Titre: Title:	Guider la recherche grâce au passage de messages pour les problèmes d'optimisation sous contraintes
Auteur: Author:	Axel Baudot
Date:	2025
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Baudot, A. (2025). Guider la recherche grâce au passage de messages pour les problèmes d'optimisation sous contraintes [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/68857/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/68857/
Directeurs de recherche: Advisors:	Gilles Pesant
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Guider la recherche grâce au passage de messages pour les problèmes d'optimisation sous contraintes

AXEL BAUDOT
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Septembre 2025

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Guider la recherche grâce au passage de messages pour les problèmes d'optimisation sous contraintes

présenté par **Axel BAUDOT**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Quentin CAPPART, président
Gilles PESANT, membre et directeur de recherche
Antoine LEGRAIN, membre

DÉDICACE

À mes parents,

REMERCIEMENTS

Ce mémoire marque la fin de trois ans d'efforts qui dépassent de loin le cadre académique : la décision de s'installer à Montréal, la procédure d'immigration, le retour à l'université, l'éloignement de ma famille, la prise de nouveaux repères... Autant d'étapes qui font que cette centaine de pages me semble une réalisation bien anecdotique face aux bouleversements de ces dernières années. Merci aux membres du jury pour le temps que vous consacrerez à ce manuscrit, je vous en souhaite une bonne lecture.

Et à vous qui m'avez accompagné durant cette période éprouvante...

Merci Gilles pour ton investissement, ta bienveillance et ton encadrement impeccable, de notre première séance de cours à la remise de ce mémoire. Je te remercie également de m'avoir offert ta confiance et des opportunités d'enseignement, qui sont pour moi des cadeaux inestimables. Ma gratitude est immense et ne saurait être exprimée adéquatement en quelques lignes.

Merci Erwan pour ton soutien indéfectible, indispensable, indescriptible.

Merci Ismael de me rappeler de prendre soin de moi et de rester fidèle à moi-même.

Merci Théau d'avoir été si bon camarade dans le dépaysement et l'excellence académique.

Merci Chloé pour ta relecture attentive, et d'avoir ensoleillé cet été de rédaction.

Merci Basile et Maxime, je dois mes débuts en recherche à votre écoute et vos encouragements.

Merci à tous les compagnons de mon aventure montréalaise, que vous ayez été présents en personne ou en paroles. Vous êtes si nombreux ! Votre anonymat ne tient aucunement d'un manque d'appréciation pour vous, mais surtout des douleurs qui me labourent les avant-bras après un mois de dactylographie effrénée.

Merci Mina pour nos moments "en famille", je t'aurais gardée à Montréal si je l'avais pu.

Merci à mes parents, pour tout...

RÉSUMÉ

Cette recherche évalue l'application du passage de messages aux problèmes d'optimisation sous contraintes (COPs). Alors que le passage de messages a prouvé son efficacité pour les problèmes de satisfaction de contraintes (CSPs) avec l'algorithme Sum-Product, son utilité pour l'optimisation restait à démontrer.

L'objectif principal consiste à introduire et évaluer l'algorithme Max-Product comme alternative à Sum-Product pour l'optimisation. L'hypothèse centrale propose que Max-Product peut fournir des marginales désignant les valeurs participant aux solutions optimales.

La méthodologie introduit une contrainte Oracle appliquée à la variable objectif, envoyant des messages proportionnels aux valeurs pour biaiser favorablement les meilleures solutions. Cette contrainte s'avère indispensable pour Max-Product qui produit sinon des marginales uniformes équivalentes à la cohérence d'arc généralisée.

Des algorithmes de calcul de messages Max-Product sont développés pour quatre contraintes globales. Pour AllDifferent, le calcul se réduit au problème d'assignation dans un graphe biparti, optimisé pour réutiliser des couplages intermédiaires. Pour les contraintes Table, Sum et Regular, les algorithmes Sum-Product existants sont adaptés en remplaçant l'accumulation par somme par un maximum.

L'évaluation empirique sur 321 exemplaires du corpus XCSP utilise le score normalisé moyen comme métrique principale. L'heuristique *DomWDeg + Max Marginal* surpasse toutes les alternatives. La recherche à divergence limitée, la réutilisation des marginales entre phases de propagation et l'interruption anticipée basée sur l'entropie améliorent l'efficacité.

Les résultats démontrent l'efficacité du passage de messages pour l'optimisation avec des améliorations de 15% comparées aux méthodes sans passage de messages, montrant que le coût accru de l'inférence est compensé par le guidage pertinent de la recherche. Sum-Product avec un faible nombre d'itérations excelle pour les résolutions courtes, tandis qu'un nombre élevé d'itérations devient avantageux sur les résolutions prolongées. Max-Product constitue une bonne alternative à Sum-Product si le temps de résolution est incertain.

Nous concluons que le passage de messages constitue une méthode d'inférence avancée efficace pour l'optimisation sous contraintes. Si Sum-Product démontre généralement une supériorité sur Max-Product, l'étude étend l'applicabilité de MiniCPBP aux COPs et ouvre des perspectives pour la résolution de problèmes d'optimisation combinatoire.

ABSTRACT

This research evaluates the application of message passing to Constraint Optimization Problems (COPs). While message passing has proven effective for Constraint Satisfaction Problems (CSPs) using the Sum-Product algorithm, its usefulness for optimization remained undemonstrated.

The primary objective introduces and evaluates the Max-Product algorithm as an alternative to Sum-Product for optimization. The central hypothesis proposes that Max-Product can provide marginals identifying values participating in optimal solutions.

The methodology introduces an Oracle constraint applied to the objective variable, sending messages proportional to values to bias toward better solutions. This constraint proves essential for Max-Product, which otherwise produces uniform marginals equivalent to generalized arc consistency.

Max-Product message computation algorithms are developed for four global constraints. For AllDifferent, computation reduces to the assignment problem in bipartite graphs, optimized to reuse intermediate matchings. For Table, Sum, and Regular constraints, existing Sum-Product algorithms are adapted by replacing sum accumulation with maximum operations.

An empirical evaluation on 321 XCSP corpus instances uses normalized mean score as the primary metric. The *DomWDeg + Max Marginal* heuristic outperforms all alternatives. Limited discrepancy search, marginal reuse between propagation phases, and entropy-based early termination improve efficiency.

Results demonstrate message passing effectiveness for optimization with 15% improvement over non-message-passing methods, showing that increased inference costs are offset by improved search guidance. Sum-Product with few iterations excels for short resolutions, while high iteration counts become advantageous for extended resolutions. Max-Product provides a compromise when resolution time is uncertain.

We conclude that message passing constitutes an effective advanced inference method for constraint optimization. While Sum-Product generally demonstrates superiority over Max-Product, this study extends MiniCPBP's applicability to COPs and opens perspectives for solving combinatorial optimization problems.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

AC4	Arc Consistency 4, cohérence d'arc 4
BP	Belief Propagation, propagation de conviction
CFN	Cost Function Network, réseau de fonctions de Coût
COP	Constrained Optimization Problem, problème d'optimisation sous constraintes
CP	Constraint Programming, programmation par contraintes
CPBP	Constraint Programming with Belief Propagation, programmation par contraintes avec propagation de conviction
CSP	Constraint Satisfaction Problem, problème de satisfaction de constraintes
DCOP	Distributed Constrained Optimization Problem, problème distribué d'optimisation sous contraintes
MaxSD	Maximal Solution Density, densité maximale de solution
OSAC	Optimal Soft Arc Consistency
PC1	Path Consistency 1, cohérence de chemin 1
SAC	Soft Arc Consistency
VAC	Virtual Arc Consistency
VCSP	Valued Constraint Satisfaction Problem

CHAPITRE 1 INTRODUCTION

Les problèmes combinatoires constituent un défi central dans de nombreux secteurs industriels : optimisation de tournées de véhicules en logistique, ordonnancement de tâches en production manufacturière, allocation de ressources en télécommunication... Résoudre ces problèmes efficacement représente un enjeu économique majeur.

Ces problèmes admettent deux formalisations principales. Les problèmes de satisfaction de contraintes (CSP) cherchent toute affectation de valeurs aux variables qui respecte l'ensemble des contraintes imposées. Les problèmes d'optimisation de contraintes (COP) ajoutent une fonction objectif à optimiser.

1.1 Problématique

Malheureusement, ces problèmes sont NP-difficiles et les traiter peut nécessiter beaucoup de temps dans le pire des cas. C'est pourquoi leur résolution implique des méthodes sophistiquées telles que la programmation par contraintes (CP), une approche pour résoudre les problèmes combinatoires qui comporte trois aspects distincts :

- Un langage de haut niveau pour modéliser le problème, où chaque contrainte permet de décrire explicitement une grande partie de sa structure combinatoire.
- La propagation de support et le filtrage.
- La recherche à retour arrière.

Ces deux derniers éléments sont les mécanismes principaux de la résolution.

La propagation de support et le filtrage constitue une étape d'inférence : le recours à des raisonnements et déductions logiques pour progresser dans la résolution. Vis-à-vis d'une contrainte, lorsqu'un couple variable/valeur fait partie d'un tuple accepté par la contrainte, le tuple est appelé un support de ce couple. Des algorithmes de filtrage associés à chaque contrainte peuvent déterminer si un couple ne possède pas de support. Le cas échéant, on déduit qu'assigner cette valeur à cette variable violera nécessairement la contrainte.

La recherche avec retour arrière implique une série de décisions réduisant les domaines des variables non affectées, menant soit à des solutions soit à des incohérences. Elle permet l'exploration exhaustive de tout l'espace de recherche défini par les variables et leurs domaines.

Les problèmes de satisfaction et optimisation sous contraintes peuvent être résolus en ne s'appuyant que sur la recherche ou sur l'inférence mais au prix d'une complexité temporelle

exponentielle [1] en fonction du nombre de variables et contraintes impliquées. En CP, associer les deux crée une synergie où la recherche déclenche un filtrage supplémentaire, qui à son tour permet un élagage de l'arbre de recherche. En CP, la recherche a été étendue avec divers mécanismes. Par exemple, on peut enregistrer et expliquer les décisions qui mènent à l'échec pour éviter de les rencontrer à nouveau [2] [3]. Lorsqu'on rencontre une incohérence, nous pouvons remonter plus loin dans l'arbre de recherche pour une exploration plus efficace [4].

D'autre part, améliorer l'inférence consiste généralement à trouver de meilleurs algorithmes de filtrage pour atteindre des niveaux de cohérence plus élevés à un coût computationnel moindre pour une gamme plus large de variables globales [5] [6].

Pour augmenter l'utilité de l'inférence dans le guidage de la recherche, Pesant [7] a introduit la CPBP (*Constraint Programming with Belief Propagation*), une extension de la propagation de support analogue à l'inférence probabiliste, basée sur le comptage de modèles et la propagation de conviction (ou BP pour *Belief Propagation*) utilisant l'algorithme de passage de messages Sum-Product. L'approche est implémentée dans le solveur MiniCPBP [7]. Elle permet de calculer la conviction associée à chaque affectation variable/valeur, à savoir la proportion de solutions contenant cette affectation. L'information plus riche permet des heuristiques de branchement qui améliorent significativement les performances de résolution pour les CSPs [8].

Dans ce mémoire sous cherchons à déterminer comment ce type d'inférence avancée peut être appliqué aux COPs ?

1.2 Objectifs de recherche

Dans ce mémoire, nous évaluons d'abord les performances de l'inférence basée sur l'algorithme Sum-Product pour les problèmes d'optimisation sous contraintes.

Nous introduisons également un autre algorithme de passage de messages, Max-Product. Cet algorithme ayant été historiquement utilisé pour l'inférence probabiliste, nous l'avons adapté à l'approche CPBP et implémenté au sein du solveur MiniCPBP. Nous apportons dans ce contexte les contributions suivantes :

- Nous expliquons pourquoi cet algorithme est plus facilement applicable aux problèmes d'optimisation sous contraintes qu'aux problèmes de satisfaction de contraintes.
- Nous décrivons plus précisément quelle information sur les solutions d'un problème le Max-Product peut fournir avant de recourir à la recherche dans le cadre de la CP.
- Nous proposons des algorithmes adaptés au calcul de message pour Max-Product pour plusieurs contraintes globales.

— Nous dérivons diverses heuristiques de branchement exploitant les résultats des algorithmes de passage de messages, et évaluons leur utilité sur une gamme de problèmes d'optimisation.
— Nous étudions l'impact de différents paramètres et optimisations du passage de messages sur les performances de résolution.
— Nous montrons que la BP avec l'algorithme Max-Product ou Sum-Product, permet de compléter l'heuristique de sélection de variable *DomWDeg* d'une sélection de valeur *Max Marginal* et de guider efficacement la recherche vers des solutions avec de meilleurs scores.

1.3 Plan du mémoire

Ce mémoire comprend 7 chapitres, cette introduction incluse. Le Chapitre 2 introduit les concepts préliminaires nécessaires à la compréhension du mémoire : une définition formelle des COPs et des descriptions détaillées des approches de CP et CPBP. La revue de littérature, présentée au Chapitre 3, offre un tour d'horizon des méthodes d'inférence avancée utilisées dans le contexte de la résolution de CSPs et COPs. Le Chapitre 4 illustre notre application des algorithmes de BP à la résolution de COPs. Celle-ci repose sur les algorithmes Sum-Product et Max-Product, associés à une contrainte Oracle. Le Chapitre 5 présente les algorithmes développés pour permettre l'usage de la BP avec Max-Product. Le Chapitre 6 rassemble les résultats expérimentaux montrant l'efficacité d'heuristiques de recherche s'appuyant sur la BP pour la résolution d'exemplaires issus du corpus XCSP. Nous concluons dans le Chapitre 7, qui résume l'ensemble de nos contributions. Elle décrit les limites de notre approche, ainsi que des perspectives d'amélioration.

CHAPITRE 2 CONCEPTS PRÉLIMINAIRES

Ce chapitre présente les concepts et notations nécessaires pour la compréhension du mémoire.

Il introduit un formalisme pour les problèmes de satisfaction (CSP) et d'optimisation sous contraintes (COP), et présente des mécanismes liés à leur résolution tels que la programmation par contraintes (CP), le passage de messages et la propagation de conviction (BP).

