emptyset$ **then**
9: **if** $n_i = n_1$ **then**
10: $\mathcal{A} \leftarrow \emptyset$
11: $restart \leftarrow 0$
12: exit /* P est insatisfiable */
13: **else**
14: $nb\_backtracks++$
15: **if** $nb\_backtracks < limit\_backtracks$ **then**
16: *Backtrack*(n_i)
17: **else**
18: $restart \leftarrow 0$
19: *Break*
20: **end if**
21: **end if**
22: **else**
23: *Filter-NG*(n_i)
24: **end if**
25: **end while**
26: **if** $restart = 1$ **then**
27: $\mathcal{A} \leftarrow \bowtie_{i=1}^{i=e} sol(P_{n_i})$
28: **return** $\mathcal{A}$
29: **else**
30: P est insatisfiable
31: **end if**
32: **end while**

3.4.2 Principe général de l'algorithme *Restart-FC-GHD+NG+DR*

L'algorithme prend en entrée une instance de CSP notée $P = \langle X, D, C \rangle$, où X est l'ensemble des variables, D leur domaine respectif, et C l'ensemble des contraintes. Il reçoit également une décomposition hypertree associée H $= \langle T, \chi, \lambda \rangle$, où T est un arbre couvrant les clusters de contraintes, χ associe à chaque nœud un ensemble de

variables, et λ associe les contraintes correspondantes. Enfin, l'ensemble ORDERS contient tous les ordres possibles de parcours des nœuds de T.

Il commence par choisir un ordre de traitement initial $\sigma_{\text{current}} \in$ ORDERS, initialise un seuil de retour-arrière appelé *limit_backtracks*, ainsi qu'un compteur de backtracks à zéro pour suivre le nombre de retours-arrière effectués. Il initialise aussi un ensemble vide de nogoods .

Ensuite, l'algorithme entre dans une boucle principale qui s'exécute tant qu'aucune solution n'est trouvée et que le problème n'a pas encore été prouvé insatisfiable. À chaque itération, il tente de résoudre le CSP selon l'ordre actuel σ_{current}, en utilisant un algorithme de type *Forward Checking* basé sur la décomposition (FC-GHD). Cela implique de résoudre les sous-problèmes associés à chaque nœud via la fonction *Solve_subpb*(P_{n_i}), tout en effectuant de la propagation de contraintes.

À chaque retour-arrière, le compteur est incrémenté et les nogoods découverts sont enregistrés à l'aide de la procédure *Record_nogood*(Algorithme 12). Si une solution est trouvée à ce stade, elle est immédiatement retournée. Sinon, après chaque retour-arrière, l'algorithme vérifie si le nombre de backtracks a atteint ou dépassé le seuil *limit_backtracks*. Si c'est le cas, il redémarre le processus avec un nouvel ordre de parcours, choisi parmi les ordres restants dans ORDERS (en excluant celui qui vient d'échouer). Ce choix peut être fait de manière aléatoire ou selon une heuristique. Les nogoods collectés sont conservés d'un redémarrage à l'autre. Si tous les ordres ont été testés sans succès, le problème est déclaré insatisfiable. L'algorithme met alors à jour l'ordre courant $\sigma_{\text{current}} \leftarrow \sigma_{\text{new}}$, réinitialise le compteur, et recommence la boucle. En sortie, il retourne soit une solution valide, soit une indication que le problème est insatisfiable, ou éventuellement un dépassement de temps.

Les foctions et procédures auxiliaires utilisées dans l'algorithme sont sont les même utilisées dans l'algorithme FC-GHD+NG+DR [15]. Elles sont décrites brièvement ci-dessous :

1. *Record_nogood*(n_i) (Algorithme 12) Lorsqu'un retour-arrière se produit du nœud n_i vers son parent n_p, la procédure `Record_nogood` est invoquée. Son rôle est d'enregistrer un nogood structurel, c'est-à-dire l'assignation des variables du séparateur entre n_p et n_i ayant mené à l'échec dans le sous-arbre enraciné en n_i. Cette information est stockée pour éviter d'explorer à nouveau cette configuration infructueuse si la recherche doit reconsidérer ce sous-arbre à l'avenir.

Algorithm 12 *Procedure Record_nogood*

Input n_i : noeud

1: $S \leftarrow \chi(Parent(n_i)) \cap \chi(n_i)$
2: $nogoods(Parent(n_i)) \longleftarrow nogoods(Parent(n_i)) \cup sol(P_{Parent(n_i)})[S]$

2. *Reorder_hypertree*(n_i, σ) (Algorithme 13) Cette procédure est utilisée dans des variantes avancées telles que FC-GHD+NG+DR, lorsqu'un blocage persistant est détecté dans le sous-arbre enraciné en n_i. `Reorder_hypertree` modifie dynamiquement l'ordre de parcours σ de la GHD en repositionnant le sous-arbre problématique pour qu'il soit traité immédiatement après son parent. Cela permet d'affronter plus tôt les sources potentielles de conflit, améliorant ainsi l'efficacité de la détection des incohérences.

Algorithm 13 *Procedure Reorder_hypertree*

In n_i : noeud,
Inout σ : liste

1: $cn_1 \longleftarrow Parent(n_i)$
2: $cn_2 \longleftarrow n_i$
3: **while** $cn_2 \neq \emptyset$ **do**
4: **if** $cn_2 \notin nodes(T_{n_i}$ **then**
5: Break
6: **end if**
7: $cn_3 \longleftarrow succ(cn_2)$
8: inserer cn_2 just après cn_1 in σ
9: $cn_2 \longleftarrow cn_3$
10: $cn_1 \longleftarrow succ(cn_1)$
11: **end while**

3. *Backtrack_DR*(n_i) (Algorithme 14) Lorsqu'un échec survient au niveau du nœud n_i, cette procédure gère le retour-arrière tout en intégrant la logique de réordonnancement dynamique. Elle commence par restaurer l'état de la recherche, puis appelle `Record_nogood` pour mémoriser le conflit détecté. Ensuite, elle décide si un appel à `Reorder_hypertree` est nécessaire. Si tel est le cas, l'ordre de la GHD est ajusté avant de revenir au nœud parent ou, si aucun autre choix n'est possible, d'initier un redémarrage global de la recherche.

Algorithm 14 *Procedure Backtrack-DR*

Inout n_i : noeud,
Inout σ : liste

1: $Restore\_removed\_tuples(n_i)$
2: $Record\_nogood(n_i)$
3: $Reorder\_hypertree(n_i)$
4: $n_i \longleftarrow Parent(n_i)$

4. *Filter_NG*(sol(P_{n_i})) (Algorithme 15) Avant de propager une solution partielle sol(P_{n_i}) trouvée pour le nœud n_i, la procédure `Filter_NG` vérifie si l'assignation des variables du séparateur entre n_i et son parent (induite par la solution trouvée) correspond à un nogood déjà enregistré. Si tel est le cas, cette voie de recherche est immédiatement élaguée, évitant d'explorer des branches reconnues comme conflictuelles.

Algorithm 15 *Procedure Filter-NG*

Inout n_i : noeud

1: **if** $\neg Nogood(sol(P_{n_i}))$ **then**
2: **if** $C_i \in \lambda(T_{n_i}, compatible(sol(P_{n_i})), Rel(C_i))$ **then**
3: $Filter\_child\_nodes(n_i)$
4: $n_i \longleftarrow succ(n_i)$
5: **end if**
6: **end if**

Exemple 10 *Considérons une instance de CSP n-aire avec une GHD donnée dans la Figure 3.1.*

Initialement l'ordre de parcours choisi est $\sigma_1 = (n_1, n_2, n_3, n_4, n_5, n_6)$*, avec* n_1 *comme racine. Supposons que lors de l'exploration avec ce premier ordre, l'algorithme rencontre un échec au niveau du nœud* n_3 *après un certain nombre de backtracks. La procédure* `Record_nogood` *est alors appelée pour enregistrer le nogood structurel correspondant à l'assignation des variables du séparateur entre* n_2 *et* n_3 *qui a conduit à cet échec.*
Ensuite, comme le nombre de backtracks a atteint le seuil défini, un redémarrage

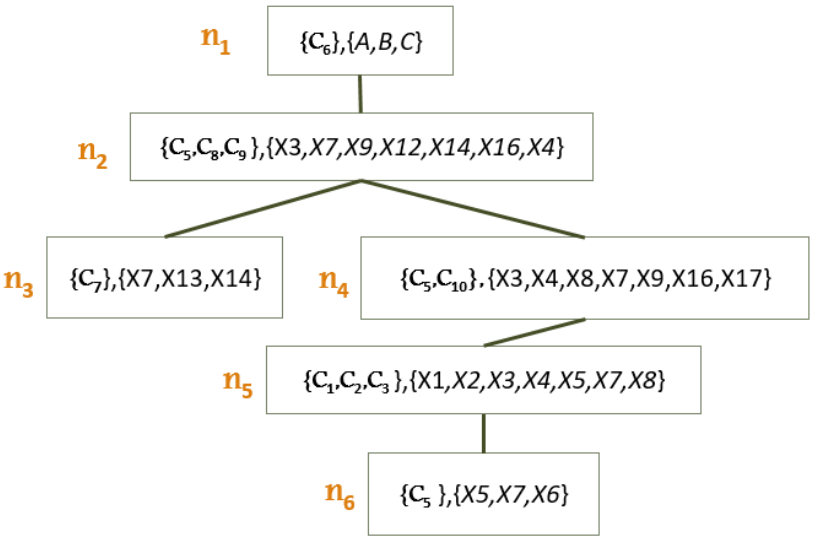

FIGURE 3.1 – Une GHD pour une instance de CSP n-aire, avec un ordre de parcours initial σ_1.

est initié. Un nouvel ordre de parcours $\sigma_2 = (n_2, n_3, n_4, n_5, n_6, n_1)$ *(voir la Figure 3.2) est sélectionné, avec* n_2 *comme nouvelle racine. La recherche reprend avec ce nouvel ordre, en bénéficiant du nogood précédemment enregistré pour éviter de retomber dans la même configuration infructueuse.*

Si lors de cette nouvelle exploration, une solution est trouvée au niveau du nœud n_5, elle est immédiatement retournée. Sinon, si un nouvel échec survient, le processus de redémarrage peut se répéter avec un autre ordre de parcours, jusqu'à ce qu'une solution soit trouvée ou que tous les ordres aient été explorés sans succès.