2.1 Problèmes Combinatoires

2.1.1 Problèmes de satisfaction de contraintes

Définition

Les problèmes de satisfaction de contraintes (ou CSP pour *Constraint Satisfaction Problems*) sont des problèmes consistant à produire un état satisfaisant pour un ensemble de contraintes donné. Un CSP est défini formellement par un triplet $\langle X, D, C \rangle$, décrivant :

- X un ensemble fini des variables. Les variables de X sont notées par les symboles x, y, éventuellement indicées.
- le domaine D, l'ensemble fini des valeurs entières que peuvent prendre les variables. Au cours de la résolution, on établit que chaque variable $x \in X$ ne prend ses valeurs que dans un sous-ensemble de D désigné par $D(x)$.
- C un ensemble de contraintes, chaque contrainte c étant un ensemble de n-uplets $t \in c$ d'assignations autorisées pour un sous ensemble de variables $S(c) \subset X$. $S(c)$ est appelé la portée (ou *scope*) de c.

Pour une variable x on définit le voisinage $N(x)$ de la variable, comme l'ensemble des contraintes dans lesquelles elle est impliquée.

$$N(x) = \{c \in C, x \in S(c)\}$$

On appelle degré d'une variable la taille de son voisinage : $deg(x) = |N(x)|$

Une valeur v est assignée à une variable x quand $D(x) = \{v\}$. On dit que l'on assigne $x = v$ lorsque l'on fait l'hypothèse $D(x) = \{v\}$, ce qui est un aspect crucial de la recherche décrite dans la Section 2.2.3.

Une solution d'un CSP est une affectation complète $\sigma : X \to D$ telle que toutes les contraintes

de C soient satisfaites.

Exemple : Carré Latin

Un carré latin est un puzzle évoquant une version simplifiée du sudoku : une grille de taille $n \times n$ où, dans chaque ligne et chaque colonne, chaque nombre entre 1 et n est présent une unique fois.

Un exemplaire du problème spécifie la valeur d'un certain nombre de cellules de la grille, tandis que d'autres cellules sont vides. Le processus de résolution consiste à remplir les cellules restantes en respectant la contrainte de chaque ligne et de chaque colonne.

Un exemplaire peut admettre plusieurs solutions. Plus le nombre de cellules vides est grand, plus le nombre de solutions est potentiellement élevé.

Formellement, un exemplaire de carré latin d'ordre n est défini comme suit :

- $X = \{x_{i,j} \mid 1 \leq i, j \leq n\}$ l'ensemble des variables représentant les cellules de la grille, où $x_{i,j}$ désigne la cellule à la ligne i et colonne j.
- $D = \{1, 2, \ldots, n\}$ le domaine commun à toutes les variables, représentant les valeurs possibles dans chaque cellule.
- C l'ensemble des contraintes comprenant :
  - **Contraintes de ligne** : pour chaque ligne $i \in \{1, \ldots, n\}$,

    $$\text{AllDifferent}(x_{i,1}, x_{i,2}, \ldots, x_{i,n})$$

    exprimant que toutes les variables de la ligne i doivent prendre des valeurs distinctes, i.e $x_{i,j} \neq x_{i,k}$ pour tout $j \neq k$.
  - **Contraintes de colonne** : pour chaque colonne $j \in \{1, \ldots, n\}$,

    $$\text{AllDifferent}(x_{1,j}, x_{2,j}, \ldots, x_{n,j})$$

    exprimant que toutes les variables de la colonne j doivent prendre des valeurs distinctes.
  - **Contraintes d'assignation** : pour chaque cellule (i, j) dont la valeur v est donnée dans l'exemplaire,

    $$x_{i,j} = v$$

    fixant la valeur de la variable correspondante.

Un exemple d'exemplaire et deux solutions possibles pour cet exemplaire sont représentés en

Figure 2.1.

2.1.2 Problèmes d'optimisation combinatoire

Les problèmes d'optimisation combinatoire (ou COP pour *Constraint Optimisation Problems*) introduisent une hiérarchie entre les solutions d'un CSP selon leur désirabilité ou leur coût, via une fonction objectif f qui associe les solutions à un score. On cherchera ensuite à maximiser ou minimiser ce score.

Les carrés latins comme COP

On peut transformer le problème du carré latin d'un problème de satisfaction en un problème d'optimisation en ajoutant une fonction objectif f qui associe à chaque solution un score.

Par exemple, lors d'une étude initiale de la résolution de COPs, on cherchera à maximiser la somme des valeurs d'un sous-ensemble de cellules formant un carré dans le coin supérieur gauche du carré latin.

Pour un carré latin de taille n et un sous-ensemble rectangulaire de dimension $k \times l$ avec $k < n$ et $l < n$:

$$f(x) = \sum_{i=1}^{k} \sum_{j=1}^{l} x_{i,j}$$

Par exemple, pour $k = 3$ et $l = 2$, les deux solutions présentées en Figure 2.1 ont respectivement un score de 20 et 17.

Le choix de sommer sur une zone rectangulaire plutôt qu'un sous-ensemble arbitraire de cases introduit des dépendances entre les cases participant à l'objectif et rendent l'optimisation plus difficile.

2.2 Programmation par contraintes (CP)

La programmation par contraintes (ou CP pour *Constraint Programming*) est une méthode de résolution de problèmes combinatoires.

2.2.1 Langage de modélisation

La programmation par contraintes introduit des contraintes globales, dont la portée peut être arbitrairement grande, et qui décrivent succinctement des relations complexes entre les variables.

				1
			5	
5				
	2			
		5		

3	5	2	4	**1**
2	1	3	**5**	4
5	4	1	2	3
1	**2**	4	3	5
4	3	**5**	1	2

2	5	3	4	**1**
1	3	2	**5**	4
5	1	4	2	3
4	**2**	1	3	5
3	4	**5**	1	2

FIGURE 2.1 Exemplaire de carré latin et deux solutions possibles

Voici par exemple les contraintes globales abordées dans ce mémoire :

— AllDifferent$(x_1, \ldots, x_n)$: toutes les variables de la portée prennent des valeurs différentes.

— Table$((x_1, \ldots, x_n), R)$, (aussi nommée contrainte Extension) : les tuples acceptés sont listés de manière explicite dans la relation R.

— Sum$((x_1, \ldots, x_n), (\sigma, \alpha_1, \ldots, \alpha_n))$: pour une portée de variables $(x_1, \ldots, x_n)$, et des scalaires $(\sigma, \alpha_1, \ldots, \alpha_n)$, la contrainte est vérifiée pour les valeurs telles que :

$$\sum_{i=1}^{n} \alpha_i x_i = \sigma$$

— Regular$((x_1, \ldots, x_n), \mathcal{A})$: une contrainte de type Regular prend comme paramètre un automate déterministe $\mathcal{A}$. Sa portée est constituée une séquence de variables $(x_1, \ldots, x_n)$ dont les valeurs doivent former un mot reconnu par l'automate.

2.2.2 Algorithmes de filtrage

À chaque contrainte globale est associée un ou plusieurs algorithmes de filtrage : à partir des domaines des variables de la portée de la contrainte, ces algorithmes déterminent que certains couples valeurs/variables ne figurent dans aucun des tuples acceptés. De telles valeurs sont alors éliminées du domaine de la variable : c'est le filtrage.

Si au contraire, un tel tuple existe, on dit qu'il est un support de la valeur en question. Une valeur ne doit pas être filtrée si elle possède un support.

Si un algorithme de filtrage peut éliminer toutes les valeurs ne possédant pas de support, on dit qu'il atteint la **cohérence d'arc généralisée**.

Les prochaines sous-sections décrivent les algorithmes de filtrage pour les contraintes globales décrites précédemment.

AllDifferent

Une contrainte AllDifferent est satisfaite si toutes les variables de sa portée prennent des valeurs différentes.

Considérons un graphe biparti $G = (V_1, V_2, E)$ où les sommets de V_1 correspondent à chaque variable de la portée et les sommets de V_2 correspondent à l'ensemble des valeurs possibles pour ces variables. E est l'ensemble des arêtes du graphe, tel que pour une variable x et ses valeurs possibles $v \in D(x)$, on a $(x, v) \in E$. Satisfaire la contrainte revient à trouver un couplage couvrant V_1 dans G, chaque arête (x, v) du couplage indiquant les assignations variables/valeurs retenues.

Par conséquent, une assignation (x, v) possède un support si et seulement si il existe un tel couplage dans G qui contient (x, v).

Le théorème suivant s'avère utile :

Théoreme 1 (Théorème de Berge) *Une arête appartient à un couplage si et seulement si pour un couplage maximal M donné, l'arête appartient à l'un des ensembles suivants :*

- *au couplage M*
- *à un chemin alternant de longueur paire*
- *à un cycle alternant de longueur paire*

L'algorithme de filtrage [9] consiste à construire un couplage maximal M, l'ensemble P des arêtes faisant partie des chemins alternants de longueur paire à partir des valeurs non assignées par M, ainsi que l'ensemble C des arêtes faisant partie des cycles alternants de longueur paire. On filtre ensuite les assignations ne figurant pas dans M, P ou C.

Table

Dans une contrainte Table, les tuples acceptés sont listés explicitement et constituent les supports.

L'algorithme *Compact Table* (CT) [10] utilise une structure de données appelée Reversible Sparse Bit-Sets (RSBS) pour stocker efficacement les supports. Un tableau contient les tuples initialement autorisés par la contrainte, tandis qu'un RSBS stocke les indices du sous-ensemble de tuples valides dont les valeurs appartiennent aux domaines des variables. Cette structure résultante est appelée `currentTable`.

Pour chaque variable x et valeur a de son domaine, un mot binaire `supports[x,a]` a son i-ème bit positionné à 1 si et seulement si le i-ème tuple constitue un support pour (x, a). Le

tableau `residues[x,a]` indexe la position où un support a été trouvé en dernier pour (x, a), permettant d'optimiser les recherches ultérieures.

La `currentTable` est mise à jour de manière incrémentale ou reconstruite entièrement lorsque le domaine d'une variable change, selon le nombre de valeurs supprimées.

D'après la définition de `currentTable` et `supports`, une valeur (x, a) possède un support si et seulement si `currentTable` $\cap$ `supports[x,a]` $\neq \emptyset$.

Lorsqu'un tuple est retiré de `currentTable`, cette condition est vérifiée pour chaque assignation (x, a) du tuple, et a est filtrée si elle n'est pas satisfaite.

Lors de la vérification de la condition de support, `residues[x,a]` est consulté avant d'itérer sur l'ensemble du mot `supports[x,a]`, réduisant ainsi le temps de calcul.

Sum

Pour une portée $(x_1, \ldots, x_n)$, et des scalaires $(\sigma, \alpha_1, \ldots, \alpha_n)$, on cherche les valeurs telles que :

$$\sum_{i=1}^{n} \alpha_i x_i = \sigma$$

Trick [6] propose un algorithme de filtrage basé sur la programmation dynamique. On construit lors d'une passe en avant un graphe représentant les sommes partielles $\sum_{i=1}^{k} \alpha_i x_i$, pour $k \leq n$. On peut filtrer chaque valeur de $D(x_s)$ telles que σ n'est atteinte par aucun chemin dans ce graphe.

Ensuite, pour chaque valeur de $D(x_s)$ restante, on effectue une passe en arrière : chaque chemin atteignant une valeur de σx_s constitue un support pour toutes les assignations qui le composent.

Les valeurs qui ne figurent dans aucun chemin peuvent être filtrées des domaines de $x_1, \ldots, x_n$.

Regular

Un automate fini déterministe est un quintuplet $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$ où Σ est un alphabet, Q un ensemble fini d'états, δ une fonction de transition de $Q \times \Sigma$ dans Q, q_0 l'état initial et F l'ensemble des états acceptants. Un mot est reconnu si la séquence de transitions depuis q_0 selon ses lettres mène à un état de F.

L'algorithme de filtrage de la contrainte Regular [5] détermine si des assignations n'apparaissent dans aucun mot reconnu par l'automate associé.

De manière similaire à Sum, on construit lors d'une passe en avant un graphe représentant les états d'arrivée pour tous les préfixes $x_1, \ldots, x_k$ pour $k \leq n$.

Puis lors d'une passe en arrière à partir des états acceptants, on marque chaque arc faisant partie d'un chemin comme possédant un support. Les arcs qui ne font pas partie d'aucun chemin peuvent être filtrés.

2.2.3 Recherche à retour arrière

Les algorithmes de filtrage de chaque contrainte sont exécutés plusieurs fois jusqu'à ce qu'un point fixe soit atteint (voir Algorithme 1). Une fois les limites de l'inférence atteintes, la CP s'appuie sur une recherche à retour arrière pour poursuivre la résolution.

Principe

La recherche consiste à sélectionner une variable x et à lui assigner une valeur v de son domaine, éliminant toutes les autres valeurs et posant l'hypothèse $D(x) = \{v\}$. Cette réduction drastique du domaine de la variable sélectionnée peut permettre de reprendre le filtrage, jusqu'à atteindre de nouveau un point fixe. On poursuit la recherche en assignant une valeur à une autre variable, ce qui permet plus de filtrage... On alterne ainsi entre inférence et recherche tout le long du processus de résolution.

Si on a assigné une valeur à chaque variable (explicitement lors de la recherche, ou parce que la taille de leur domaine est de 1), l'assignation complète est une solution.

Si lors du filtrage, le domaine d'une variable devient vide, on se trouve dans une situation d'insatisfaisabilité, on dit qu'on rencontre un **conflit** ou **échec**. On revient alors sur certaines décisions de la recherche (c'est le retour arrière) et on poursuit la recherche en prenant de nouvelles décisions.

L'ensemble des décisions prises et des retours arrières définissent un arbre de recherche, qui finit par explorer l'intégralité des combinaisons possibles. Le filtrage joue un rôle crucial pour rendre cette recherche efficace, car il permet l'élagage de nombreuses branches de l'arbre.

Heuristiques de branchement

La variable à fixer ainsi que la valeur assignée suit une **heuristique**. On peut en général décomposer l'heuristique en un critère de sélection de variable, ainsi qu'un critère de sélection de valeur pour cette variable. La qualité de l'heuristique choisie peut grandement impacter la performance de la résolution.

Algorithm 1: Filtrage jusqu'à point fixe

Data: Ensemble de contraintes $\mathcal{C}$, ensemble de variables $\mathcal{X}$
Result: Domaines des variables filtrés jusqu'au point fixe
repeat
  CHANGE $\leftarrow$ false;
  foreach *contrainte* $c \in \mathcal{C}$ **do**
    CHANGE $\leftarrow$ Filtrer(c) $\vee$ CHANGE;
  end
until $\neg$*CHANGE*;

Il s'agit souvent de choisir une variable dont l'assignation va induire un maximum de filtrage, au risque de causer un conflit (approche *fail first*). On privilégie pour cela les variables ayant des petits domaines ou participant à un grand nombre de contraintes (un haut degré). L'heuristique de sélection de variable *DomDeg* marie ces deux considérations, en sélectionnant la variable ayant le plus petit ratio de taille de domaine sur son degré.