3.4.3 Complétude de l'algorithme *Restart-FC-GHD+NG+DR*

L'algorithme *Restart-FC-GHD+NG+DR* repose sur le cœur algorithmique *FC-GHD+NG+DR*, démontré comme complet [15]. Cela signifie qu'il est capable de trouver une solution si elle existe, ou de prouver l'insatisfiabilité si aucune solution n'est possible, à condition de disposer de ressources suffisantes.

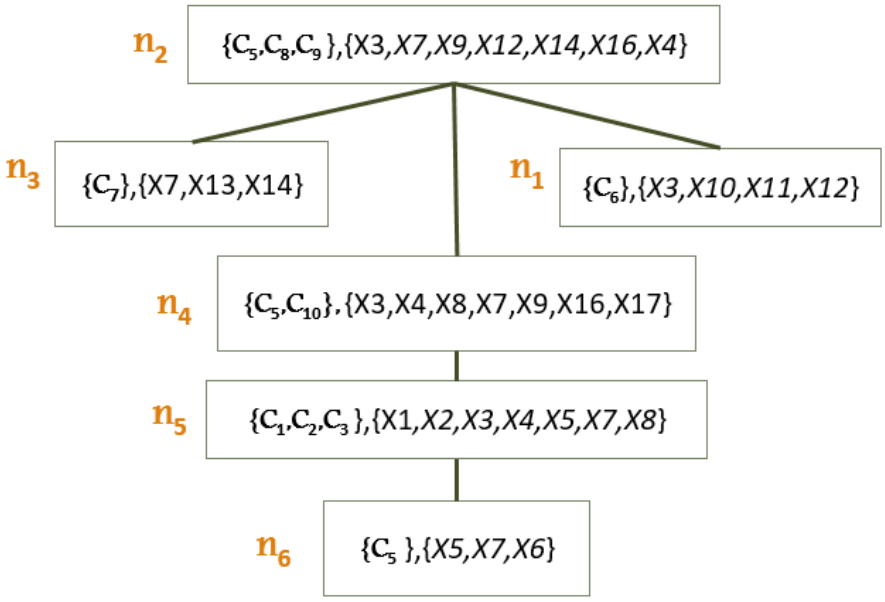

FIGURE 3.2 – Nouvel ordre de parcours σ_2 après redémarrage, avec n_2 comme racine.

La stratégie de redémarrage introduit un mécanisme de contrôle du seuil de retour-arrière *limit_backtracks* et permet de modifier dynamiquement l'ordre de parcours des clusters dans la GHD. Malgré ces ajouts, la complétude de l'approche reste garantie sous certaines conditions.

1. **Exploration exhaustive des ordres :** Si l'ensemble *ORDERS* contient tous les ordres possibles induits par tous les choix de racines de la GHD, et que chaque ordre peut être exploré avec un budget de retour-arrière suffisamment grand, alors la complétude est assurée. Une solution sera inévitablement trouvée via au moins un ordre, ou l'insatisfiabilité sera prouvée après exploration de tous les ordres.

2. **Préservation des nogoods :** Les nogoods découverts sont conservés entre les phases de recherche, ce qui n'entrave pas la complétude. Au contraire, cela permet d'éviter de réexplorer des configurations déjà reconnues comme infructueuses, ce qui accélère la recherche.

3. **Limites pratiques :** En pratique, des contraintes comme un temps maximal ou un nombre maximal de redémarrages peuvent être imposées. Si ces limites sont atteintes sans qu'une réponse soit trouvée, l'algorithme devient incomplet dans ce contexte restreint. Toutefois, le mécanisme reste fondamentalement complet en l'absence de telles restrictions.

En résumé, *Restart-FC-GHD+NG+DR* est un algorithme complet dans un cadre théorique où chaque ordre peut être exploré avec des ressources adéquates. Les redémarrages et la mémorisation des nogoods améliorent l'efficacité sans compromettre cette complétude. Les éventuelles limites de ressources n'affectent que le comportement pratique, non la validité théorique de l'approche.

3.5 Résultats expérimentaux

Dans cette section, nous présentons les résultats expérimentaux de l'évaluation des algorithmes *Restart-FC-GHD+NG+DR* sur des instances de CSP n-aires consistante et inconsistante du monde reel. Nous comparerons notre algorithme avec l'algorithme de base *FC-GHD+NG+DR* pour évaluer l'impact des stratégies de redémarrage sur la performance et la robustesse de la résolution des CSP.
Nous avons utilisé les instances de CSP n-aires issues des benchmark que nous allons décrire dans ce qui suit, l'implémentation de notre méthode a été faite en utilisant le langage C++, et les expérimentations ont été éffectué sur une machine Core i5-10400 CPU @ 2.90 GHz with 8 GB of RAM under Linux Ubuntu 22.04.2 LTS. pour le calcule de la GHD nous avons utiliser l'heuristiques BE, qui est l'une des meilleures permettant d'obtenir une GHD quasi optimales dans un temps CPU raisonnable.
Dans les tableaux, $|\mathcal{X}|$ représente le nombre de variables, $|\mathcal{C}|$ le nombre de contraintes, r le nombre de tuples dans les relations des contraintes, et w la largeure moyenne de la GHD. Les temps d'exécution sont donnés en secondes.

3.5.1 Description des benchmarks

Une sélection de benchmarks issus des compétitions internationales de solveurs CSP (CPAI'08, CPAI'09) [1] a été utilisée. Ces benchmarks peuvent être classés comme suit :

1. Instances Structurées Issues d'Applications Réelles :

   - Série Renault : Cette série comprend deux instances (renault-ext et renault-mgd-ext) dérivées du problème de configuration de la Renault Mégane. Elles sont caractérisées par un grand nombre de variables (101), un nombre important de contraintes (134 et 113 respectivement), et des relations de contraintes de grande taille (la plus grande contenant 48,721 tuples), reflétant la complexité des problèmes industriels.
   - Série Renault-Mod (Renault Modifié) : Cette classe contient 50 instances structurées, également basées sur le problème de configuration Renault, mais avec des variations incluant des domaines de variables pouvant atteindre jusqu'à 42 valeurs possibles. La taille des relations de contraintes peut également être très grande (jusqu'à 48,721 tuples). Le nombre de variables est typiquement de 108 ou 111, et le nombre de contraintes de 149 ou 154.

2. Instances Quasi-Aléatoires : Ces instances combinent une génération aléatoire avec une certaine structure sous-jacente.

   - Série Pret : Comprend 8 instances qui encodent des problèmes de 2-coloriage, conçues pour être insatisfiables. Elles impliquent 60 ou 150 variables, 40 ou 100 contraintes, avec une arité maximale des contraintes de 3 (problèmes 3-SAT) et des relations contenant 4 tuples.

1. http ://www.cril.univ-artois.fr/CPAI08/
http ://www.cril.univ-artois.fr/CPAI09/

– Série Dubois : Contient 13 instances 3-SAT insatisfiables, générées aléatoirement. Le nombre de variables varie de 60 à 300, et le nombre de contraintes de 40 à 200. Chaque relation de contrainte contient 4 tuples.

– Série VarDimacs : Ces instances proviennent de la formalisation SAT de problèmes classiques. Elles incluent 4 instances insatisfiables issues de l'analyse de fautes de circuits. Ces problèmes ont des arités de contraintes supérieures à 2, et la plus grande relation peut contenir jusqu'à 1023 tuples.

3.5.2 Comparaison de Restart-FC-GHD+NG+DR avec FC-GHD+NG+DR

Cette section présente l'évaluation expérimentale de l'algorithme Restart-FC-GHD+NG+DR en le comparant à l'algorithme FC-GHD+NG+DR sur les instances présentées dans la section précédente.

- **Résultats sur les instances normalized Renault-serie** (Table 3.1)

  Sur deux instances issues d'applications réelles et très structurées, *Restart* montre un léger avantage sur FC-GHD. Pour *renault_ext*, les temps d'exécution sont de 0.6s contre 0.83s, et pour *renault-mgd_ext*, 0.7s contre 0.96s. Ces différences restent modestes, ce qui est attendu dans le cas de problèmes bien structurés, où la décomposition hypertree (GHD) permet déjà une résolution rapide. Toutefois, même dans ce cadre favorable, la capacité à redémarrer avec un nouvel ordre de traitement semble offrir un chemin de résolution parfois plus direct, montrant que la diversification peut être bénéfique, même de manière marginale.

Problems normalized	size			w	*FC-GHD+NG+DR*	*Restart-FC-GHD+NG+DR*
Renault-serie	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r		Time	Time
-renault_ext	101	134	48,721	3	0.83	**0.6**
-renault-mgd_ext	101	113	48,721	2	0.96	**0.7**

TABLE 3.1 – Comparaison de FC-GHD+NG+DR et Restart-FC-GHD+NG+DR sur les problèmes Renault-serie

- **Résultats sur les instances normalized Renault-Mod** (Table 3.2)

  Cette série d'instances plus vaste permet une analyse plus représentative. Sur les 49 cas testés, *Restart-FC-GHD+NG+DR* surpasse FC-GHD, l'avantage est particulièrement marqué sur les instances satisfiables, ces résultats suggèrent que la capacité à varier l'ordre de parcours aide à trouver une solution plus rapidement, probablement en évitant certaines branches de recherche peu prometteuses que l'ordre initial pourrait suivre trop longtemps.

Problems normalized	size			w	*FC-GHD+NG+DR*	*Restart-FC-GHD+NG+DR*	Consistence
Renault-mod	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r		Time	Time	
-0_ext	111	147	48,721	4	0.26	**0.17**	Oui
-1_ext	111	147	48,721	3	2.84	6.95	Non
-2_ext	111	151	48,721	5	0.57	**0.44**	Oui
-3_ext	111	147	48,721	3	1.87	**1.86**	Non
-4_ext	111	147	48,721	4	0.38	**0.29**	Oui
-5_ext	111	147	48,721	3	4.85	14.15	Non
-6_ext	111	147	48,721	3	0.25	**0.24**	Non
-7_ext	111	147	48,721	4	0.72	1.15	Oui
-8_ext	111	147	48,721	3	0.21	**0.20**	Non
-9_ext	111	147	48,721	3	0.19	**0.18**	Oui
-10_ext	111	149	48,721	3	2.32	**2.21**	Non
-11_ext	111	149	48,721	3	0.38	**0.37**	Oui
-12_ext	111	149	48,721	3	10.86	36.53	Non
-13_ext	111	149	48,721	3	0.43	**0.41**	Oui
-14_ext	111	149	48,721	3	2.73	7.12	Non
-15_ext	111	149	48,721	3	1.33	3.84	Non
-16_ext	111	149	48,721	3	3.12	**3,01**	Non
-17_ext	111	149	48,721	3	0.51	**0.44**	Non
-18_ext	111	149	48,721	3	0.57	**0.004**	Non
-19_ext	111	149	48,721	3	0.29	**0.25**	Non
-20_ext	111	159	48,721	3	3.23	3.23	Non
-21_ext	111	159	48,721	3	15.79	127.80	Non
-22_ext	111	159	48,721	3	8.83	9,36	Non
-23_ext	111	159	48,721	4	0.5	**0.41**	Non
-24_ext	111	159	48,721	4	0.98	**0.95**	Non
-25_ext	111	159	48,721	3	17.04	45.97	Non
-27_ext	111	159	48,721	3	0.54	**0.49**	Non
-28_ext	111	159	48,721	3	20.29	20.99	Non
-29_ext	111	159	48,721	4	4.88	**2.8**	Non
-30_ext	111	154	48,721	3	1.36	2.97	Non
-31_ext	111	154	48,721	3	0.47	**0,46**	Oui
-32_ext	111	154	48,721	4	1.56	4.41	Oui
-33_ext	111	154	48,721	5	3.66	3.66	Non
-34_ext	111	154	48,721	4	2.17	3.45	Oui
-35_ext	111	154	48,721	3	6.11	14,61	Non
-36_ext	111	154	48,721	4	8.62	18.76	Oui
-37_ext	111	154	48,721	4	3.16	7.93	Non
-38_ext	111	154	48,721	4	0.41	0.62	Oui
-39_ext	111	154	48,721	4	17.31	213.65	Non
-40_ext	108	149	48,721	3	2.24	6.31	Non
-41_ext	108	149	48,721	4	2.46	5,89	Oui
-42_ext	108	149	48,721	3	0.30	**0.28**	Non
-43_ext	108	149	48,721	3	0.48	**0.37**	Oui
-44_ext	108	149	48,721	4	0.25	**0.22**	Oui
-45_ext	108	149	48,721	4	6.29	14.1	Oui
-46_ext	108	149	48,721	4	1.88	**0.29**	Oui
-47_ext	108	149	48,721	4	0.16	**0.07**	Non
-48_ext	108	149	48,721	4	6.63	11,87	Oui
-49_ext	108	149	48,721	4	8.28	29.09	Oui

TABLE 3.2 – Comparaison de FC-GHD+NG+DR et Restart-FC-GHD+NG+DR sur les problèmes Renault-mod

En revanche, sur les instances insatisfiables, les résultats sont plus contrastés. Restart-FC surpasse FC-GHD sur certaines, parfois de manière spectaculaire comme `-18 ext`, mais est moins performant sur d'autres, telles que `-5 ext` ou *-12 ext*. Cela montre que si le premier ordre de FC-GHD est bien adapté pour explorer efficacement tout l'espace de recherche, les redémarrages peuvent parfois introduire un surcoût. Néanmoins, ils sont également capables d'éviter des situations où un mauvais ordre initial rendrait la recherche extrêmement lente. Le choix de la valeur de *limit_backtracks* joue ici un rôle crucial et mériterait d'être ajusté dynamiquement pour maximiser les bénéfices du redémarrage.

- **Résultats sur les instances normalized Dubois Series (Table 3.3) et Pret Series (Table 3.4)**

  Pour ces deux séries d'instances booléennes insatisfiables, Restart-FC-GHD-NG+DR résout tous les cas en un temps quasi nul, tandis que FC-GHD-NG+DR affiche des temps très faibles mais mesurables (0.007s en moyenne pour Pret, 0.0035s pour Dubois). Ces résultats suggèrent que les deux algorithmes sont particulièrement efficaces sur ces problèmes très simples, et que le coût du mécanisme de redémarrage est négligeable dans ces contextes. Il est probable qu'un des premiers ordres essayés par Restart-FC suffise à prouver l'insatisfiabilité presque instantanément.

Problems normalized	size			w	*FC-GHD+NG+DR*	*Restart-FC-GHD+NG+DR*	Consistence
Dubois	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r		Time	Time	
20_ext	60	40	4	2	0.043	$\simeq 0$	Non
21_ext	63	42	4	2	0.005	$\simeq 0$	Non
22_ext	66	44	4	2	0.004	$\simeq 0$	Non
23_ext	69	46	4	2	0.005	$\simeq 0$	Non
24_ext	72	48	4	2	0.005	$\simeq 0$	Non
25_ext	75	50	4	2	0.005	$\simeq 0$	Non
26_ext	78	52	4	2	0.006	$\simeq 0$	Non
27_ext	81	54	4	2	0.006	$\simeq 0$	Non
28_ext	84	56	4	2	0.006	$\simeq 0$	Non
29_ext	87	58	4	2	0.007	$\simeq 0$	Non
30_ext	90	60	4	2	0.006	$\simeq 0$	Non
50_ext	150	100	4	2	0.011	$\simeq 0$	Non
100_ext	300	200	4	2	0.049	$\simeq 0$	Non

TABLE 3.3 – Comparaison de FC-GHD+NG+DR et Restart-FC-GHD+NG+DR sur les problèmes normalized Dubois

- **Résultats sur les instances normalized VarDimacs Series**(Table 3.5)

  Sur cette série, Restart-FC est significativement plus rapide sur trois des quatre instances évaluées. Par exemple, *bf-1355-075 ext* est résolue en 0.81s contre 9.74s, *bf-1355-638 ext* en $\simeq 0s$ contre 0.18s, et *bf-2670-001 ext* en 0.29s contre 0.31s. FC-GHD n'est meilleur que sur une instance : *bf-0432-007_ext*, avec un temps de 35.15s contre 129.87s pour Restart-FC. Ces résultats confirment que le redémarrage peut être très bénéfique lorsque l'ordre initial est mal choisi,

Problems normalized	size			w	*FC-GHD+NG+DR*	*Restart-FC-GHD+NG+DR*	Consistence
Renault-pret	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r		Time	Time	
60-25_ext	60	40	4	5	0.36	$\simeq 0$	Non
60-40_ext	60	40	4	5	0.008	$\simeq 0$	Non
60-60_ext	60	40	4	5	0.01	$\simeq 0$	Non
60-75_ext	60	40	4	5	0.01	$\simeq 0$	Non
150-25_ext	150	100	4	5	0.05	$\simeq 0$	Non
150-40_ext	150	100	4	5	0.17	$\simeq 0$	Non
150-60_ext	150	100	4	5	0.37	$\simeq 0$	Non
150-75_ext	150	100	4	5	0.023	$\simeq 0$	Non

Table 3.4 – Comparaison de FC-GHD+NG+DR et Restart-FC-GHD+NG+DR sur les problèmes Renault Pret series

mais qu'il peut également être contre-productif si l'ordre initial de FC-GHD est déjà optimal. Cela renforce l'idée qu'une politique de redémarrage plus fine, adaptée à la nature du problème et à la progression de la recherche, pourrait améliorer encore les performances globales.