Il existe des heuristiques possédant un aspect d'apprentissage : *DomWDeg* (*domain over weighted degree*) [11] intègre un poids qui reflète le nombre de fois où la variable a été impliquée dans des conflits et sélectionne les variables qui l'ont souvent été. On définit :

$$domwdeg(x) = \frac{|D(x)|}{w(x).deg(x)}$$

Pour la sélection de valeur, on cherche à sélectionner une valeur appartenant à une solution, pour éviter un retour arrière (approche *succeed first*). Cette approche est en tension avec le *fail first* de la sélection de variable et la question de la sélection de valeur, pour *DomWDeg* par exemple, reste ouverte. En l'absence d'information sur la participation des valeurs du domaine à des solutions, *MinValue* est un exemple simple qui sélectionne la valeur la plus basse du domaine de la variable préalablement sélectionnée.

On désignera parfois la variable sélectionnée par x_s et la valeur sélectionnée par v_s. Par exemple, on décrira la sélection de variable *DomDeg* et la sélection de valeur *Min Value* :

$$x_s = \underset{x \in X}{\operatorname{argmin}}\, domwdeg(x)$$

$$v_s = \underset{v \in D(x_s)}{\operatorname{argmin}}\, v$$

Les méthodes d'inférence avancées comme la propagation de conviction (ou BP, voir Sec-

tion 2.3.1) sont utilisées afin d'obtenir plus d'informations que n'en fournirait la seule cohérence d'arc généralisée, et de servir de base à des heuristiques de branchement plus sophistiquées et efficaces.

Types de recherche

Deux types de recherche sont considérés dans ces travaux.

La recherche en profondeur (ou DFS pour *Depth First Search*) va revenir, lorsqu'un conflit est détecté, sur la dernière décision prise et emprunter la branche complémentaire : si l'on avait assigné $x = v$, on va alors retirer v de $D(x)$ pour garantir $x \neq v$ dans la nouvelle branche de l'arbre de recherche. On effectue ensuite une nouvelle assignation suivant l'heuristique de branchement. Par conséquent, on revient en priorité sur les décisions les plus récentes, et rarement sur les choix faits au début de la recherche.

La recherche à divergence limitée (ou LDS pour *Limited Discrepancy Search*), introduit la notion de divergence, désignant le nombre de décisions prises qui ne respectent pas l'heuristique. Autrement dit, la divergence est le nombre de fois où la recherche a emprunté la branche complémentaire $x \neq d$ d'une décision $x = d$ prise par l'heuristique. Du début de la résolution jusqu'au premier conflit, la recherche suit exactement l'heuristique de branchement et la divergence est nulle. Au premier conflit, la recherche va annuler une décision $x = v$ et emprunter la branche complémentaire $x \neq v$. La divergence est alors égale à 1. Au second conflit, on va rétablir cette première décision $x = v$ et aller contre une autre décision $x' = v'$ en imposant $x' \neq v'$. On reste ainsi à une divergence de 1. La divergence augmente à nouveau lorsque l'on a annulé une fois chacune des décisions originales. On choisit alors deux décisions à annuler (par exemple, on impose $x \neq v$ *et* $x' \neq v'$) et on atteint une divergence de 2.

La recherche à divergence limitée cherche à s'appuyer au maximum sur les décision de l'heuristique de branchement, mais permet d'annuler les décisions prises en début de recherche.

Mécanismes d'amélioration

Plusieurs mécanismes ont été développés pour améliorer l'efficacité de la recherche. L'apprentissage des *no-goods* enregistre et explique les décisions qui mènent à l'échec pour éviter de répéter les mêmes erreurs lors de l'exploration d'autres branches [3]. Le *backjumping* permet, lorsqu'une incohérence est détectée, de revenir plus loin dans l'arbre de recherche que le simple retour arrière chronologique, en identifiant la cause réelle du conflit pour une exploration plus efficace [4]. Les *relances* suppriment périodiquement l'arbre de recherche, dans un effort de diversifier l'exploration de l'espace de recherche en repartant avec de nouvelles

heuristiques ou un ordre différent [2].

2.2.4 Propagation des contraintes

En pratique, déclencher le filtrage à un moment opportun en économisant du temps de calcul repose sur l'approche de propagation des contraintes.

Lorsqu'une valeur est retirée du domaine d'une variable, cette dernière envoie l'information que son domaine a changé aux contraintes qui la concernent.

À la réception de ce message, une contrainte peut vérifier incrémentalement si la réduction du domaine prive des valeurs d'autres variables de leur support. Le cas échéant, elle envoie un message aux variables les informant que leur domaine doit être réduit.

Les variables recevant ce type de message préviennent alors les contraintes environnantes, et ainsi de suite jusqu'à atteindre un point fixe.

2.2.5 Exemple : Résolution du Carré Latin

Nous reprenons l'exemplaire présenté à la Figure 2.1 et montrons quelques étapes de résolution en Figure 2.2.

Si l'on effectue le filtrage, le domaine initial $\{1, 2, 3, 4, 5\}$ est réduit pour chacune des cellules. Pour deux d'entre elles, il est réduit à l'unique valeur de 5, ce qui permet de fixer les variables correspondantes. L'état de la résolution une fois que le filtrage a atteint son point fixe est représenté en Figure 2.2, dans le carré de gauche.

Considérons les cases de la première ligne. Au point fixe, les domaines des trois cases vides sont $\{2, 3, 4\}$. Si l'on assigne à la case de gauche la valeur 3, et qu'on effectue du filtrage jusqu'à un nouveau point fixe (Figure 2.2, centre), les domaines des deux autres cases sont réduits à $\{2, 4\}$. On assigne à présent la valeur 2 à la case suivante, le filtrage réduit le domaine de la dernière case restante à une unique valeur et on peut la fixer à 4 (Figure 2.2, droite).

2.3 Passage de messages

Le passage de messages (*message passing*) est une méthode d'inférence en modèles graphiques, applicables aux réseaux de contraintes et basée sur l'échange de messages entre les variables et les contraintes.

Le passage de messages peut être considéré comme une extension de la propagation des

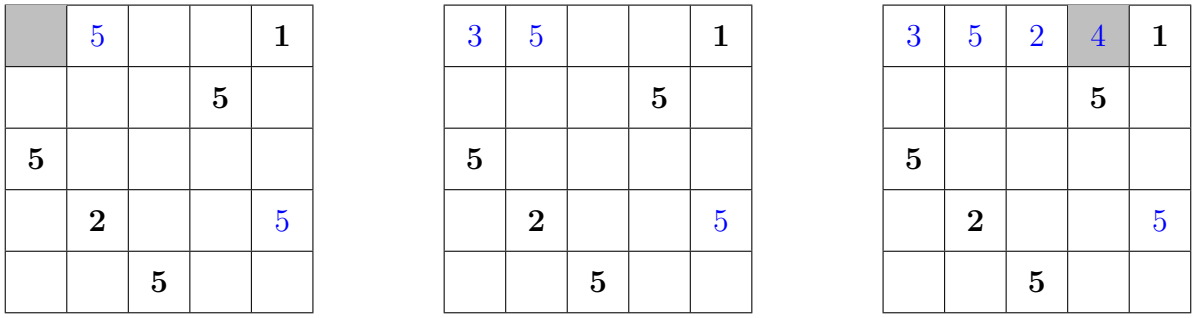

FIGURE 2.2
(gauche) Au point fixe, le filtrage a permis d'inférer les valeurs de deux cellules. Le domaine de la case (1, 1) de la grille (en gris) est $\{2, 3, 4\}$.
(centre) La recherche assigne une valeur et poursuit la résolution après que le point fixe ait été atteint.
(droite) Le filtrage se poursuit à chaque nœud de la recherche. Il réduit le domaine de la case (1, 4) (en gris) à 4, ce qui permet de fixer sa valeur.

contraintes, où les messages contiennent non seulement l'information des valeurs présentes dans le domaine, mais un scalaire pour chacune des valeurs qui composent le domaine [7]. Dans un souci de concision, on désignera parfois l'échange de messages par **propagation**.

Le principe du *message passing* repose sur l'échange bidirectionnel d'informations en variables et contraintes : les variables envoient leurs distributions marginales courantes aux contraintes, qui calculent en retour des informations locales basées sur leurs ensembles de solutions.

Une variable accumule pour chaque valeur de son domaine le produit (non normalisé) des messages qu'elle a reçu des contraintes auxquelles elle participe :

$$\hat{\Theta}_x(v) = \prod_{c \in N(x)} \mu_{c \to x}(v) \tag{2.1}$$

Une variable x envoie à une contrainte c le produit des messages reçus des autres contraintes, excluant c elle-même :

$$\mu_{x \to c}(v) = \prod_{c' \in N(x) \setminus c} \mu_{c' \to x}(v) \tag{2.2}$$

Une contrainte c calcule une information locale sur les tuples qu'elle accepte, puis envoie à chaque variables x un message $\mu_{c \to x}(v)$ reflétant cette connaissance.

Dans la suite on appellera **une itération** de la propagation de messages l'exécution de l'ensemble des actions suivantes :

— l'envoi de tous les messages par les variables (dont les valeurs correspondent aux distributions marginales à l'itération précédente)

— l'envoi de tous les messages des contraintes

— l'accumulation (i.e. le produit) des messages des contraintes au niveau des variables.

Dans cette section, nous décrivons deux algorithmes de passage de messages, qui diffèrent dans le calcul des messages des contraintes aux variables : Sum-Product et Max-Product.

Un abus de langage est parfois fait dans le mémoire : les termes de conviction (*belief*), ou distribution marginale sont fortement associés, voire réservés à Sum-Product. Cependant, la propagation Max-Product a été implémentée en réutilisant des mécanismes servant alors uniquement à la BP. On désignera donc occasionnellement les messages Max-Product comme des convictions, et les grandeurs accumulées par les variables comme des distributions marginales.

Nous introduisons un exemple simple et soulignons comment le passage de messages fournit des informations sur les solutions d'un problème avant de recourir à la recherche.

2.3.1 Sum-Product ou propagation de conviction (BP)

L'algorithme Sum-Product, aussi désigné par propagation de conviction (ou BP pour *Belief Propagation*), calcule une approximation de la distribution des solutions d'un problème de contraintes [7], sans recourir à la recherche. Il calcule itérativement les marginales de chaque variable dans un graphe de contraintes, en propageant et accumulant les messages entre les nœuds variables et les nœuds contraintes.

Les messages des contraintes vers les variables correspondent à un dénombrement pondéré des tuples acceptés contenant une assignation. Le dénombrement est pondéré par la valeur des messages reçus des variables pour chaque assignation. Les messages sont calculés par :

$$\mu_{c\to x}(v) = \sum_{\mathbf{x}_c : x=v} \left(f_c(\mathbf{x}_c) \prod_{x^* \in S(c)\setminus\{x\}} \mu_{x^*\to c}(v) \right) \tag{2.3}$$

où $N(x)$ l'ensemble des contraintes c où $x \in S(c)$ et f_c est la fonction associée à c telle que $f(\mathbf{x_c}) = 1$ si $\mathbf{x_c}$ satisfait la contrainte c et 0 sinon. L'algorithme Sum-Product tire son nom du calcul des messages émis par les contraintes décrit dans l'Équation 2.3.

La variable x accumule une marginale (non normalisée) comme le produit des messages qu'elle a reçus des contraintes environnantes.

$$\hat{\Theta}_x(v) = \prod_{c\in N(x)} \mu_{c\to x}(v) \tag{2.4}$$

On normalise les marginales d'une variable sur l'ensemble des valeurs de son domaine :

$$\hat{\theta}_x(v) = \frac{\hat{\Theta}_x(v)}{\prod_{v' \in D(x)} \hat{\Theta}_x(v)} \tag{2.5}$$

La valeur marginale normalisée $\hat{\theta}_x(v)$ approxime la proportion de solutions contenant l'affectation $x = v$, i.e. la marginale réelle notée $\theta_x(v)$:

$$\theta_x(v) = \frac{\sum_{\mathbf{x}:x=v} \prod_{c \in C} f_c(\mathbf{x})}{\sum_{\mathbf{x}} \prod_{c \in C} f_c(\mathbf{x})} \tag{2.6}$$

Un exemple

Reprenons un CSP basique précédemment introduit par Pesant [8] :

- **Variables** : a, b, c, d
- **Domaines** : $D_a = D_b = D_c = D_d = \{1, 2, 3, 4\}$
- **Contraintes** :
  - AllDifferent(a, b, c)
  - $a + b + c + d = 7$
  - $c \leq d$

Il y a exactement deux solutions réalisables :

- $a = 2, b = 3, c = 1, d = 1$
- $a = 3, b = 2, c = 1, d = 1$

La cohérence de domaine sur chaque contrainte ne fournit aucun filtrage, les domaines restent donc inchangés. Les marginales exactes sont simples à calculer puisque les solutions sont connues (voir Tableau 2.2).

On applique la BP : chaque variable possède initialement une distribution uniforme sur son domaine, comme illustré dans le Tableau 2.1. Après 10 itérations de l'algorithme Sum-Product, les marginales des variables, présentées dans le Tableau 2.3, sont proches des marginales réelles.

L'algorithme Sum-Product a fait ressortir des informations importantes sur la distribution des solutions, au-delà de la cohérence de domaine, qui peuvent être exploitées pour la recherche [12].

Pour chaque paire variable/valeur, un message Sum-Product nécessite d'effectuer un comptage pondéré des tuples acceptés par la contrainte. La pondération correspond aux messages envoyés par les variables aux contraintes pour chaque assignation. Un algorithme dédié est utilisé pour chaque type de contrainte.

	1	2	3	4
a	0.25	0.25	0.25	0.25
b	0.25	0.25	0.25	0.25
c	0.25	0.25	0.25	0.25
d	0.25	0.25	0.25	0.25

TABLEAU 2.1 Initialement chaque variable possède une distribution uniforme

	1	2	3	4
a	0	0.5	0.5	0
b	0	0.5	0.5	0
c	1	0	0	0
d	1	0	0	0

TABLEAU 2.2 Marginales exactes pour l'Exemple 2.3.1

AllDifferent

Comme vu en Section 2.2.2, un tuple accepté par une contrainte AllDifferent est un couplage dans un graphe biparti correspondant. Le calcul d'un message de la contrainte AllDifferent vers les variables de sa portée consiste en un comptage pondéré des couplages. Ceci est équivalent au calcul du permanent de la matrice d'adjacence du graphe biparti [7]. Ce problème se trouvant dans la classe de complexité $\#P$, on a en pratique recours à une borne supérieure proposée par Soules [13], pour un calcul plus rapide des messages.