Problems normalized	size			w	*FC-GHD+NG+DR*	*Restart-FC-GHD+NG+DR*	Consistence
VarDimacs	$\|\mathcal{X}\|$	$\|\mathcal{C}\|$	r		Time	Time	
VarDimacs-bf-0432-007_ext	970	1,943	31	29	35.15	129.87	Oui
VarDimacs-bf-1355-075_ext	1,818	2,049	5	5	9.74	**0.81**	Oui
VarDimacs-bf-1355-638_ext	532	339	31	2	0.18	$\simeq 0$	Oui
VarDimacs-bf-2670-001_ext	1,244	1,354	31	7	0.31	**0.29**	Non

Table 3.5 – Comparaison de FC-GHD+NG+DR et Restart-FC-GHD+NG+DR sur les problèmes normalized VarDimacs

3.5.3 Discussion Générale et Avantages de l'Approche par Redémarrage

Afin de faciliter la compréhension des tendances globales extraites des tableaux de résultats précédents, nous présentons une série de figures synthétisant les performances comparatives de nos algorithmes. La figure 3.3 illustre la part de victoires (instances où l'algorithme est le plus rapide) agrégée sur l'ensemble des familles de benchmarks. Nous observons une domination nette de l'approche Restart-FC-GHD+NG+DR, particulièrement marquée sur les instances inconsistentes (insatisfiables) où elle l'emporte dans 69,2 % des cas, contre 58,3 % pour les instances consistentes. Cette représentation met en lumière la capacité de la stratégie de redémarrage à accélérer la preuve d'insatisfiabilité en diversifiant les perspectives d'exploration de l'espace de recherche.

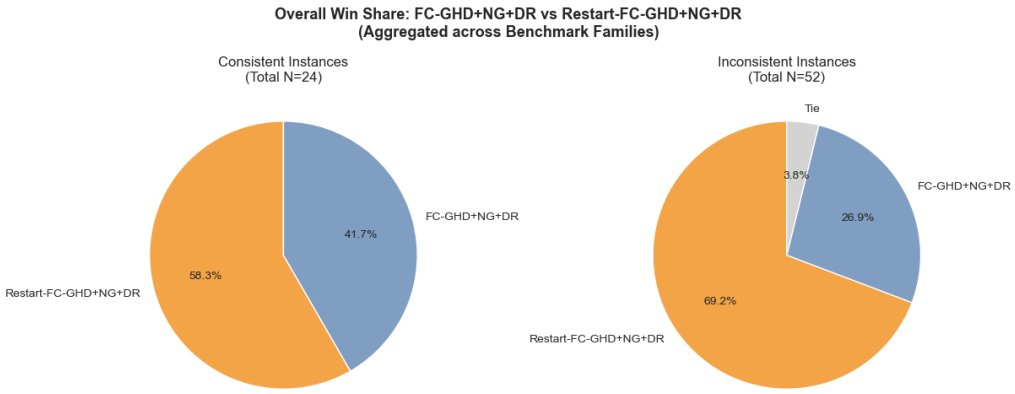

FIGURE 3.3 – Part de victoires de Restart-FC-GHD+NG+DR vs FC-GHD+NG+DR sur les différentes familles d'instances

L'impact de la dimension du problème sur l'efficacité de la résolution est détaillé dans les figures 3.4 et 3.5. La figure 3.4 montre que notre algorithme avec redémarrage conserve l'avantage sur la majorité des tailles de contraintes, avec une performance notable pour $|C| = 149$. Toutefois, la figure révèle également que pour certaines configurations spécifiques, notamment $|C| = 154$, l'ordre initial de l'algorithme de base reste supérieur, suggérant que le surcoût du mécanisme de restart n'est pas toujours compensé si la racine initiale est déjà optimale. Parallèlement, la figure 3.5 confirme la robustesse de notre approche sur les instances à 111 variables, qui représentent le cœur de notre jeu de tests, où le mécanisme de redémarrage permet de surpasser systématiquement la version statique.

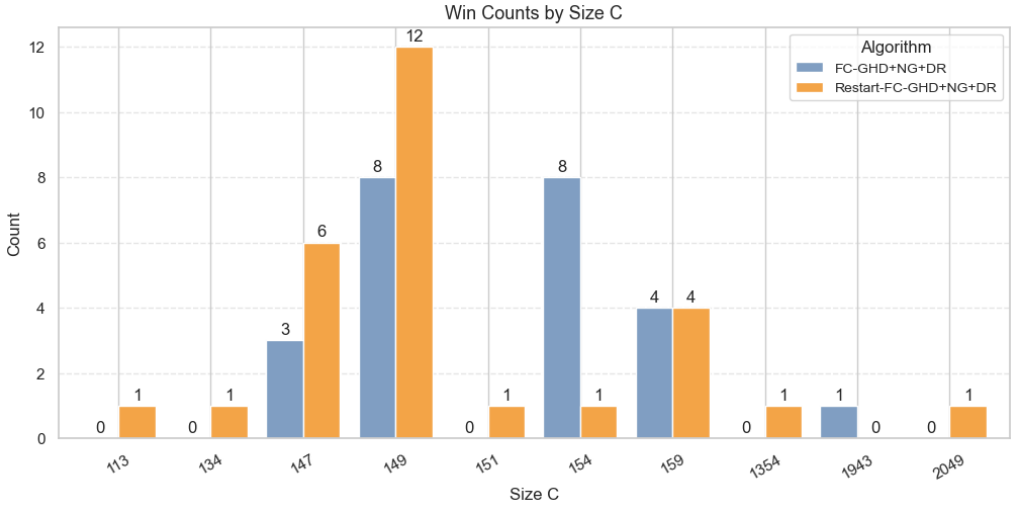

FIGURE 3.4 – Restart-FC-GHD+NG+DR vs FC-GHD+NG+DR en fonction de la taille des contraintes $|C|$

Enfin, l'analyse de l'influence des paramètres structurels et de la taille des données est complétée par les figures 3.6 et 3.7. La figure 3.6 indique que pour une largeur d'hyperarbre $W = 3$, l'algorithme avec restart est significativement plus efficace avec 16 victoires recensées. Cependant, pour $W = 4$, l'algorithme de réfé-

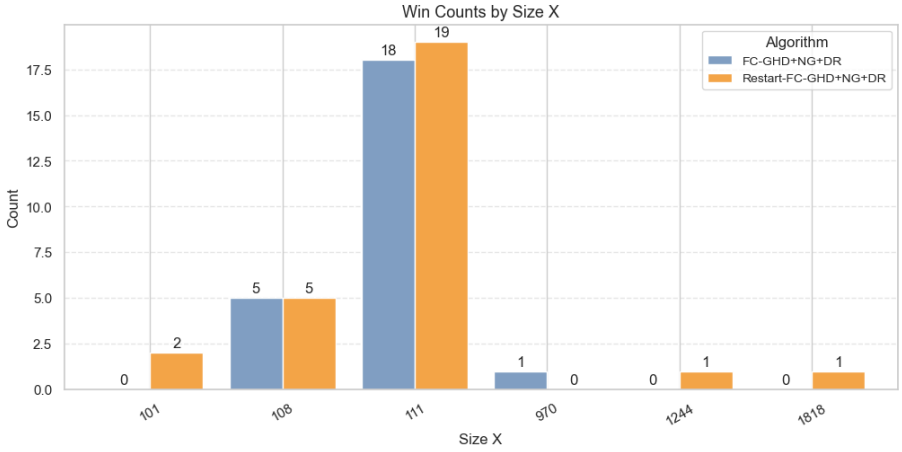

FIGURE 3.5 – Restart-FC-GHD+NG+DR vs FC-GHD+NG+DR en fonction de la taille des variables $|X|$

rence maintient une certaine compétitivité, ce qui souligne une corrélation entre la complexité des clusters et l'efficacité des redémarrages. Concernant la taille des relations, la figure 3.6 illustre les performances sur les instances série Renault. Sur ces problèmes complexes caractérisés par $r = 48721$ tuples, l'approche par redémarrage confirme sa supériorité avec 26 victoires, prouvant que la diversification par changement de racine est un levier de performance fiable, même face à des contraintes de très grande arité.

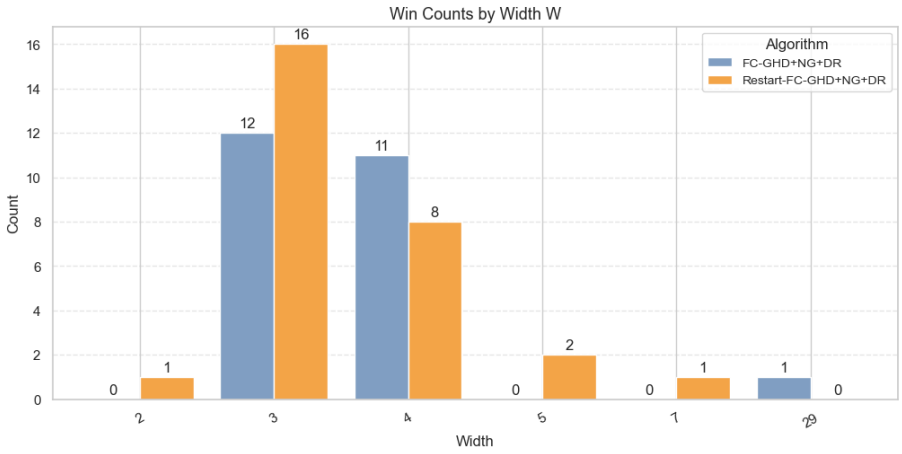

FIGURE 3.6 – Restart-FC-GHD+NG+DR vs FC-GHD+NG+DR en fonction de la largeur d'hyperarbre W

L'intégration d'une stratégie de redémarrage dans les algorithmes basés sur GHD, comme FC-GHD+NG+DR, se révèle être une avancée significative pour la résolution des CSP n-aires. En variant dynamiquement l'ordre de traitement des clusters, les approches basées sur le redémarrage [106][107] améliorent notablement les performances, en particulier sur les instances satisfiables (ex : série *Renault-Mod*), où elles facilitent une découverte plus rapide des solutions en évitant les blocages liés à un ordre initial défavorable. Pour les instances insatisfiables, l'effet est plus nuancé, bénéfique si l'ordre initial est mauvais, il peut induire un léger surcoût sinon. Cela souligne l'importance d'optimiser les politiques de redémarrage, notamment le seuil

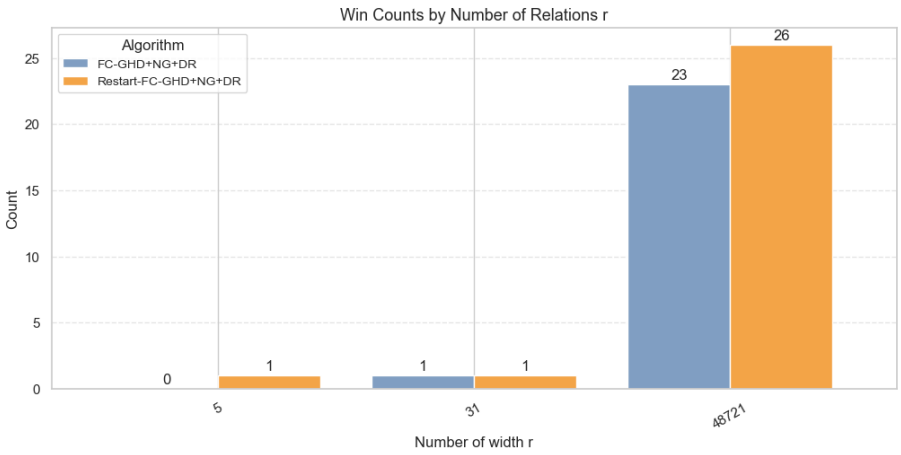

FIGURE 3.7 – Restart-FC-GHD+NG+DR vs FC-GHD+NG+DR en fonction de la taille des relations r

limit_backtracks. Néanmoins, l'approche par redémarrage confère une robustesse accrue, évitant les performances catastrophiques dues à un mauvais ordre initial et stabilisant les temps de résolution.

L'efficacité est renforcée par la préservation des nogoods à travers les redémarrages, assurant un apprentissage cumulatif qui limite la réexploration de conflits. Bien que dépendante de la qualité de la GHD initiale, la stratégie de redémarrage permet une exploitation plus intelligente de la structure décomposée. Le surcoût du mécanisme est minime sur les instances simples et compensé par les gains sur les cas complexes.

En conclusion, la combinaison du redémarrage avec la gestion des nogoods dans les solveurs GHD offre un excellent équilibre entre diversification de la recherche et exploitation des connaissances, améliorant la robustesse, l'efficacité et la stabilité globales de la résolution.

3.6 Conclusion

En conclusion, la contribution présentée dans ce chapitre montre que le paradigme du redémarrage, lorsqu'il est judicieusement intégré à des algorithmes exploitant la structure des CSP via des décompositions comme la GHD, constitue une piste prometteuse pour le développement de solveurs plus performants et plus robustes. Ces résultats ouvrent la voie à d'autres investigations, notamment sur l'optimisation des politiques de redémarrage et leur combinaison avec des techniques d'apprentissage plus avancées, qui seront discutées plus amplement dans la conclusion générale et les perspectives de cette thèse.

Conclusion générale

Cette thèse s'est attaquée au défi persistant de la résolution efficace des Problèmes de Satisfaction de Contraintes (CSP), en particulier ceux impliquant des contraintes n-aires et possédant des structures complexes. L'objectif principal était d'explorer et d'améliorer les techniques de résolution basées sur la Décomposition Hypertree Généralisée (GHD), un cadre théorique puissant mais dont l'application pratique peut être sensible à divers facteurs initiaux.
Pour ce faire, nous avons débuté par une analyse approfondie des fondements des CSP et un examen critique de l'état de l'art des méthodes de résolution. Cette revue a couvert les algorithmes de recherche systématique classiques et leurs améliorations, les techniques cruciales d'inférence et de filtrage, les stratégies de décomposition structurelle avec un accent particulier sur la GHD ainsi que les approches de recherche locale et les contributions émergentes de l'apprentissage profond. Cette étude contextuelle a permis d'identifier les forces et les faiblesses des approches existantes, notamment la sensibilité des méthodes basées sur GHD à la qualité de la décomposition et à l'ordre de traitement initial des clusters.
Nous avons ensuite détaillé le fonctionnement des algorithmes FC-GHD et de ses variantes FC-GHD+NG et FC-GHD+NG+DR, qui combinent une recherche de type forward-checking avec la structure guidée par la GHD, l'apprentissage de nogoods structurels et le réordonnancement dynamique des sous-arbres. Ces algorithmes ont servi de point de départ et de base de comparaison pour nos contributions.

La contribution centrale de cette thèse réside dans la conception, la mise en œuvre et l'évaluation rigoureuse d'une nouvelle stratégie de résolution pour les CSP n-aires basée sur l'intégration du redémarrage (restart) au sein des algorithmes exploitant la GHD. Nos principaux apports peuvent être résumés comme suit :

1. Développement d'un algorithme avec redémarrage et gestion des nogoods $Restart - FC - GHD + NG + DR$: Nous avons proposé des mécanismes qui permettent d'interrompre périodiquement la recherche guidée par la GHD lorsque celle-ci s'avère peu productive. Lors d'un redémarrage, un nouvel ordre de traitement des clusters de la GHD est sélectionné en choisissant un nouveau nœud racine, offrant ainsi une nouvelle perspective d'exploration de l'espace de recherche.

2. Exploitation cumulative des connaissances : Une caractéristique essentielle de nos approches est la préservation des nogoods structurels à travers les redémarrages. Cette stratégie garantit que les informations sur les conflits identifiés lors des phases de recherche précédentes ne sont pas perdues et continuent

d'élaguer l'espace de recherche, même après un changement global de l'ordre de traitement.

3. Validation expérimentale approfondie : L'efficacité de nos algorithmes a été démontrée par des expérimentations complètes sur des benchmarks reconnus de CSP n-aires. Les résultats indiquent que notre approche avec redémarrage surpassent significativement les performances de l'algorithme FC-GHD+NG+DR sur de nombreuses instances, en particulier les problèmes satisfiables et ceux présentant une structure complexe. Ces améliorations se traduisent par une réduction du temps de résolution et une plus grande robustesse face aux mauvais choix initiaux.

4. Analyse de l'impact du redémarrage : Nous avons fourni des éléments d'analyse sur les conditions dans lesquelles le redémarrage est le plus bénéfique, soulignant son rôle dans la diversification de la recherche et sa capacité à aider le solveur à s'échapper des régions difficiles de l'espace de recherche.

Malgré ces résultats prometteurs, certaines limitations demeurent. L'efficacité du redémarrage dépend fortement de paramètres tels que le seuil de retour-arrière initial, la politique de mise à jour de ce seuil, ou encore la manière dont le nouvel ordre de traitement est choisi. Une exploration plus systématique de ces paramètres pourrait améliorer encore les performances. De plus, bien que la GHD ne soit calculée qu'une seule fois, son coût initial peut devenir un facteur limitant sur des instances de grande taille. L'accumulation de nogoods, quant à elle, peut engendrer un surcoût en mémoire ou en temps si elle n'est pas maîtrisée, la gestion dynamique de ces informations reste un axe à approfondir. Enfin, nos expérimentations, bien qu'appuyées sur des benchmarks standards, gagneraient à être étendues à des problèmes issus d'environnements applicatifs réels plus diversifiés.

Perspectives de recherche

Les travaux réalisés ouvrent plusieurs directions de recherche prometteuses pour prolonger et enrichir les résultats obtenus :

- **Apprentissage adaptatif des redémarrages :** Utiliser des techniques d'apprentissage automatique, notamment par renforcement, pour apprendre dynamiquement quand redémarrer, avec quel seuil, et comment choisir la nouvelle racine ou l'ordre de parcours, en fonction des caractéristiques de l'instance et de l'historique de la recherche.

- **Optimisation de la politique de seuils :** Étudier des stratégies de redémarrage plus sophistiquées, comme les séquences de Luby ou des ajustements adaptatifs non linéaires, pour permettre une exploration plus équilibrée entre profondeur et diversification.

- **Choix informé des ordres de parcours :** Développer des heuristiques de sélection de racine fondées sur l'analyse des nogoods accumulés ou des métriques locales de difficulté dans la GHD.