Table

Le calcul de message Sum-Product est décrit par l'algorithme 2.

Les structures de données utilisées pour l'algorithme Compact Tables sont augmentées d'un tableau `tupleWeight`, qui associe un poids à chaque tuple t dans l'ensemble T des tuples acceptés. Le poids de chaque tuple est le produit des poids des assignations qui le composent. Le poids de chaque assignation correspond au message correspondant reçu d'une variable :

$$w(t) = \prod_{(x,v)\in t} \mu_{x\to c}(v)$$

	1	2	3	4
a	0.01	0.52	0.46	0.01
b	0.01	0.52	0.46	0.01
c	0.98	0.01	0.005	0.005
d	0.897	0.09	0.003	0.00003

TABLEAU 2.3 Conviction après 10 itérations avec Sum-Product

Pour chaque couple variable/valeur (x, v), on récupère l'ensemble des tuples qui supportent cette assignation. Pour chaque tuple t supportant (x, v), on divise le poids du tuple par le poids de l'assignation $\mu_{x\to c}(v)$. On obtient un terme correspondant au poids d'une solution :

$$\frac{w(t)}{\mu_{x\to c}(v)} = \prod_{\substack{(x',v')\in t \\ x'\neq x}} \mu_{x'\to c}(v')$$

On accumule ces termes pour chaque couple variable/valeur en les sommant sur l'ensemble des tuples acceptés et on obtient :

$$\forall (x, v), \quad \mu_{c\to x}(v) = \sum_{t\in T:(x,v)\in t} \prod_{\substack{(x',v')\in t \\ x'\neq x}} \mu_{x'\to c}(v')$$

Sum

Pesant [7] présente un dénombrement pondéré pour la contrainte Sum basé sur la programmation dynamique, décrit par l'algorithme 3.

Dans le graphe de programmation dynamique contenant les valeurs des sommes partielles que l'on utilise pour le filtrage, chaque arête représentant une assignation se voit assigner le poids correspondant. La passe en avant permet de construire le graphe des sommes partielles. À chaque extrémité d'une arête, on somme le poids des chemins entrants et sortants, ce qui permet de calculer le poids des chemins traversant une arête lors de la passe en arrière, en excluant celle-ci conformément à la formule de calcul des messages de propagation.

Algorithm 2: Mise à jour Sum-Product pour contrainte Table

Data: Variables $x_1, \ldots, x_n$, table de contrainte T, ensembles de support $supports[i][v]$
Result: Convictions locales mises à jour $\mu_{c \to x_i}(v)$

```
// Calculer les tuples supportés
```
$supportedTuples \leftarrow \emptyset$;
for $i \leftarrow 1$ **to** n **do**
  $support_i \leftarrow \bigcup_{v \in D(x_i)} supports[i][v]$;
  $supportedTuples \leftarrow supportedTuples \cap support_i$;
end
```
// Calculer les poids des tuples
```
for $t \in supportedTuples$ **do**
  $tupleWeight[t] \leftarrow \prod_{i=1}^{n} \mu_{x_i \to c}(t[i])$;
end
```
// Mettre à jour les convictions pour chaque couple variable/valeur
```
for $i \leftarrow 1$ **to** n **do**
  for $v \in D(x_i)$ **do**
    $conviction \leftarrow 0$;
    $outsideConviction \leftarrow \mu_{x_i \to c}(v)$;
    for $t \in supports[i][v] \cap supportedTuples$ **do**
      `// On accumule une somme`
      $conviction \leftarrow conviction + \frac{tupleWeight[t]}{outsideConviction}$;
    end
    $\mu_{c \to x_i}(v) \leftarrow conviction$;
  end
end

Regular

Une méthode analogue au comptage pondéré de la contrainte Sum est utilisée dans le graphe des préfixes décrit pour l'algorithme de filtrage. Elle introduit des valeurs pour les chemins entrants et sortants, afin de calculer le poids des chemins traversant une arête et le message correspondant.

2.3.2 Max-Product

Dans un contexte d'inférence probabiliste, l'algorithme Max-Product est un algorithme de passage de messages présentant une alternative à l'algorithme Sum-Product. Il est typiquement utilisé pour identifier la valeur la plus probable pour l'ensemble des variables [14]. Son message variable-vers-contrainte est le même que (2.2) mais son message contrainte-vers-

Algorithm 3: Dénombrement pondéré la contrainte Sum (adapté de [7])

Data: Variables $x_1, \ldots, x_n$, bornes des domaines j_{min}, j_{max}, messages reçus $outsideBelief[x_i][v]$
Result: Convictions locale mises à jour $localBelief[x_i][v]$
Initialize π_{in}, π_{out} to 0;
$\pi_{in}[1][0] \leftarrow 1$; $\pi_{out}[n][0] \leftarrow 1$;
// Passe en avant
for $i \leftarrow 1$ **to** $n-1$ **do**
  foreach $v \in D(x_i)$, $j \in [j_{min} - \min(0,v), j_{max} - \max(0,v)]$ **do**
    if $\pi_{in}[i][j] > 0$ **then**
      $\pi_{in}[i+1][j+v] \leftarrow \pi_{in}[i+1][j+v] + \pi_{in}[i][j] \times outsideBelief[x_i][v]$;
    end
  end
end
// Passe en arrière et mise à jour des convictions locales
for $i \leftarrow n$ **to** 2 **do**
  foreach $v \in D(x_i)$ **do**
    $localBelief[x_i][v] \leftarrow 0$;
    for $j \leftarrow j_{min} - \min(0,v)$ **to** $j_{max} - \max(0,v)$ **do**
      if $\pi_{out}[i][j+v] > 0$ **then**
        $\pi_{out}[i-1][j] \leftarrow \pi_{out}[i-1][j] + \pi_{out}[i][j+v] \times outsideBelief[x_i][v]$;
        $localBelief[x_i][v] \leftarrow localBelief[x_i][v] + \pi_{in}[i][j] \times \pi_{out}[i][j+v]$;
      end
    end
  end
end
// Pour la première variable
$localBelief[x_1][v] \leftarrow \pi_{out}[1][v]$ pour tout $v \in D(x_1)$;

variable agrège en prenant le maximum au lieu de la somme :

$$\mu_{c \to x}(v) = \max_{\mathbf{x}_c : x = v} \left(f_c(\mathbf{x}_c) \prod_{x^* \in N(c) \setminus \{x\}} \mu_{x^* \to c}(v_{x*}) \right) \tag{2.7}$$

On utilise la même définition d'une **itération** de propagation décrite pour Sum-Product dans la Section 2.3.

Équivalence à la cohérence d'arc généralisée

Si les variables commencent avec des marginales uniformes, l'algorithme Max-Product est équivalent à la cohérence d'arc généralisée [15] : à la convergence, les valeurs possédant un

support ont toutes le même poids, tandis que les autres ont un poids nul. Dans l'Exemple 2.3.1 l'algorithme ne produit aucune information utile pour guider vers la satisfaction des contraintes, car les domaines sont déjà filtrés et les marginales restent uniformes (voir Tableau 2.4).

Si l'on représente les marginales par leur logarithme, dans le but par exemple d'éviter des problèmes de précision numérique, on peut sommer les marginales au lieu de prendre leur produit. On désignera alors l'algorithme de calcul des messages comme Max-Sum [16]. Cette représentation sera utile pour le calcul de message pour la contrainte AllDifferent, présenté en Section 5.4.

2.3.3 Limitations du passage de messages

Si le graphe de contraintes est un arbre, propager les messages des feuilles vers l'intérieur s'apparente à un algorithme de programmation dynamique et permettra d'obtenir des marginales exactes [14].

Ce n'est presque jamais le cas en pratique : les contraintes globales à fortes arités, les modèles complexes de problèmes réels... De nombreux facteurs introduisent des cycles dans les réseaux de contraintes que nous devons résoudre.

La présence de cycles cause deux difficultés principales [17] :

— On ne peut garantir que l'on va converger vers les distributions marginales exactes du problème.
— Il peut se produire, au lieu d'une convergence, des oscillations dans les marginales.

Une solution au second problème est d'introduire un coefficient d'amortissement (*damping*) $d < 1$ sur les messages envoyés. Les messages envoyés sont alors :

$$\mu^*_{x\to c}(v) = d\mu_{x\to c}(v) \tag{2.8}$$

$$\mu^*_{c\to x}(v) = d\mu_{c\to x}(v) \tag{2.9}$$

L'échange de messages amortis va permettre aux marginales de converger à nouveau. Ce mécanisme est intégré au solveur MiniCPBP et activé si une oscillation des marginales est détectée [7]. Puisqu'un amortissement trop fort ralentit la propagation, le choix d'un coefficient judicieux est également important.

	1	2	3	4
a	0.25	0.25	0.25	0.25
b	0.25	0.25	0.25	0.25
c	0.25	0.25	0.25	0.25
d	0.25	0.25	0.25	0.25

TABLEAU 2.4 Conviction après 10 itérations avec Max-Product : marginales uniformes

2.3.4 Entropie des variables

La disponibilité des distributions marginales estimées par le pasage de messages permet d'introduire le concept d'entropie de Shannon dans le contexte de la programmation par contraintes. L'entropie d'une variable x est définie par :

$$H(x) = -\sum_{v \in D(x)} \hat{\theta}_x(v) \log(\hat{\theta}_x(v)) \tag{2.10}$$

où $\hat{\theta}_x(v)$ représente la marginale estimée par la propagation pour l'assignation (x, v).

Cette quantité non négative peut être interprétée comme l'incertitude sur la valeur que devrait prendre la variable x dans une solution. Une entropie nulle correspond à $\hat{\theta}_x(v) = 1$ pour une certaine valeur v, indiquant avec certitude que x doit prendre cette valeur :

— Pour Sum-Product, cela signifie qu'elle a toujours cette valeur dans toutes les solutions.
— Pour Max-Product, cela signifie qu'elle a cette valeur dans la ou les solutions optimales.

L'entropie maximale $\log(|D(x)|)$ est atteinte lorsque les marginales sont uniformément distribuées sur le domaine de x.

L'entropie normalisée d'une variable divise son entropie par le logarithme de la cardinalité de son domaine :

$$H_{\text{norm}}(x) = \frac{H(x)}{\log(|D(x)|)} \tag{2.11}$$

Cette mesure, comprise entre 0 et 1, permet de comparer l'incertitude entre variables ayant des domaines de tailles différentes.

L'entropie constitue un critère naturel pour les heuristiques de branchement [18] : une variable avec une faible entropie présente une forte certitude sur sa valeur optimale, tandis qu'une

entropie élevée indique une grande incertitude. Cette information peut guider la sélection de variables lors de la recherche, en privilégiant par exemple les variables les plus certaines ou au contraire les plus incertaines selon la stratégie adoptée.

2.3.5 Heuristiques de branchement exploitant le passage de messages

Exploiter le passage de messages pour la résolution nécessite de nouvelles heuristiques de branchement basée sur les marginales obtenues. Diverses heuristiques ont été proposées :

— **Max Marginal** [12] assigne le couple variable/valeur présentant la plus haute marginale parmi toutes les distributions.

$$x_s = \underset{x \in X}{\operatorname{argmax}} \left(\max_{v \in D(x)} \hat{\theta}_x(v) \right)$$
$$v_s = \underset{v \in D(x_s)}{\operatorname{argmax}} \hat{\theta}_{x_s}(v)$$

— **Max Marginal Regret** : on définit le regret d'une distribution comme la différence entre la plus haute marginale et la seconde plus haute.

$$\hat{\theta}_{max}(x) = \max_{v \in D(x)} \hat{\theta}_x(v)$$
$$v_{max}(x) = \underset{v \in D(x)}{\operatorname{argmax}} \hat{\theta}_x(v)$$
$$\hat{\theta}_{2nd}(x) = \max_{v \in D(x) \setminus \{v_{\max}\}} \hat{\theta}_x(v)$$
$$regret(x) = \hat{\theta}_{max}(x) - \hat{\theta}_{2nd}(x)$$

L'heuristique *Max Marginal Regret* sélectionne la variable présentant le plus haut regret, et lui assigne sa valeur présentant la plus haute marginale.

$$x_s = \underset{x \in X}{\operatorname{argmax}}\, regret(x)$$
$$v_s = \underset{v \in D(x_s)}{\operatorname{argmax}} \hat{\theta}_{x_s}(v)$$

— **Max Marginal Strength** [7] : la force d'une marginale correspond à l'écart entre la plus grande marginale de la distribution et la valeur qu'aurait une distribution uniforme.

$$strength(x) = \max_{v \in D(x)} \left| \hat{\theta}_x(v) - \frac{1}{|D(x)|} \right|$$

L'heuristique *Max Marginal Strength* sélectionne la variable dont la distribution marginale possède la plus haute force, et lui assigne la valeur de marginale maximale.

$$x_s = \operatorname*{argmax}_{x \in X} strength(x)$$
$$v_s = \operatorname*{argmax}_{v \in D(x_s)} \hat{\theta}_{x_s}(v)$$

— **Min Entropy** [18] sélectionne la variable dont la distribution marginale possède l'entropie la plus basse (voir Section 2.3.4), et lui assigne la valeur de marginale maximale.

$$x_s = \operatorname*{argmin}_{x \in X} H(x)$$
$$v_s = \operatorname*{argmax}_{v \in D(x_s)} \hat{\theta}_{x_s}(v)$$

— **Min Normalized Entropy** [18] sélectionne la variable dont la distribution marginale possède l'entropie normalisée la plus basse, et lui assigne la valeur de marginale maximale.

$$x_s = \operatorname*{argmin}_{x \in X} H_{norm}(x)$$
$$v_s = \operatorname*{argmax}_{v \in D(x_s)} \hat{\theta}_{x_s}(v)$$

— **DomWDeg + Max Marginal** sélectionne la variable selon l'heuristique *DomWDeg* et sélectionne la valeur possédant la plus haute marginale.

$$x_s = \operatorname*{argmin}_{x \in X} domwdeg(x)$$
$$v_s = \operatorname*{argmax}_{v \in D(x_s)} \hat{\theta}_{x_s}(v)$$

CHAPITRE 3 REVUE DE LITTÉRATURE

Dans cette revue de littérature, nous proposons un tour d'horizon des techniques avancées d'inférence utilisées dans le contexte de la programmation par contraintes.

L'objectif de ces méthodes est de dépasser la cohérence d'arc généralisée et de disposer de plus d'informations sur lesquelles baser les décisions de branchement.