- **Amélioration du filtrage et de la gestion des nogoods :** Intégrer des structures de données plus efficaces et des stratégies de filtrage dynamique pour réduire le coût lié à la gestion d'un grand nombre de nogoods.
- **Hybridation avec d'autres approches :** Étudier l'intégration de méthodes de recherche locale entre les redémarrages, ou comme mécanismes d'affinage des solutions déjà trouvées.

Bibliographie

[1] U. Montanari, « Networks of constraints : Fundamental properties and applications to picture processing, » *Information sciences*, t. 7, p. 95-132, 1974.

[2] S. Choudhury, J. K. Gupta, M. J. Kochenderfer, D. Sadigh et J. Bohg, « Dynamic multi-robot task allocation under uncertainty and temporal constraints, » *Autonomous Robots*, t. 46, n° 1, p. 231-247, 2022.

[3] J. K. Behrens, R. Lange et M. Mansouri, « A constraint programming approach to simultaneous task allocation and motion scheduling for industrial dual-arm manipulation tasks, » in *2019 International Conference on Robotics and Automation (ICRA)*, IEEE, 2019, p. 8705-8711.

[4] F. S. Alhaijawy et A. M. Florea, « Scheduling People's Daily Activities Using Temporal Constraints Satisfaction Problem, » in *2018 5th International Conference on Control, Decision and Information Technologies (CoDIT)*, IEEE, 2018, p. 876-881.

[5] P. D. Vascik et R. J. Hansman, « Scaling constraints for urban air mobility operations : Air traffic control, ground infrastructure, and noise, » in *2018 aviation technology, integration, and operations conference*, 2018, p. 3849.

[6] M. Stojadinović, « Air traffic controller shift scheduling by reduction to CSP, SAT and SAT-related Problems, » in *International Conference on Principles and Practice of Constraint Programming*, Springer, 2014, p. 886-902.

[7] O. Nempont, J. Atif et I. Bloch, « A constraint propagation approach to structural model based image segmentation and recognition, » *Information Sciences*, t. 246, p. 1-27, 2013.

[8] K. Petridis, S. Bloehdorn, C. Saathoff, N. Simou, S. Dasiopoulou, V. Tzouvaras, S. Handschuh, Y. Avrithis, Y. Kompatsiaris et S. Staab, « Knowledge representation and semantic annotation of multimedia content, » *IEE Proceedings-Vision, Image and Signal Processing*, t. 153, n° 3, p. 255-262, 2006.

[9] C. Saathoff et S. Staab, « Exploiting spatial context in image region labelling using fuzzy constraint reasoning, » in *2008 Ninth International Workshop on Image Analysis for Multimedia Interactive Services*, IEEE, 2008, p. 16-19.

[10] G. Fouquier, J. Atif et I. Bloch, « Sequential spatial reasoning in images based on pre-attention mechanisms and fuzzy attribute graphs, » in *ECAI 2008*, IOS Press, 2008, p. 611-615.

[11] F. Goodarzian, S. F. Wamba, K. Mathiyazhagan et A. Taghipour, « A new bi-objective green medicine supply chain network design under fuzzy environment : Hybrid metaheuristic algorithms, » *Computers & industrial engineering*, t. 160, p. 107 535, 2021.

[12] M. Karimi et Y. Shen, « iCFN : an efficient exact algorithm for multistate protein design, » *Bioinformatics*, t. 34, nº 17, p. i811-i820, 2018.

[13] M. Ruffini, J. Vucinic, S. de Givry, G. Katsirelos, S. Barbe et T. Schiex, « Guaranteed diversity & quality for the weighted CSP, » in *2019 IEEE 31st international conference on tools with artificial intelligence (ICTAI)*, IEEE, 2019, p. 18-25.

[14] C. Viricel, S. de Givry, T. Schiex et S. Barbe, « Cost function network-based design of protein–protein interactions : predicting changes in binding affinity, » *Bioinformatics*, t. 34, nº 15, p. 2581-2589, 2018.

[15] Z. Habbas, K. Amroun et D. Singer, « A forward-checking algorithm based on a generalised hypertree decomposition for solving non-binary constraint satisfaction problems, » *Journal of Experimental & Theoretical Artificial Intelligence*, t. 27, nº 5, p. 649-671, 2015.

[16] C. P. Gomes, B. Selman et H. Kautz, « Boosting combinatorial search through randomization, » *AAAI/IAAI*, t. 98, nº 1998, p. 431-437, 1998.

[17] T. Walsh, « Search in a small world, » in *Ijcai*, Citeseer, t. 99, 1999, p. 1172-1177.

[18] Z. Ayadi, W. Boulila, I. R. Farah, A. Leborgne et P. Gancarski, « Resolution methods for constraint satisfaction problem in remote sensing field : A survey of static and dynamic algorithms, » *Ecological Informatics*, t. 69, p. 101 607, 2022.

[19] M. R. Garey et D. S. Johnson, *Computer and Intractability : A Guide to the NP-Completeness, vol. 238*, 1979.

[20] E. C. Freuder, « A sufficient condition for backtrack-free search, » *Journal of the ACM (JACM)*, t. 29, nº 1, p. 24-32, 1982.

[21] C. Nash, « Four colours suffice : how the map problem was solved, » *The Mathematical Intelligencer*, t. 25, nº 4, p. 80-83, 2003.

[22] M. Dohmen, « A survey of constraint satisfaction techniques for geometric modeling, » *Computers & Graphics*, t. 19, nº 6, p. 831-845, 1995.

[23] R. Barták, M. A. Salido et F. Rossi, « New trends in constraint satisfaction, planning, and scheduling : a survey, » *The Knowledge Engineering Review*, t. 25, nº 3, p. 249-279, 2010.

[24] F. Ait Hatrit et K. Amroun, « From Backtracking To Deep Learning : A Survey On Methods For Solving Constraint Satisfaction Problems, » *ITEGAM-JETIA*, t. 11, nº 51, p. 119-126, 2025.

[25] R. M. Haralick et G. L. Elliott, « Increasing tree search efficiency for constraint satisfaction problems, » *Artificial intelligence*, t. 14, nº 3, p. 263-313, 1980.

[26] R. Dechter et I. Meiri, « Experimental evaluation of preprocessing algorithms for constraint satisfaction problems, » *Artificial Intelligence*, t. 68, n° 2, p. 211-241, 1994.

[27] C. Bessiere et J.-C. Régin, « MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems, » in *International conference on principles and practice of constraint programming*, Springer, 1996, p. 61-75.

[28] D. Frost et R. Dechter, « Look-ahead value ordering for constraint satisfaction problems, » in *IJCAI (1)*, Citeseer, 1995, p. 572-578.

[29] R. Dechter et J. Pearl, « Network-based heuristics for constraint-satisfaction problems, » *Artificial intelligence*, t. 34, n° 1, p. 1-38, 1987.

[30] S. W. Golomb et L. D. Baumert, « Backtrack programming, » *Journal of the ACM (JACM)*, t. 12, n° 4, p. 516-524, 1965.

[31] J. R. Bitner et E. M. Reingold, « Backtrack programming techniques, » *Communications of the ACM*, t. 18, n° 11, p. 651-656, 1975.

[32] R. Dechter, *Constraint processing*. Elsevier, 2003.

[33] J. G. Gaschnig, *Performance measurement and analysis of certain search algorithms.* Carnegie Mellon University, 1979.

[34] P. Prosser, « Hybrid algorithms for the constraint satisfaction problem, » *Computational intelligence*, t. 9, n° 3, p. 268-299, 1993.

[35] R. Dechter, « Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition, » *Artificial Intelligence*, t. 41, n° 3, p. 273-312, 1990.

[36] J. J. McGregor, « Relational consistency algorithms and their application in finding subgraph and graph isomorphisms, » *Information Sciences*, t. 19, n° 3, p. 229-250, 1979.

[37] M. Shahraeeni, « Enhanced multiple-point statistical simulation with backtracking, forward checking and conflict-directed backjumping, » *Mathematical Geosciences*, t. 51, n° 2, p. 155-186, 2019.

[38] C. Bessiere, P. Meseguer, E. C. Freuder et J. Larrosa, « On forward checking for non-binary constraint satisfaction, » *Artificial Intelligence*, t. 141, n° 1-2, p. 205-224, 2002.

[39] T. Schiex et G. Verfaillie, « Nogood recording for static and dynamic constraint satisfaction problems, » *International Journal on Artificial Intelligence Tools*, t. 3, n° 02, p. 187-207, 1994.

[40] M. L. Ginsberg, « Dynamic backtracking, » *Journal of artificial intelligence research*, t. 1, p. 25-46, 1993.

[41] R. Dechter et D. Frost, « Backjump-based backtracking for constraint satisfaction problems, » *Artificial Intelligence*, t. 136, n° 2, p. 147-188, 2002.

[42] C. Bessiere, « Constraint propagation, » in *Foundations of Artificial Intelligence*, t. 2, Elsevier, 2006, p. 29-83.

[43] R. H. Yap, W. Xia et R. Wang, « Generalized arc consistency algorithms for table constraints : A summary of algorithmic ideas, » in *Proceedings of the AAAI conference on artificial intelligence*, t. 34, 2020, p. 13 590-13 597.

[44] A. K. Mackworth, « Consistency in networks of relations, » *Artificial intelligence*, t. 8, n° 1, p. 99-118, 1977.

[45] R. Mohr et T. C. Henderson, « Arc and path consistency revisited, » *Artificial intelligence*, t. 28, n° 2, p. 225-233, 1986.

[46] P. Van Hentenryck, Y. Deville et C.-M. Teng, « A generic arc-consistency algorithm and its specializations, » *Artificial intelligence*, t. 57, n° 2-3, p. 291-321, 1992.

[47] C. Bessiere, « Arc-consistency and arc-consistency again, » *Artificial intelligence*, t. 65, n° 1, p. 179-190, 1994.

[48] C. Bessiere et J.-C. Régin, « Refining the Basic Constraint Propagation Algorithm., » in *IJCAI*, t. 1, 2001, p. 309-315.

[49] C. Bessiere, J.-C. Régin, R. H. Yap et Y. Zhang, « An optimal coarse-grained arc consistency algorithm, » *Artificial Intelligence*, t. 165, n° 2, p. 165-185, 2005.

[50] M. Arangú, M. A. Salido et F. Barber, « AC2001-OP : An arc-consistency algorithm for constraint satisfaction problems, » in *Trends in Applied Intelligent Systems : 23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2010, Cordoba, Spain, June 1-4, 2010, Proceedings, Part III 23*, Springer, 2010, p. 219-228.

[51] C.-C. Han et C.-H. Lee, « Comments on Mohr and Henderson's path consistency algorithm, » *Artificial Intelligence*, t. 36, n° 1, p. 125-130, 1988.

[52] M. C. Cooper, « An optimal k-consistency algorithm, » *Artificial Intelligence*, t. 41, n° 1, p. 89-95, 1989.

[53] D. Sabin et E. C. Freuder, « Contradicting conventional wisdom in constraint satisfaction, » in *International Workshop on Principles and Practice of Constraint Programming*, Springer, 1994, p. 10-20.

[54] J.-C. Régin, « Maintaining arc consistency algorithms during the search without additional space cost, » in *Principles and Practice of Constraint Programming-CP 2005 : 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005. Proceedings 11*, Springer, 2005, p. 520-533.

[55] H. Li, « Narrowing support searching range in maintaining arc consistency for solving constraint satisfaction problems, » *IEEE access*, t. 5, p. 5798-5803, 2017.

[56] P. Prosser, « Mac-cbj : maintaining arc consistency with con ict-directed backjumping, » *Submitted to ECAI-96*, 1995.

[57] G. Gottlob, N. Leone et F. Scarcello, « A comparison of structural CSP decomposition methods, » *Artificial Intelligence*, t. 124, n° 2, p. 243-282, 2000.

[58] H. Marc, « On the universal relation, » Technical report, University of Toronto, rapp. tech., 1979.

[59] M. Yannakakis, « Algorithms for acyclic database schemes, » in *VLDB*, t. 81, 1981, p. 82-94.

[60] C. Beeri, R. Fagin, D. Maier et M. Yannakakis, « On the desirability of acyclic database schemes, » *Journal of the ACM (JACM)*, t. 30, nº 3, p. 479-513, 1983.

[61] R. E. Tarjan et M. Yannakakis, « Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs, » *SIAM Journal on computing*, t. 13, nº 3, p. 566-579, 1984.

[62] P. Jégou et C. Terrioux, « Tree-decompositions with connected clusters for solving constraint networks, » in *International Conference on Principles and Practice of Constraint Programming*, Springer, 2014, p. 407-423.

[63] P. Jégou et C. Terrioux, « Combining restarts, nogoods and bag-connected decompositions for solving CSPs, » *Constraints*, t. 22, p. 191-229, 2017.

[64] N. Robertson et P. D. Seymour, « Graph minors. II. Algorithmic aspects of tree-width, » *Journal of algorithms*, t. 7, nº 3, p. 309-322, 1986.

[65] S. Arnborg, D. G. Corneil et A. Proskurowski, « Complexity of finding embeddings in ak-tree, » *SIAM Journal on Algebraic Discrete Methods*, t. 8, nº 2, p. 277-284, 1987.

[66] E. C. Freuderl, « Complexity of K-Tree Structurecl Constraint Satisfaction Problems, » 1990, p. 4-9.

[67] P. Jégou et C. Terrioux, « Hybrid backtracking bounded by tree-decomposition of constraint networks, » *Artificial Intelligence*, t. 146, nº 1, p. 43-75, 2003.

[68] G. Gottlob, N. Leone et F. Scarcello, « Robbers, marshals, and guards : game theoretic and logical characterizations of hypertree width, » in *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, 2001, p. 195-206.

[69] G. Gottlob et M. Samer, « A backtracking-based algorithm for hypertree decomposition, » *Journal of Experimental Algorithmics (JEA)*, t. 13, p. 1-1, 2009.

[70] Z. Younsi, K. Amroun, F. Bouarab-Dahmani et S. Bennai, « HSJ-Solver : a new method based on GHD for answering conjunctive queries and solving constraint satisfaction problems, » *Applied Intelligence*, t. 53, nº 13, p. 17 226-17 239, 2023.

[71] E. Aarts et L. JK, *Local Search in Combinatorial Optimization, eds*, 2003.

[72] M. Pasaribu, Y. Pratama, A. M. Sianipar, G. A. Sihombing et Y. Purba, « Implementation of Backtracking and Steepest Ascent Hill Climbing Algorithms on Ferry Scheduling in Lake Toba, » in *2022 IEEE International Conference of Computer Science and Information Technology (ICOSNIKOM)*, IEEE, 2022, p. 1-7.

[73] M. Bohlin, Y. Lu, J. Kraft, P. Kreuger et T. Nolte, « Best-effort simulation-based timing analysis using hill-climbing with random restarts, » in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, IEEE Computer Society*, 2009.

[74] S. Kirkpatrick, C. D. Gelatt Jr et M. P. Vecchi, « Optimization by simulated annealing, » *science*, t. 220, nº 4598, p. 671-680, 1983.

[75] V. Černỳ, « Thermodynamical approach to the traveling salesman problem : An efficient simulation algorithm, » *Journal of optimization theory and applications*, t. 45, p. 41-51, 1985.

[76] S. Minton, M. D. Johnston, A. B. Philips et P. Laird, « Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method, » in *Proceedings of the eighth National conference on Artificial intelligence-Volume 1*, 1990, p. 17-24.

[77] M. Gelain, M. S. Pini, F. Rossi, K. B. Venable et T. Walsh, « A local search approach to solve incomplete fuzzy CSPs, » in *International Conference on Agents and Artificial Intelligence*, SciTePress, t. 2, 2011, p. 582-585.

[78] A. Manno, E. Amaldi, F. Casella et E. Martelli, « A local search method for costly black-box problems and its application to CSP plant start-up optimization refinement, » *Optimization and Engineering*, t. 21, p. 1563-1598, 2020.

[79] F. Glover, « Tabu search—part II, » *ORSA Journal on computing*, t. 2, nº 1, p. 4-32, 1990.

[80] J. H. Holand, « Adaptation in natural and artificial systems, » *Ann Arbor : The University of Michigan Press*, t. 32, 1975.

[81] D. E. Goldberg, « Genetic Algorithm in Search, Optimization and Machine Learning, Addison, » *W esley Publishing Company, R eading, MA*, t. 1, nº 98, p. 9, 1989.

[82] H. R. Lourenço, O. C. Martin et T. Stützle, « Iterated local search, » in *Handbook of metaheuristics*, Springer, 2003, p. 320-353.

[83] P. Hansen et N. Mladenović, « Variable neighborhood search : Principles and applications, » *European journal of operational research*, t. 130, nº 3, p. 449-467, 2001.

[84] A. M. Frisch, T. J. Peugniez, A. J. Doggett et P. W. Nightingale, « Solving non-boolean satisfiability problems with stochastic local search : A comparison of encodings, » in *SAT 2005 : Satisfiability Research in the Year 2005*, Springer, 2006, p. 143-179.

[85] B. Selman, H. A. Kautz et B. Cohen, « Local search strategies for satisfiability testing., » *Cliques, coloring, and satisfiability*, t. 26, p. 521-532, 1993.

[86] Y. Bengio, A. Lodi et A. Prouvost, « Machine learning for combinatorial optimization : a methodological tour d'horizon, » *European Journal of Operational Research*, t. 290, nº 2, p. 405-421, 2021.

[87] A. Lodi et G. Zarpellon, « On learning and branching : a survey, » *Top*, t. 25, nº 2, p. 207-236, 2017.

[88] H. Xu, S. Koenig et T. S. Kumar, « Towards effective deep learning for constraint satisfaction problems, » in *International Conference on Principles and Practice of Constraint Programming*, Springer, 2018, p. 588-597.

[89] W. Song, Z. Cao, J. Zhang, C. Xu et A. Lim, « Learning variable ordering heuristics for solving constraint satisfaction problems, » *Engineering Applications of Artificial Intelligence*, t. 109, p. 104 603, 2022.

[90] J. C. Ortiz-Bayliss, I. Amaya, J. M. Cruz-Duarte, A. E. Gutierrez-Rodriguez, S. E. Conant-Pablos et H. Terashima-Marín, « A general framework based on machine learning for algorithm selection in constraint satisfaction problems, » *Applied Sciences*, t. 11, n° 6, p. 2749, 2021.

[91] Y. Xu, D. Stern et H. Samulowitz, « Learning adaptation to solve constraint satisfaction problems, » *Proceedings of Learning and Intelligent Optimization (LION)*, p. 14, 2009.