Ce chapitre mentionne plusieurs niveaux de cohérence plus forts que la cohérence d'arc généralisée, ou bien des approches enrichissant les informations du domaine, avec des distributions marginales par exemple.

3.1 Cohérence de chemin

Au sein d'un réseau de contraintes binaires, la cohérence d'arc est atteinte si chaque valeur v de chaque variable x possède un support dans le domaine des variables avec lesquelles elle partage des contraintes.

On dit que la cohérence de chemin est atteinte sur une paire de variable (x_i, x_j) par rapport à une troisième variable x_k si chaque paire de valeur $(v_i, v_j) \in D(x_i) \times D(x_j)$ peut être augmentée d'une valeur $v_k \in D(x_k)$ telle que le triplet (x_i, x_j, x_k) est cohérent avec l'ensemble des contraintes.

Ce niveau de cohérence plus élevé que la cohérence d'arc nécessite une complexité plus grande pour être établie. Dechter [1] présente un algorithme PC1 (*Path Consistency 1*) pour établir la cohérence de chemin dont la complexité est en $O(n^5k^5)$ où n est le nombre de variables et k borne la taille des domaines. En comparaison, l'algorithme AC4 (*Arc Consistency 4*) établit la cohérence d'arc avec une complexité dans $O(ek^2)$ avec e le nombre de contraintes binaires.

De plus, la cohérence de chemin n'est définie que dans le cadre des réseaux de contraintes binaires. Il est admis que modéliser les problèmes à l'aide de contraintes globales de forte arité est préférable pour faciliter la résolution des exemplaires [19]. Pour cette raison, la cohérence de chemin est d'une utilité limitée en CP, hors de certains exemplaires spécifiques comportant un grand nombre de variables binaires comme le corpus d'exemplaires Dubois [12].

3.2 Inférence par élimination de variable

Dechter [20] présente une méthode complète de résolution de CSPs et COPs, combinant, comme la CP, des aspects d'inférence et de recherche.

L'inférence prend la forme d'élimination de variable (ou *Bucket Elimination*), qui fait disparaître une variable tout en déduisant son impact sur le reste du CSP. Pour procéder à l'élimination d'une variable, on effectue d'abord une jointure naturelle entre les relations associées à cette variable (i.e. les contraintes auxquelles elle participe). On effectue ensuite une projection de la relation résultant de la jointure, afin que la variable éliminée n'y figure plus. Cette opération d'élimination est aussi appelée Cohérence Adaptative.

Cette technique d'élimination de variables est suffisante pour résoudre CSPs et COPs en s'appuyant uniquement sur de l'inférence, et plusieurs raffinements sont proposés par Dechter [20]. Cependant, elle possède une complexité spatiale exponentielle en fonction de la largeur de l'arbre induit par l'ordre d'élimination choisi.

Une opération de recherche consistant, comme en CP, à assigner une valeur à une variable, permet de mitiger le besoin en mémoire au prix d'une complexité temporelle plus élevée [20]. Une représentation par arbre ET/OU permet une représentation compacte de l'arbre de recherche.

L'assignation d'une variable par la recherche change la connectivité du graphe de contraintes et peut considérablement simplifier l'inférence. L'approche décrite par [20] alterne entre assignation pour décomposer le graphe et élimination de variable pour résoudre les sous-problèmes qui résultent de l'assignation.

3.3 Inférence basée sur le comptage de solution

Les méthodes d'inférence présentées dans cette section ont d'abord été introduites dans le contexte de CSPs, puis étendues aux COPs.

3.3.1 MaxSD : Densité Maximale de Solutions

Le comptage de solutions a été introduit par Pesant [21], permettant d'évaluer la densité de solutions pour une assignation donnée : la structure combinatoire mise en évidence par les contraintes globales se prête bien au comptage des solutions et des algorithmes de comptage ont été proposés pour diverses contraintes globales [22].

Définition 1 (Nombre de solutions) *Étant donnée une contrainte* $c(x_1, \ldots, x_n)$ *et des*

domaines finis respectifs D_i pour $1 \leq i \leq n$, soit $\#c(x_1, \ldots, x_n)$ le nombre de tuples dans la relation correspondante, appelé son nombre de solutions.

Définition 2 (Densité de solutions) *Étant donnée une contrainte $c(x_1, \ldots, x_n)$, des domaines finis respectifs D_i pour $1 \leq i \leq n$, une variable x_i dans la portée de c, et une valeur $v \in D_i$, on définit la densité de solution de la paire (x_i, d), mesurant si l'assignation est fréquente dans les tuples acceptés par c, par la formule :*

$$\sigma(x_i, v, c) = \frac{\#c(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_n)}{\#c(x_1, \ldots, x_n)}$$

MaxSD [23] est une heuristique de recherche sélectionnant des variables et valeurs associées à de fortes densités de solutions et permet une recherche performante de solutions pour les CSPs.

$$(x_s, v_s) = \underset{(x,v) \in X \times D}{\operatorname{argmax}} \max_{c \in C(x)} \sigma(x, v, c)$$

3.3.2 BP : Propagation de conviction avec Sum-Product

La propagation de conviction et Sum-Product représentent un raffinement de la méthode de comptage des solutions, basés sur un comptage pondéré des supports pour chaque contrainte globale et décrits dans la Section 2.3.1. Sum-Product permet d'obtenir pour chaque variable une distribution marginale approximant la proportion des solutions comportant une valeur donnée.

Plusieurs heuristiques exploitant ces marginales ont été explorées : l'heuristique *Max Marginal*, proposées par Babaki et al. [12], accélère significativement la recherche et est compétitive avec *DomWDeg*. Ce gain en performance est atteint malgré le coût considérable de l'inférence à chaque nœud, en comparaison avec l'utilisation des algorithmes de filtrage. Pesant [7] présente l'heuristique *Max Strength*, qui atteint également des performances de résolutions intéressantes pour les CSPs.

3.4 Problèmes d'optimisation sous contraintes

3.4.1 MaxSD* : Densité Maximale de Solutions de qualité satisfaisante

Pesant [8] a généralisé la densité de solutions aux problèmes d'optimisation par la densité de solutions basée sur les coûts. Pour des problèmes de minimisation, soit $\varepsilon \geq 0$ un paramètre de tolérance et $\#c^{\varepsilon}(x_1, \ldots, x_k, z, f)$ le nombre de solutions à $c(x_1, \ldots, x_k, z, f)$ avec $z \leq$

$(1+\varepsilon)\cdot \min_{t\in c(x_1,\ldots,x_k)} f(t)$. La densité de solutions basée sur les coûts est définie par :

$$\sigma^{\varepsilon}(x_i,d,c,\varepsilon)=\frac{\#c^{\varepsilon}(x_1,\ldots,x_{i-1},d,x_{i+1},\ldots,x_k,z,f)}{\#c^{\varepsilon}(x_1,\ldots,x_k,z,f)}$$

Pour $\varepsilon=0$, cela correspond à la densité sur les solutions optimales uniquement.

MaxSD* applique ce concept en comptant les solutions de qualité satisfaisante pour guider la recherche en sélectionnant le couple variable/valeur de densité maximale :

$$(x_s,v_s)=\underset{(x,v)\in X\times D}{\operatorname{argmax}}\ \max_{c\in C(x)}\sigma^{\varepsilon}(x,v,c,\varepsilon)$$

La difficulté principale consiste à choisir le seuil ε approprié : suffisamment restrictif pour orienter efficacement vers les meilleures solutions, mais assez permissif pour maintenir un nombre significatif de solutions candidates.

3.4.2 Max-Product

L'application du passage de messages au COPs privilégie souvent Max-Product à Sum-Product.

Max-Product a notamment été appliqué à des problèmes d'optimisation sous contraintes Distribués (DCOPs), où des agents indépendants fixent les valeurs des variables dont ils sont responsables. Un échange de messages Max-Product entre ces agents permet leur coordination et une résolution plus efficace [24] [25].

Toulbar et la Cohérence d'Arc Virtuelle

Le solveur Toulbar [26] est une adaptation fructueuse de l'algorithme Max-Product.

Avant de décrire cette approche, notons que la formulation des problèmes diffère puisque les COPs sont remplacés par des VCSPs (*Valued Constraint Satisfaction Problems*), où pour chaque contrainte c, chaque tuple accepté se voit assigné un coût non-négatif. Le coût total d'une solution est la somme des coûts pour l'ensemble des contraintes : $\sum_{c\in C} c(X)$. L'objectif est la minimisation de ce coût. Une telle représentation est aussi appelée CFN pour *Cost Function Network*.

Les contraintes d'un VCSP P sont considérées selon leurs arités : une contrainte d'arité nulle $c_\emptyset$ représente une borne inférieure du coût du VCSP. Les autres sont unaires (notées $c_{\{x\}}$) ou

d'arité multiple (notées c_S).

La résolution consiste en une série de réécritures des coûts associés à chaque contrainte, par des opérations SAC (*Soft Arc Consistency*) :

- `Project` : projette un poids d'une fonction de coût c_S d'arité deux ou plus à une fonction de coût unaire $c_{\{x\}}$, i.e. diminue le coût des tuples de c_S d'une valeur $\alpha \geq 0$ et augmente d'autant les coûts de $c_{\{x\}}$.
- `Extend` : opération inverse de `Project`. Projette le poids d'une fonction unaire vers une fonction d'arité multiple.
- `UnaryProject` : projette une fonction unaire vers la fonction d'arité nulle $c_\emptyset$.

Ces opérations permettent d'obtenir un VCSP équivalent (i.e. ayant les mêmes solutions) possédant une borne $c_\emptyset$ plus haute.

Cooper et al. [26] définissent un état de cohérence OSAC (*Optimal Soft Arc Consistency*), tel qu'il n'existe plus d'opération SAC augmentant $c_\emptyset$: on a atteint l'optimalité vis-à-vis du coût à minimiser. La complexité pour atteindre OSAC reste cependant prohibitive sur des problèmes de grande taille.

La Cohérence d'Arc Virtuelle (VAC) est une méthode moins coûteuse pour définir des séquences d'opérations SAC permettant d'augmenter $c_\emptyset$. On trouve de telles séquences en considérant un CSP $Bool(P)$, où les contraintes acceptent uniquement les tuples ayant un coût nul dans P. On effectue du filtrage grâce aux algorithmes classiques dont on a donné quelques exemples, afin d'atteindre la cohérence d'arc généralisée dans $Bool(P)$. Lorsqu'un domaine est vidé par le filtrage, c'est qu'une opération SAC est possible sur la contrainte responsable du filtrage. On trouve alors la valeur maximale λ dont on peut augmenter $c_\emptyset$. Cette maximisation est analogue au calcul d'un message Max-Product dans notre application.

CHAPITRE 4 PASSAGE DE MESSAGES POUR L'OPTIMISATION

Alors que l'utilité de CPBP a été démontrée pour les CSPs [7], certaines adaptations sont nécessaires dans le contexte des COPs et sont présentées dans ce chapitre. Nous décrivons ici l'usage de la contrainte Oracle pour introduire un biais favorable dans les distributions calculées et présentons des résultats préliminaires sur des exemplaires de carrés latins.

4.1 Contrainte Oracle

Dans les problèmes d'optimisation, il existe une forte préférence pour certaines solutions selon la valeur que leur associe la fonction objectif. On cherche à représenter cette préférence dans la propagation : obtenir un fort gradient dans les marginales de la variable objectif plutôt qu'une distribution uniforme.

Pour ce faire, nous avons recours à la contrainte Oracle [27] : une contrainte unaire qui n'effectue aucun filtrage mais peut envoyer des messages de force arbitraire à une variable.

On ajoute une contrainte Oracle portant sur la variable objectif. Pour un problème de maximisation, on configure la contrainte Oracle pour envoyer des messages normalisés proportionnels à chaque valeur :

$$\mu_{Oracle \to x}(v) = \left(\frac{v}{\sum_{v' \in D(x)} v'} \right)^{\alpha} \tag{4.1}$$

Nous supposons dans l'équation (4.1) que toutes les valeurs sont non négatives. Dans le cas contraire, nous pouvons les rendre non négatives en y ajoutant une constante. La contrainte Oracle exprime ainsi un biais pour les solutions où la variable objectif est plus grande, qui peut être propagé dans le réseau.

On considère ici un problème de maximisation sans perte de généralité : pour un problème de minimisation d'une variable objectif z, on peut introduire dans le modèle une variable $z' = -z$, une contrainte $Oracle(z')$, et maximiser z'.

Afin de contrôler la force du biais, on utilise un coefficient $\alpha > 0$ comme exposant dans la force du message.

4.1.1 Effet sur Max-Product

Dans un problème de satisfaction, toutes les solutions sont considérées équivalentes. La propagation de Max-Product va alors produire des distributions uniformes et n'apporte pas

d'information qui ne soit déjà fournie par la cohérence d'arc généralisée (voir Section 2.3.2).

L'Oracle permet à Max-Product de converger vers des distributions marginales plus utiles : le gradient introduit dans la distribution de la variable objectif va être propagé dans les distributions des variables de décisions. Idéalement, nous souhaitons converger vers une distribution où pour chaque variable, l'affectation représentée dans la solution préférée est approximativement 1, tandis que les autres sont proches de 0.

En revenant à l'Exemple 2.3.1 et en considérant un objectif de maximiser la valeur de a et en ajoutant l'$Oracle(a)$ approprié, nous obtenons une distribution qui désigne fortement la solution (a = 3, b = 2, c = 1, d = 1), qui présente la plus grande valeur de a (voir Tableau 4.1).

4.1.2 Influence du poids de l'Oracle

La préférence introduite par la contrainte Oracle est un biais fort qui peut parfois produire des distributions ne reflétant pas les solutions du problème.

Considérons le même exemple, mais avec l'objectif de minimiser la valeur de a et la contrainte $Oracle(a)$ correspondante. Avec Max-Product et Oracle, les marginales convergent après seulement 5 itérations. Le résultat est montré dans le Tableau 4.2. Les distributions calculées ne donnent pas une réponse claire et sont en fait trompeuses car elles suggèrent $a = 1$. Dans ce cas, le biais vers une valeur faible pour a introduit par la contrainte Oracle est trop fort. Celui-ci peut être atténué en choisissant judicieusement un poids $\alpha < 1$. Les distributions obtenues pour $\alpha = 0.8$ sont montrées dans le Tableau 4.3 (5 itérations) et le Tableau 4.4 (20 itérations). Nous permettons ici un maximum de 20 itérations car un poids plus faible fait converger les distributions plus lentement. Les distributions après 5 itérations sont peu satisfaisantes, mais la solution optimale est désignée clairement après 20 itérations.

Empiriquement, il s'est avéré qu'un poids de $\alpha = 0.8984$ ou moins permet de converger vers une distribution appropriée similaire au cas de maximisation. Un exposant plus grand que $\alpha = 0.8984$ conduira au comportement problématique introduit dans cette section. Cette valeur de $\alpha = 0.8984$ n'a ce rôle particulier que pour cet exemplaire précis, et l'influence du poids de l'Oracle sera l'objet d'autres expérimentations dans la Section 6.2.5. Il faut noter qu'un poids plus petit ralentit la convergence des distributions, et nécessitera plus d'itérations pour obtenir une distribution informative. Le poids de la contrainte Oracle peut être un paramètre important pour que l'algorithme Max-Product soit efficace.

	1	2	3	4
a	0.00003	0.017	0.95	0.03
b	0.01	0.98	0.008	0.00008
c	0.998	0.002	0	0
d	0.98	0.02	0	0

TABLEAU 4.1 Distributions après 10 itérations avec Max-Product et Oracle pour maximiser a ($\alpha = 1$)

	1	2	3	4
a	0.48	0.27	0.18	0.07
b	0.14	0.26	0.34	0.26
c	0.2	0.37	0.27	0.15
d	0.24	0.33	0.24	0.18

TABLEAU 4.2 Distributions avec Max-Product et Oracle pour minimiser a ($\alpha = 1$), après seulement 5 itérations car l'algorithme converge plus rapidement dans ce cas. Les distributions ne mettent pas en évidence les solutions désirées.

4.1.3 Effet sur Sum-Product

Les messages de la contrainte Oracle affecteront aussi les distributions calculées par l'algorithme Sum-Product. Avec un poids $\alpha = 1$, la propagation converge après 10 itérations. Comme le montrent le Tableau 4.6 et le Tableau 4.7, les marginales obtenues sont proches de 1 pour l'affectation associée à la solution optimale et proches de 0 pour les autres affectations.

Pour Sum-Product comme pour Max-Product, on peut expliquer comme suit la plus grande facilité à obtenir les bonnes distributions en maximisation. La valeur 4 pour a et b, si elle n'est pas filtrée par la cohérence d'arc, est problématique pour satisfaire à la fois AllDifferent(a, b, c) et $a + b + c + d = 7$. Le poids de $a = 4$ diminue donc rapidement lors de l'inférence par propagation, et on privilégie ainsi $a = 3$ pour la maximisation. En comparaison, l'assignation $a = 1$ pose un problème moins évident, bien qu'elle n'appartienne à aucune solution. En minimisation, l'Oracle crée un biais supplémentaire pour les valeurs basses de a et le poids de $a = 1$ ne tend pas vers 0.

	1	2	3	4
a	0.44	0.27	0.20	0.09
b	0.16	0.26	0.32	0.26
c	0.22	0.34	0.27	0.17
d	0.25	0.31	0.25	0.20

TABLEAU 4.3 Distributions après 5 itérations avec Max-Product et Oracle pour minimiser a avec $\alpha = 0.8$. L'algorithme n'a pas encore convergé, supposément à cause du poids réduit de l'Oracle.

	1	2	3	4
a	0.01	0.94	0.05	0
b	0	0.06	0.92	0.02
c	0.99	0.01	0	0
d	0.93	0.07	0	0

TABLEAU 4.4 Distributions après 20 itérations avec Max-Product et Oracle pour minimiser a avec $\alpha = 0.8$. L'algorithme converge vers la distribution appropriée mais plus lentement.

4.2 Expériences préliminaires

Des tests préliminaires ont été effectués sur un nombre limités d'exemplaires de carrés latins afin de mettre en évidence l'utilité des marginales produites par Max-Product dans le guidage de la recherche et la résolution de COPs.

4.2.1 Configuration expérimentale

Considérant l'adaptation du CSP des carrés latins en COPs définie en Section 2.1.2, nous avons considéré 30 exemplaires de carrés latins de taille 20×20, avec pour objectif de maximiser la somme des cases dans une zone rectangulaire de taille 5×4, située en haut à gauche du carré. Les exemplaires se répartissent en 3 séries de 10 exemplaires comportant respectivement 200, 250 et 300 "trous". Ils ont été générés avec le programme `lsencode` [28] conçu pour produire des exemplaires difficiles à résoudre.

On utilise le solveur MiniCPBP [7] avec un processeur AMD EPYC 7532 (Zen 2) @ 2.40 GHz, 256M cache L3, sur Linux, et on accorde à chaque résolution 8GB de RAM et une limite de temps de 30 minutes.

	1	2	3	4
a	0.46	0.27	0.19	0.08
b	0.15	0.26	0.33	0.26
c	0.21	0.35	0.27	0.16
d	0.24	0.33	0.24	0.19

TABLEAU 4.5 Distributions après 5 itérations avec Max-Product et Oracle pour minimiser a avec $\alpha = 0.9$, la distribution désirée n'apparaît pas

	1	2	3	4
a	0	0.01	0.99	0
b	0	0.99	0.01	0
c	1	0	0	0
d	0.99	0.01	0	0

TABLEAU 4.6 Distributions après 10 itérations avec Sum-Product et Oracle pour maximiser a ($\alpha = 1$). Les distributions sont biaisées en faveur de la solution optimale.

Pour les résolutions impliquant le passage de messages, on se limite à 10 itérations de propagation pour chaque nœud de la recherche.

4.2.2 Heuristiques évaluées

On compare les performances de plusieurs heuristiques.

Les deux premières n'exploitent pas le passage de messages et sont des heuristiques *ad hoc* conçues spécifiquement pour le problème d'optimisation du carré latin, sur la base de notre connaissance de la contribution de chaque variable de décision à l'objectif (i.e. si c'est une case "objectif" ou non). Dans la suite, on note $O \subset X$ les variables faisant partie de l'ensemble à maximiser :

— **Max Value** : heuristique "gloutonne" qui fixe en priorité les variables de l'objectif, sélectionnant la plus haute valeur disponible. En reprenant le formalisme de la Sec-

	1	2	3	4
a	0.03	0.86	0.11	0
b	0	0.15	0.83	0.02
c	0.98	0.02	0	0
d	0.93	0.07	0	0

TABLEAU 4.7 Distributions après 10 itérations avec Sum-Product et Oracle pour minimiser a ($\alpha = 1$). Les distributions sont biaisées en faveur de la solution optimale.

tion 2.2.3 :

$$x_s = \underset{x \in O}{\operatorname{argmax}} \left(\max D(x) \right)$$
$$v_s = \max(D(x_s))$$

— **DomWDeg + Best Value** : On utilise l'heuristique bien établie pour la sélection de variable *DowWDeg* [11] avec une sélection de valeur conçue pour le problème proposé : une fois la variable sélectionnée, on prend sa valeur maximale si elle correspond à une case objectif et sa valeur minimale sinon.

Formellement :

$$x_s = \underset{x \in X}{\operatorname{argmin}} \, domwdeg(x)$$
$$v_s = \begin{cases} \max(D(x_s)) & \text{si } x_s \in O \\ \min(D(x_s)) & \text{sinon} \end{cases}$$

La dernière heuristique exploite les marginales calculées par propagation et plus précisément l'entropie normalisée de la distribution de chaque variable.

— **Min Normalized Entropy** : heuristique introduite en Section 2.3.5, on choisit la variable dont la distribution a la plus basse entropie normalisée, autrement dit celle pour laquelle notre certitude est la plus forte. On sélectionne bien entendu la valeur présentant la marginale la plus haute.

$$x_s = \underset{x \in X}{\operatorname{argmin}} H_{norm}(x)$$
$$v_s = \underset{v \in D(x_s)}{\operatorname{argmax}} \hat{\theta}_{x_s}(v)$$

Cette heuristique est utilisée avec trois approches de passage de messages :

— Sum-Product, sans contrainte Oracle sur la variable objectif.
— Sum-Product, avec une contrainte Oracle sur la variable objectif.
— Max-Product, impliquant nécessairement une contrainte Oracle sur la variable objectif, car le résultat est sinon équivalent à la cohérence d'arc généralisée (voir la Section 2.3.2).

4.2.3 Résultats

Résolution à l'optimalité

Un exemplaire est dit résolu à l'optimalité si le solveur a trouvé la solution optimale et prouvé son optimalité dans le temps imparti. La Figure 4.1 représente l'évolution du nombre d'exemplaires résolus à l'optimalité.

Max-Product est efficace pour résoudre les exemplaires à l'optimalité et en résout la moitié à l'optimalité en moins de 20 échecs. Sur des nombres d'échecs bas, l'algorithme a une nette avance sur Sum-Product et les heuristiques ad hoc. Il est en revanche rattrapé par Sum-Product autour de 200 échecs, puis dépassé par celui-ci. On remarque également qu'un des exemplaire n'est pas résolu par Max-Product dans le temps imparti.

On note également le peu d'influence qu'à la présence d'Oracle sur la résolution à l'optimalité : si la contrainte Oracle biaise les distributions vers des scores plus haut, elle ne fait pas ressortir la solution optimale aussi fortement que Max-Product.

Lorsqu'on considère le temps écoulé pour la résolution, on prend la mesure du coût que représente l'inférence. Les heuristiques n'utilisant pas la propagation mettent moins de temps à résoudre l'ensemble des exemplaires, bien qu'elles rencontrent plus d'échecs. Max-Product surpasse cependant Sum-Product pour la vitesse de résolution à l'optimalité.

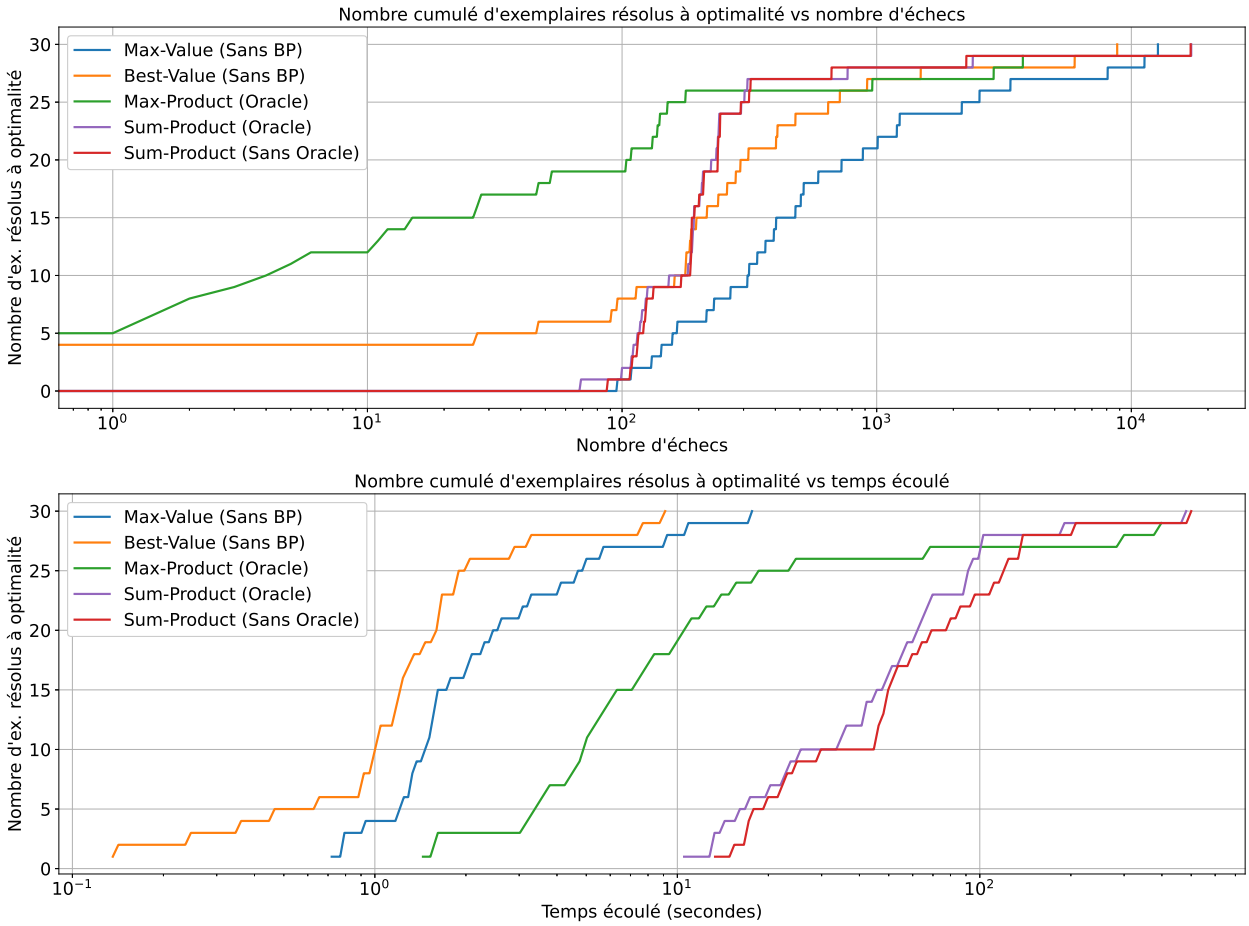

FIGURE 4.1 Évolution du nombre d'exemplaires résolus à l'optimalité

Score normalisé moyen

Afin d'évaluer la qualité des solutions trouvées au-delà de leur caractère optimal, on introduit la notion de **score normalisé**.

Pour une instance donnée, soit s_{best} le meilleur score trouvé parmi toutes les résolutions (score le plus faible pour un problème de minimisation) et s_{worst} le pire score trouvé. Pour une résolution avec un score courant s, le score normalisé $\hat{s}$ est défini par :

$$\hat{s} = \frac{s_{\text{worst}} - s}{s_{\text{worst}} - s_{\text{best}}} \tag{4.2}$$

Cette normalisation garantit que $\hat{s} = 1$ pour le meilleur score possible ($s = s_{\text{best}}$) et $\hat{s} = 0$ pour le pire score ($s = s_{\text{worst}}$). Dans le cas particulier où $s_{\text{best}} = s_{\text{worst}}$ (une seule valeur de score pour l'instance), on attribue $\hat{s} = 1$ si une solution existe et $\hat{s} = 0$ sinon.

Le **score normalisé moyen** sur un ensemble de résolutions R est alors :

$$\bar{S}_{\text{norm}} = \frac{1}{|R|} \sum_{r \in R} \hat{s}_r \tag{4.3}$$

Cette métrique permet de comparer la performance relative des algorithmes indépendamment de l'échelle des scores des différentes instances, et d'analyser l'évolution de la qualité des solutions en fonction du nombre d'échecs durant la recherche.