[92] Z. Yang, A. Ishay et J. Lee, « Learning to solve constraint satisfaction problems with recurrent transformer, » *arXiv preprint arXiv :2307.04895*, 2023.

[93] A. Galassi, M. Lombardi, P. Mello et M. Milano, « Model agnostic solution of CSPs via deep learning : A preliminary study, » in *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Springer, 2018, p. 254-262.

[94] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner et G. Monfardini, « The graph neural network model, » *IEEE transactions on neural networks*, t. 20, n° 1, p. 61-80, 2008.

[95] T. N. Kipf et M. Welling, « Semi-supervised classification with graph convolutional networks, » *arXiv preprint arXiv :1609.02907*, 2016.

[96] K. Xu, W. Hu, J. Leskovec et S. Jegelka, « How powerful are graph neural networks ? » *arXiv preprint arXiv :1810.00826*, 2018.

[97] R. S. Sutton et A. G. Barto, *Reinforcement learning : An introduction*. MIT press Cambridge, 1998, t. 1.

[98] F. Chalumeau, I. Coulon, Q. Cappart et L.-M. Rousseau, « Seapearl : A constraint programming solver guided by reinforcement learning, » in *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Springer, 2021, p. 392-409.

[99] J. Tönshoff, B. Kisin, J. Lindner et M. Grohe, « One model, any CSP : graph neural networks as fast global search heuristics for constraint satisfaction, » *arXiv preprint arXiv :2208.10227*, 2022.

[100] G. Gottlob, C. Okulmus et R. Pichler, « Fast and parallel decomposition of constraint satisfaction problems, » *Constraints*, t. 27, n° 3, p. 284-326, 2022.

[101] A. Kaznatcheev, D. Cohen et P. Jeavons, « Representing fitness landscapes by valued constraints to understand the complexity of local search, » *Journal of Artificial Intelligence Research*, t. 69, p. 1077-1102, 2020.

[102] V. P. Guddeti, « A dynamic restart strategy for randomized BT search, » in *Principles and Practice of Constraint Programming–CP 2004 : 10th International Conference, CP 2004, Toronto, Canada, September 27-October 1, 2004. Proceedings 10*, Springer, 2004, p. 796-796.

[103] C. Lecoutre, L. Sais, S. Tabary et V. Vidal, « Recording and minimizing nogoods from restarts, » *Journal on Satisfiability, Boolean Modelling and Computation*, t. 1, nº 3-4, p. 147-167, 2006.

[104] H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou et S. Tabary, « Learning variable ordering heuristics with multi-armed bandits and restarts, » in *ECAI 2020*, IOS Press, 2020, p. 371-378.

[105] F. Koriche, C. Lecoutre, A. Paparrizou et H. Wattez, « Best heuristic identification for constraint satisfaction, » in *31st International Joint Conference on Artificial Intelligence (IJCAI'22)*, International Joint Conferences on Artificial Intelligence Organization, 2022, p. 1859-1865.

[106] F. Ait Hatrit et K. Amroun, « Solving non-binary constraint satisfaction problems using GHD and restart., » *ITEGAM-JETIA*, t. 11, nº 51, p. 72-79, 2025.

[107] F. Ait Hatrit et K. Amroun, « Enhancing Constraint Satisfaction Problem Solving with a Restart-Nogood-Based Approach, » in *Proceedings of the First International Conference on Artificial Intelligence, Smart Technologies and Communications (AISTC 2025)*, Springer Nature, t. 197, 2025, p. 141.

Résumé

Les problèmes de satisfaction de contraintes (CSP) constituent un cadre formel essentiel en intelligence artificielle pour modéliser une large gamme de problèmes combinatoires. Toutefois, leur résolution reste un défi majeur en raison de leur complexité intrinsèque (NP-complète). Cette thèse explore différentes approches de résolution des CSP, en mettant un accent particulier sur les techniques de décomposition structurelle, notamment la Décomposition Hypertree Généralisée (GHD), particulièrement adaptée aux contraintes d'arité élevée. Après une étude approfondie des méthodes existantes, recherche systématique (Backtracking et ses variantes), techniques d'inférence (consistance d'arc, consistance de chemin), recherche locale et approches par apprentissage profond, ce travail propose une amélioration aux algorithmes guidés par la GHD. L'algorithme FC-GHD est introduit, ainsi que deux extensions : FC-GHD+NG, intégrant l'apprentissage de nogoods structurels, et FC-GHD+NG+DR, basé sur un réordonnancement dynamique des clusters. La contribution principale de cette thèse réside dans l'introduction d'une stratégie de redémarrage adaptatif, visant à pallier la sensibilité des algorithmes GHD à la racine initiale. L'algorithme Restart-FC-GHD+NG+DR permet de diversifier l'exploration en sélectionnant dynamiquement un nouveau nœud racine tout en réutilisant les nogoods appris lors des phases précédentes. Une évaluation expérimentale sur des benchmarks standards de CSP n-aires démontre une amélioration significative des performances et de la robustesse de algorithme proposé, notamment pour les instances satisfiables et les structures complexes. Ce travail ouvre également des perspectives vers l'intégration de techniques d'apprentissage automatique pour guider dynamiquement les stratégies de redémarrage et d'ordonnancement des clusters.

Mots clés : Problèmes de Satisfaction de Contraintes, CSP, Décomposition Hypertree Généralisée, GHD, Algorithmes de résolution, Nogoods structurels, Redémarrage, Algorithmes systématiques, Apprentissage automatique

Abstract

Constraint Satisfaction Problems (CSPs) form a fundamental formal framework in artificial intelligence for modeling a wide range of combinatorial problems. However, solving them remains a major challenge due to their intrinsic complexity (NP-complete). This thesis explores various approaches for solving CSPs, with a particular focus on structural decomposition techniques, especially Generalized Hypertree Decomposition (GHD), which is well-suited to high-arity constraints. After a thorough review of existing methods systematic search (Backtracking and its variants), inference techniques (arc consistency, path consistency), local search, and deep learning approaches this work proposes improvements to GHD-guided algorithms. The FC-GHD algorithm is introduced, along with two extensions: FC-GHD+NG, which incorporates structural nogood learning, and FC-GHD+NG+DR, which is based on dynamic cluster reordering. The main contribution of this thesis lies in the introduction of an adaptive restart strategy designed to address the sensitivity of GHD-based algorithms to the initial root. The Restart-FC-GHD+NG+DR algorithm diversifies the search by dynamically selecting a new root node while reusing nogoods learned during previous phases. An experimental evaluation on standard n-ary CSP benchmarks demonstrates a significant improvement in the performance and robustness of the proposed algorithm, particularly on satisfiable instances and complex structures. This work also opens up new directions toward the integration of machine learning techniques to dynamically guide restart and cluster ordering strategies.

Keywords: Constraint Satisfaction Problems, CSP, Generalized Hypertree Decomposition, GHD, Solving Algorithms, Structural Nogoods, Restart, Systematic Algorithms, Machine Learning.

ملخص

تُعدّ مشاكل الرضا عن القيود (CSP) إطارًا شكليًا أساسيًا في الذكاء الاصطناعي لنمذجة مجموعة واسعة من المشكلات التوليفية. ومع ذلك، فإن حلّها لا يزال يشكّل تحديًا كبيرًا بسبب تعقيدها الداخلي(NP -كاملة).تستعرض هذه الأطروحة مختلف أساليب حلّ مشكلات الـCSP، مع تركيز خاص على تقنيات التحليل البنيوي، وخاصة التحليل العام لشجرة الارتباط (GHD) ، والذي يتناسب بشكل خاص مع القيود ذات الأريّة العالية.بعد دراسة معمّقة للأساليب الموجودة مثل البحث المنهجي (Backtrackingو تعديلاته)، تقنيات الاستدلال (اتساق القوس، اتساق المسار)، البحث المحلي، والمقاربات المعتمدة على التعلم العميق يقترح هذا العمل تحسينات على الخوارزميات المستندة إلى .GHD تُقدَّم خوارزمية FC-GHD ، إلى جانب امتدادين لها FC-GHD+NG :التي تدمج تعلّم nogoods البنيوية، و-FC GHD+NG+DRالمعتمدة على إعادة ترتيب ديناميكي للعناقيد. وتتمثل المساهمة الرئيسية لهذه الأطروحة في تقديم استراتيجية إعادة تشغيل تكيفية تهدف إلى معالجة حساسية خوارزميات GHD تجاه التحليل الأولي. تتيح خوارزمية Restart-FC-GHD+NG+DR تنويع عملية البحث عبر اختيار جذر جديد ديناميكيًا مع إعادة استخدام الـ Nogoods المكتسبة في المراحل السابقة .وقد أظهرت التقييمات التجريبية على معايير قياسية لمشاكل CSP n-ary تحسنًا ملحوظًا في الأداء والموثوقية، لا سيما في الحالات القابلة للرضا والهياكل المعقدة. ويفتح هذا العمل آفاقًا مستقبلية نحو دمج تقنيات التعلم الآلي لتوجيه استراتيجيات إعادة التشغيل وترتيب العناقيد بطريقة ديناميكية.

مفاتيح الكلمات :مشاكل إرضاء القيود ، تقسيم الشجرة الموسعة العامة ، خوارزميات الحل ، القيود الهيكلية غير الجيدة ، إعادة التشغيل ، خوارزميات منهجية، التعلم الآلي