Pour les exemplaires de carrés latins, on a résolu à l'optimalité le problème en minimisation et maximisation. Ainsi, le score est normalisé entre le score optimal et le score de la pire solution satisfaisant les contraintes.

Considérons l'évolution du score normalisé moyen présenté en Figure 4.2.

On remarque d'abord que Sum-Product a le plus haut score pour de très bas nombres d'échecs. En effet, Sum-Product trouve une solution pour chaque exemplaire sans échec, tandis que les autres heuristiques n'ont pas encore de solution pour certains exemplaires. Le score de 0 associé à l'absence de solution fait chuter leur moyenne. On observe toujours une légère influence positive de l'Oracle sur les scores des solutions trouvées par Sum-Product.

Rappelons que comme vu dans la section détaillant les premières solutions, Max-Product a requis de 20 à 200 échecs pour trouver une première solution pour 6 des 30 exemplaires. Cela contribue à un score moyen assez bas jusqu'à ce nombre d'échecs, et proche de 1.0 une fois qu'une solution est trouvée pour chaque exemplaire. Sur le graphe décrivant l'évolution en fonction du temps, Max-Product a un profil similaire à Sum-Product, bien que prenant un léger retard entre 5 et 10 secondes de résolution.

On note encore une fois l'avantage des heuristiques ne nécessitant pas de propagation du point de vue du temps.

Première solution trouvée

On considère d'abord les caractéristiques de la première solution trouvée par chacune des heuristiques, qui sont présentées dans la Figure 4.3.

Sum-Product, algorithme privilégié pour les problèmes de satisfaction, trouve pour chaque exemplaire une solution sans rencontrer aucun échec. L'ajout de la contrainte Oracle introduit un biais dans les distributions qui provoque un échec sur un exemplaire, mais diminue significativement l'écart d'optimalité des premières solutions trouvées.

Max-Product trouve la première solution avec un nombre d'échecs assez bas, bien que rare-

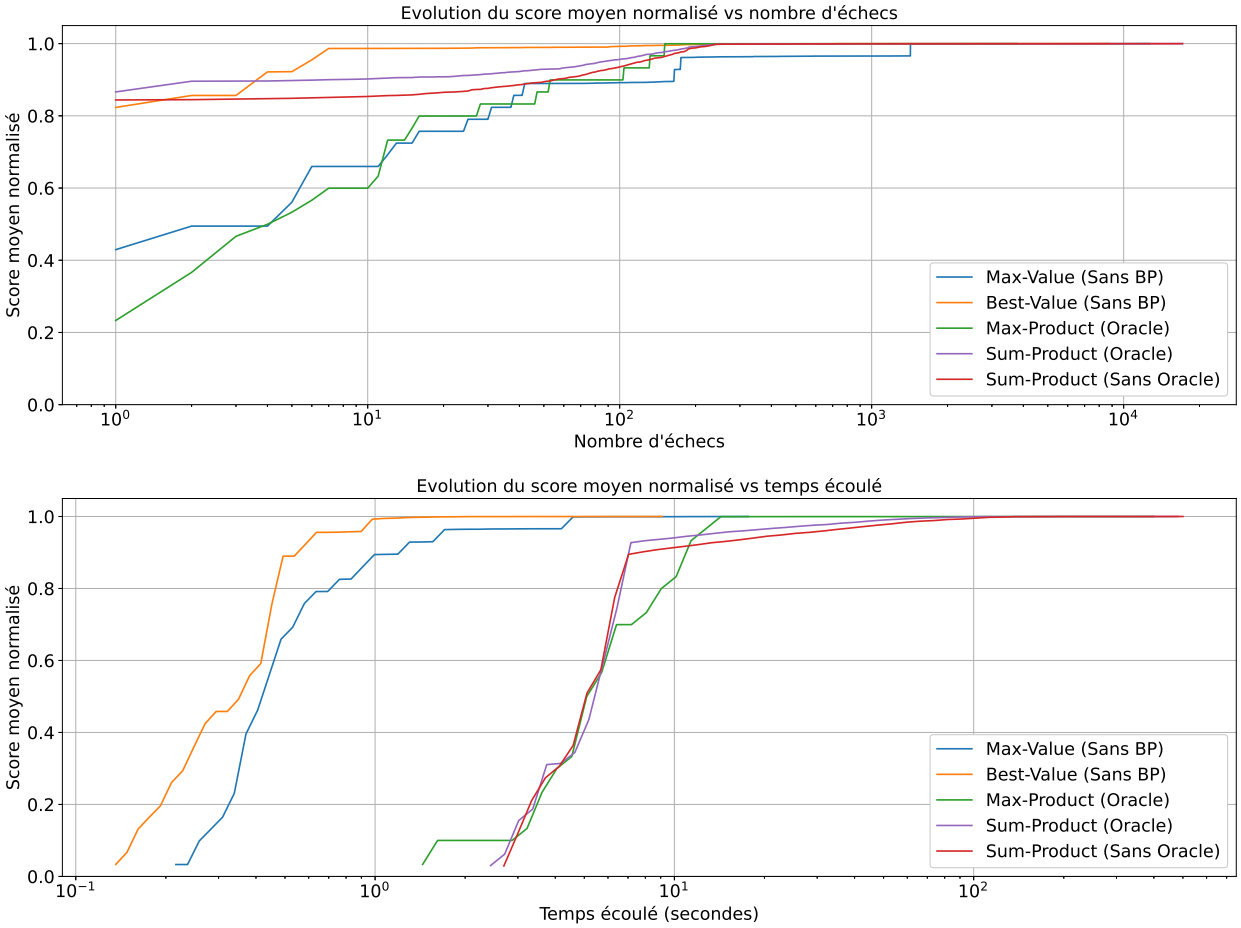

FIGURE 4.2 Évolution du score normalisé moyen pour l'ensemble des exemplaires

ment nul. La première solution est optimale dans 21 cas sur 30 et dans les 9 cas restants, l'écart d'optimalité est de moins de 10%. En comparaison, aucune des premières solutions trouvées par Sum-Product n'est optimale.

Les heuristiques ad hoc ont de bonnes perfomances, mais sont battues par Sum-Product et Max-Product dans leur spécialité respective, le nombre d'échecs et le score de la première solution.

Conclusions préliminaires

Cette expérience à petite échelle permet de mettre en valeur certaines caractéristiques des algorithmes de passage de messages dans le contexte des COPs :

— **Sum-Product** présente l'avantage de trouver rapidement une première solution sans échec pour tous les exemplaires, mais sans optimisation particulière du score. L'ajout

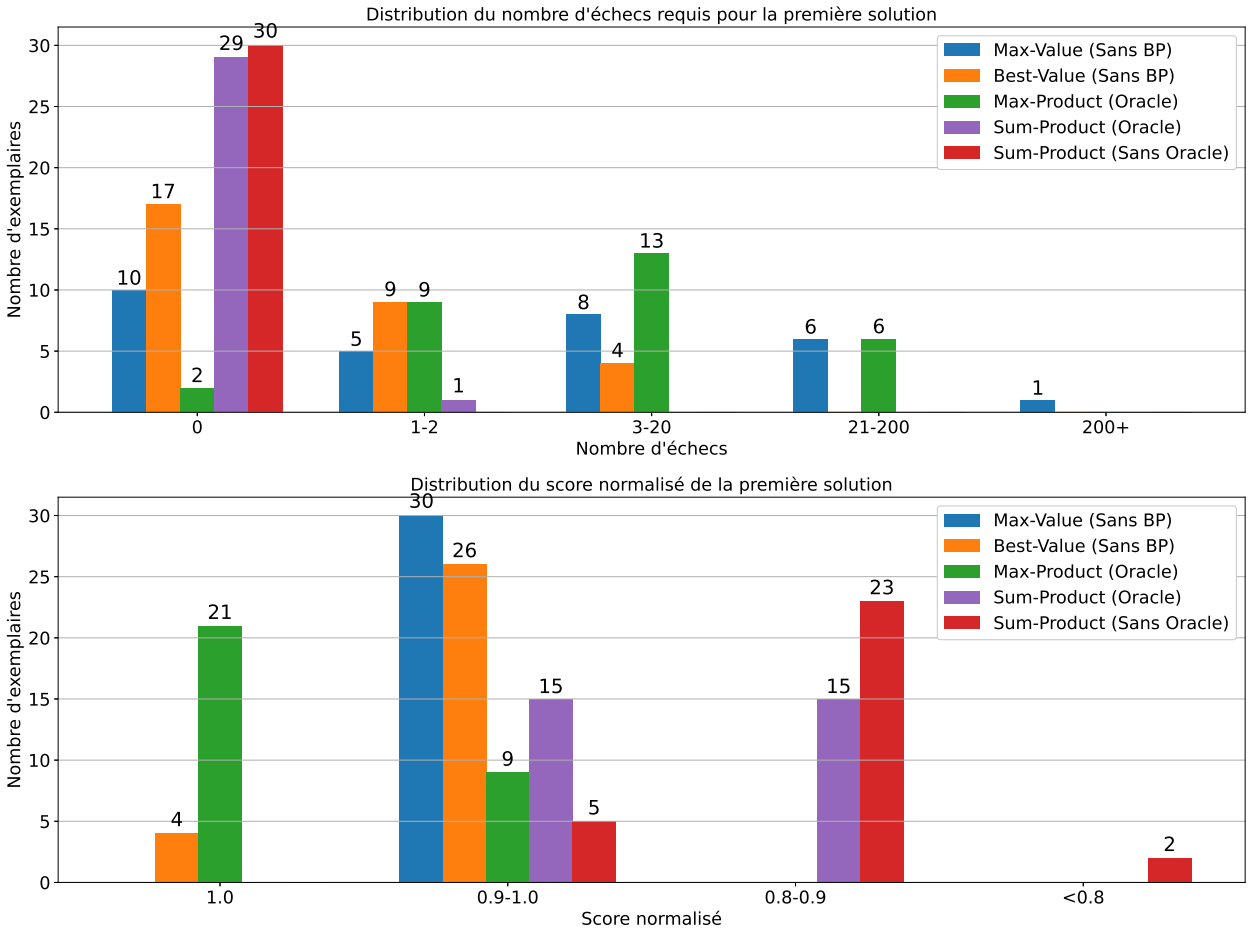

FIGURE 4.3 Caractéristiques de la première solution trouvée

d'un Oracle améliore légèrement la qualité des solutions au prix de la possibilité de quelques échecs supplémentaires.

— **Max-Product** privilégie la qualité des solutions, trouvant l'optimal en première solution dans 70% des cas et maintenant des écarts d'optimalité inférieurs à 10% dans les autres cas. Il se montre efficace pour atteindre l'optimalité avec un nombre d'échecs relativement faible.

— Le **coût computationnel** de la propagation est notable, les heuristiques sans passage de messages résolvant l'ensemble des exemplaires plus rapidement malgré un nombre d'échecs supérieur. Néanmoins, Max-Product surpasse Sum-Product en termes de temps de résolution.

— Les **heuristiques ad hoc** (Max Value et DomWDeg + Best Value) montrent de bonnes performances mais restent spécifiques au problème de carré latin, nécessitant une connaissance *a priori* de la structure du problème. Ici, on exploite une bonne

connaissance de la contribution de chaque variable de décision à la variable objectif.

— Les résultats suggèrent un **potentiel de généralisation** pour le passage de messages : on espère que les algorithmes basés sur les marginales permettront de proposer des heuristiques efficaces et polyvalentes sur des problèmes plus variés, sans intervention de l'opérateur, contrairement aux heuristiques ad hoc qui nécessiteraient une adaptation spécifique à chaque problème.

On tentera par la suite de valider ces comportements dans des expériences de plus grande envergure.

CHAPITRE 5 ALGORITHMES MAX-PRODUCT

La propagation de conviction avec l'algorithme Max-Product nécessite des algorithmes de calcul de messages pour chaque contrainte globale. Ce chapitre développe ces algorithmes pour quatre contraintes fondamentales.

Pour les contraintes Table, Sum et Regular, le Max-Product utilise les même structures de données et parcours que le Sum-Product en modifiant l'opération d'accumulation. La contrainte AllDifferent requiert une approche qualitativement très différente, basée sur les problèmes d'assignation dans les graphes bipartis.

5.1 Table

Pour calculer le message Max-Product de la contrainte Table (Algorithme 4, des structures de données et algorithmes décrites pour Sum-Product (Algorithme 2) nous permettent d'obtenir le poids des solutions :

$$\frac{w(t)}{\mu_{x\to c}(v)} = \prod_{\substack{(x',v')\in t \\ x'\neq x}} \mu_{x'\to c}(v')$$

On accumule ces termes avec un maximum plutôt qu'une somme :

$$\forall(x,v), \quad \mu_{x\to c}(v) = \max_{t\in T:(x,v)\in t} \prod_{\substack{(x',v')\in t \\ x'\neq x}} \mu_{x'\to c}(v')$$

5.2 Sum

Pour le calcul du message de la contrainte Sum, on parcourt le graphe de programmation dynamique décrivant les sommes partielles comme pour le message Sum-Product, mais en conservant le maximum rencontré plutôt que d'accumuler une somme.

Pour un unique message Max-Product, il pourrait être plus efficace de calculer un plus court chemin plutôt que d'effectuer la passe complète qu'impose le Sum-Product. Cependant la passe en arrière complète permet le calcul de l'ensemble des messages en une unique passe, et reste l'approche la plus avantageuse.

Algorithm 4: Mise à jour Max-Product pour contrainte Table

Data: Variables $x_1, \ldots, x_n$, table de contrainte T, ensembles de support $supports[i][v]$
Result: Convictions locales mises à jour $\mu_{c \to x_i}(v)$

```
// Calculer les tuples supportés : identique à Sum-Product
// Calculer les poids des tuples : identique à Sum-Product
// Mettre à jour les convictions pour chaque couple variable/valeur
```

for $i \leftarrow 1$ **to** n **do**
  for $v \in D(x_i)$ **do**
    $conviction \leftarrow 0$;
    $outsideConviction \leftarrow \mu_{x_i \to c}(v)$;
    for $t \in supports[i][v] \cap supportedTuples$ **do**
      `// Ici, on conserve le maximum`
      $conviction \leftarrow \max(conviction, \frac{tupleWeight[t]}{outsideConviction})$;
    end
    $\mu_{c \to x_i}(v) \leftarrow conviction$;
  end
end

5.3 Regular

Comme pour le filtrage et le Sum-Product, la contrainte Regular est analogue à la contrainte Sum. Dans le graphe de programmation dynamique des états atteints pour les préfixes, on effectue une passe en avant et en arrière, en conservant le poids maximum des chemins.

5.4 AllDifferent

Le calcul des messages Max-Product de la contrainte AllDifferent se distingue particulièrement de son équivalent Sum-Product.

Comme vu en Section 2.2.2, on peut représenter les variables de la portée d'une contrainte AllDifferent dans un graphe biparti $G = (X, V, E)$. X est alors un sous-ensemble de sommets représentant les variables, et V un sous-ensemble de sommets représentants les valeurs. L'ensemble des arêtes représente l'appartenance d'une valeur au domaine de la variable correspondante, autrement dit :

$$\forall (x, v) \in X \times V, (x, v) \in E \iff v \in D(x)$$

Un tuple accepté par une contrainte AllDifferent correspond à un couplage couvrant X dans un tel graphe.

Le message Max-Product $\mu_{c\to x}(v)$ correspond au produit des poids des arêtes dans un couplage couvrant X, incluant l'arc (x, v), et pour lequel ce produit est maximal.

La matrice d'adjacence initiale est composée des messages normalisés reçus des variables $\hat{\mu}_{x\to c}(v)$ et a donc des coefficients compris entre 0 et 1. Par conséquent, si l'on transforme ces coûts en prenant l'opposé du logarithme $-log(\hat{\mu}_{x\to c}(v))$, on obtient une matrice d'adjacence de coûts positifs, et l'on cherche à présent un couplage minimisant la somme du poids des arêtes qui le compose. Ceci correspond au problème d'assignation étudié extensivement dans la littérature [29].

Alors que le calcul du message Sum-Product est un problème de la classe de complexité $\#P$, le problème d'assignation se trouve dans P et il existe plusieurs algorithmes pour le résoudre en temps polynomial.

5.4.1 Trouver un unique couplage minimal

L'algorithme Hongrois est une méthode bien établie pour résoudre le problème d'assignation [30] mais son implémentation reposant sur la mise à jour d'une matrice de coûts réduits ne se prête pas bien à l'optimisation lorsque de nombreux messages doivent être calculés.

Crouse [31] présente une approche basée sur le calcul du chemin augmentant le plus court qui résout le problème d'assignation rectangulaire, i.e. où le nombre de colonnes candidates est plus grand que le nombre de lignes à assigner. Cette approche décrite par l'algorithme est incrémentale et construit le couplage une ligne à la fois. Pour chaque ligne non appariée, l'algorithme trouve le chemin augmentant le plus court dans le graphe, qui se termine à une colonne non appariée (puits). L'idée clé est que chaque itération augmente le couplage partiel actuel d'exactement une paire. L'algorithme maintient des variables duales u_i et v_j qui satisfont les conditions de complémentarité du programme linéaire dual. Lorsqu'un chemin augmentant est trouvé, l'algorithme met à jour ces variables duales pour maintenir l'optimalité tout en étendant le couplage.

La nature incrémentale assure qu'une fois qu'une ligne est appariée, elle reste dans le couplage tout au long des itérations suivantes, seule l'assignation spécifique de colonne pouvant changer lorsque les chemins sont augmentés à travers la structure existante.

On peut calculer un message $\mu_{c\to x}(v)$ grâce à cet algorithme. On retire d'abord la variable x et la valeur v du graphe (ou autrement dit, on retire la ligne et la colonnes correspondantes de la matrice d'adjacence). On trouve le couplage minimal dans le graphe restant ; son poids correspond à la valeur du message à envoyer.

5.4.2 Pour l'ensemble des messages d'une variable

Si l'algorithme présenté dans la sous-section précédente est efficace pour calculer un couplage minimal, et donc un unique message, calculer de multiples couplages pour l'ensemble des messages envoyés par la contrainte reste très coûteux et implique une grande quantité de calculs redondants. Dans une portée de n variables, chaque message nécessite $n-1$ appels à la procédure du plus court chemin augmentant. On propose dans l'Algorithme 6 une variante pour calculer plus efficacement tous les messages concernant une variable x en réutilisant un couplage intermédiaire.

Pour ce faire, on place la variable x en dernière position, puis on construit le couplage minimal impliquant les $n-1$ premières variables.

À partir de ce couplage initial, on construit pour une valeur v un chemin augmentant à partir de x en forçant la première arête à (x, v), avec l'Algorithme 7. Le couplage résultant est le couplage de poids minimal parmi ceux contenant (x, v). On retranche le poids de l'arête (x, v) pour obtenir la valeur du message $\mu_{c\to x}(v)$.

Pour chaque valeur v on réutilise le couplage initial de taille $n-1$. Par conséquent, on effectue la procédure du plus court chemin augmentant $n-1$ fois par variable, ainsi qu'une seule fois par valeur du domaine de la variable.

Si l'on considère un cas simplifié où chacune des n variables possède un domaine de taille d, la variante proposée calcule l'ensemble des messages en $n(n-1)+nd$ appels à la procédure du plus court chemin augmentant, contre $n(n-1)d$ pour l'approche précédente.

La validation empirique a été minimale, mais des accélérations autour de $\times 10$ ont été observées sur quelques instances de carrés latins. L'introduction de cette optimisation a rendu compétitive l'approche Max-Product dans l'expérience préliminaire présentée dans la Section 4.2 .

Algorithm 5: Max-Product pour la contrainte Sum, cet algorithme est presque identique à l'Algorithme 3 : seule l'accumulation est changée d'une somme à un maximum

Data: Variables $x_1, \ldots, x_n$, Bornes des domaines j_{min}, j_{max}, messages reçus $outsideBelief[x_i][v]$

Result: Convictions locales mises à jour $localBelief[x_i][v]$

`// Initialisation des passes en avant et en arrière`

Initialize π_{in}, π_{out} to 0;

$\pi_{in}[1][0] \leftarrow 1$; $\pi_{out}[n][0] \leftarrow 1$;

`// Passe en avant`

for $i \leftarrow 1$ **to** $n-1$ **do**

  foreach $v \in D(x_i)$, $j \in [j_{min} - \min(0, v), j_{max} - \max(0, v)]$ **do**

    if $\pi_{in}[i][j] > 0$ **then**

      `// On conserve le maximum au lieu d'accumuler une somme`

      $\pi_{in}[i+1][j+v] \leftarrow \max(\pi_{in}[i+1][j+v], \pi_{in}[i][j] \times outsideBelief[x_i][v])$;

    end

  end

end

`// Passe en arrière et mise à jour des convictions locales`

for $i \leftarrow n$ **to** 2 **do**

  foreach $v \in D(x_i)$ **do**

    $localBelief[x_i][v] \leftarrow 0$;

    for $j \leftarrow j_{min} - \min(0, v)$ **to** $j_{max} - \max(0, v)$ **do**

      if $\pi_{out}[i][j+v] > 0$ **then**

        $\pi_{out}[i-1][j] \leftarrow \pi_{out}[i-1][j] + \pi_{out}[i][j+v] \times outsideBelief[x_i][v]$;

        `// On conserve le maximum au lieu d'accumuler une somme`

        $localBelief[x_i][v] \leftarrow \max(localBelief[x_i][v], \pi_{in}[i][j] \times \pi_{out}[i][j+v])$;

      end

    end

  end

end

`// Pour la première variable`

$localBelief[x_1][v] \leftarrow \pi_{out}[1][v]$ pour tout $v \in D(x_1)$;

Algorithm 6: CalculerTousMessagesVariable(C, x)

Input: matrice de coûts $C \in \mathbb{R}^{n \times m}$; variable cible x
Output: ensemble des messages $\{\mu_{c \to x}(v) : v \in \text{domaine}(x)\}$
`// Placer la variable` x `en dernière position`
Réorganiser C pour que x corresponde à la dernière ligne;
`// Construire le couplage initial sans la variable` x
$C_{\text{réduite}} \leftarrow$ sous-matrice de C excluant la ligne x;
couplage_initial $\leftarrow$ AssignationRectangulaire($C_{\text{réduite}}$);
`// Initialiser le tableau des messages`
messages$[v] \leftarrow 0$ pour tout $v \in$ domaine(x);
`// Calculer le message pour chaque valeur du domaine de` x
foreach $v \in$ *domaine*(x) **do**
    `// Trouver le couplage optimal contenant` (x, v)
    couplage_avec_xv $\leftarrow$
     CheminAugmentantPlusCourtArêteForcée(C, couplage_initial, x, v);
    `// Calculer le poids total du couplage`
    poids_total $\leftarrow \sum_{(i,j) \in \text{couplage\_avec\_xv}} C[i, j]$;
    `// Soustraire le poids de l'arête forcée` (x, v)
    messages$[v] \leftarrow$ poids_total $- C[x, v]$;
return *messages*;

Algorithm 7: CheminAugmentantPlusCourtArêteForcée$(C, \text{matching}, x, v)$

Input: matrice de coûts $C \in \mathbb{R}^{n \times m}$; couplage actuel *matching* ; variable x ; valeur forcée v

Output: couplage optimal contenant l'arête (x, v)

colonnes_restantes $\leftarrow \{0, 1, \ldots, m-1\} \setminus \{v\}$;
coût_min$[j] \leftarrow \infty$ pour tout j;
prédécesseur$[j] \leftarrow -1$ pour tout j;
ligne_courante $\leftarrow x$;
colonne_forcée $\leftarrow v$;
`// Forcer la première arête` (x, v)
if *matching*$[v]$ = *libre* **then**
    `// Colonne` v `libre, chemin trouvé directement`
    return *couplage augmenté avec* (x, v);
else
    ligne_courante $\leftarrow$ matching$[v]$;
while *aucun puits trouvé* **do**
    foreach $j \in$ *colonnes_restantes* **do**
        coût_réduit $\leftarrow C[\text{ligne\_courante}, j] + \text{ajustement\_dual}$;
        if *coût_réduit* $<$ *coût_min*$[j]$ **then**
            coût_min$[j] \leftarrow$ coût_réduit;
            prédécesseur$[j] \leftarrow$ ligne_courante;
    $j^* \leftarrow \arg\min_{j \in \text{colonnes\_restantes}} \text{coût\_min}[j]$;
    if *matching*$[j^*]$ = *libre* **then**
        return *chemin augmentant se terminant en* j^*;
    else
        ligne_courante $\leftarrow$ matching$[j^*]$;
        Retirer j^* de colonnes_restantes;

CHAPITRE 6 RÉSULTATS EXPÉRIMENTAUX

Ce chapitre propose une analyse empirique plus approfondie de l'efficacité des heuristiques basées sur le passage de messages pour l'optimisation.

6.1 Configuration expérimentale

On évalue la résolution à l'aide de MiniCPBP [7] sur 321 exemplaires issus du corpus XCSP [32]. Ces exemplaires ont été sélectionnés car ils sont entièrement modélisés à l'aide de l'ensemble de contraintes pour lesquelles nous avons pu implanter un calcul de message Max-Product : AllDifferent, Table, Sum et Regular qui sont décrites au chapitre 5, ainsi que diverses contraintes plus simples comme Equal, NotEqual et LessThan. Les problèmes représentés sont : *Clique*, *Drinking*, *GolombRuler*, *Knapsack*, *LitPuzzle*, *LowAutocorrelation*, *MaximumDensityOscillatingLife*, *Pyramid*, *SameQueensKnights*, *StillLife* et *StillLifeWastage*.

On a également sélectionné des exemplaires de tailles modérées afin de pouvoir effectuer la résolution en allouant 8GB de RAM par résolution. On configure la JVM pour que la taille de pile maximale et la taille de pile initiale soient toutes les deux égales à 7GB, dans le but d'exploiter l'intégralité de la RAM disponible et d'effectuer l'allocation en une fois en début de résolution.

La limite de temps pour chaque résolution est de 30 minutes.

La résolution est effectuée avec un processeur AMD EPYC 7532 (Zen 2) @ 2.40 GHz, 256M cache L3, sur Linux.

6.2 Résultats

De nombreux paramètres peuvent influencer la propagation et la recherche, tels que le nombre d'itération d'envoi de messages, le critère de branchement, le poids accordé à l'Oracle. On cherche à étudier leur influence et à identifier une combinaison performante.

6.2.1 Type de recherche

Notre première comparaison concerne le type de recherche, i.e. la manière dont est créé l'arbre de recherche et dont on effectue les retours arrières en cas de conflit. Les types de recherche considérés, DFS et LDS, sont décrits en Section 2.2.3.

Une recherche LDS améliore significativement les performances de résolutions (présentées en Figure 6.1) pour les deux types d'algorithmes.

La LDS sert précisément à limiter l'impact d'une mauvaise décision en début de résolution. Or les premières décisions de branchement sont les moins fiables :

— En général les domaines sont plus grands et on a plus de chance de se tromper. Pour la propagation en particulier, les plus grands domaines correspondent à des distributions ayant de plus grandes entropies et plus d'incertitudes quant à la distribution des solutions.
— Le réseau de contraintes est également plus grand et comporte plus de cycles en début de recherche. Les assignations successives vont réduire sa taille et briser certains cycles, mais c'est au début de la recherche que les difficultés mentionnées en Section 2.3.3 se font le plus sentir, et les marginales calculées sont les plus approximatives.

6.2.2 Remise à l'uniforme des marginales

Au début de la résolution, toutes les variables ont une distribution marginale uniforme. On effectue la propagation jusqu'à la convergence ou un nombre maximal d'itérations et on se base sur les marginales obtenues pour prendre une décision de branchement.

Une fois l'assignation effectuée, doit-on repartir de marginales uniformes pour les variables ou réutiliser les marginales issues de la propagation effectuée précédemment ?

Ces deux options ont été évaluées empiriquement et l'évolution du score normalisé moyen est représentée en Figure 6.2. On constate que pour les deux types de messages Sum-Product et Max-Product, il est préférable de réutiliser les marginales (courbes désignées par "No Reset"), tant sur le plan de la pertinence des décisions de branchement (qu'on associe au score en fonction du nombre d'échecs) que de l'efficacité globale de la résolution (relative au temps écoulé). Ce mécanisme, bien qu'il ne soit pas utilisé par MiniCPBP dans un contexte de satisfaction, semble d'une importance considérable pour l'optimisation.

On émet les hypothèses suivantes quant aux raisons de cet impact positif de la réutilisation :

— La nouvelle distribution marginale à approximer après l'assignation présente des similitudes avec celle obtenue avant l'assignation. En réutilisant cette dernière, on peut espérer converger plus vite vers la nouvelle distribution. On note que les algorithmes de propagation commencent normalement avec des marginales uniformes. Partir d'une distribution différente pour ce qui est effectivement un nouveau réseau de contraintes (car il a été modifié par l'assignation) présente le risque de converger vers des marginales inexactes. Les résultats montrent toutefois que cette approche résulte en de

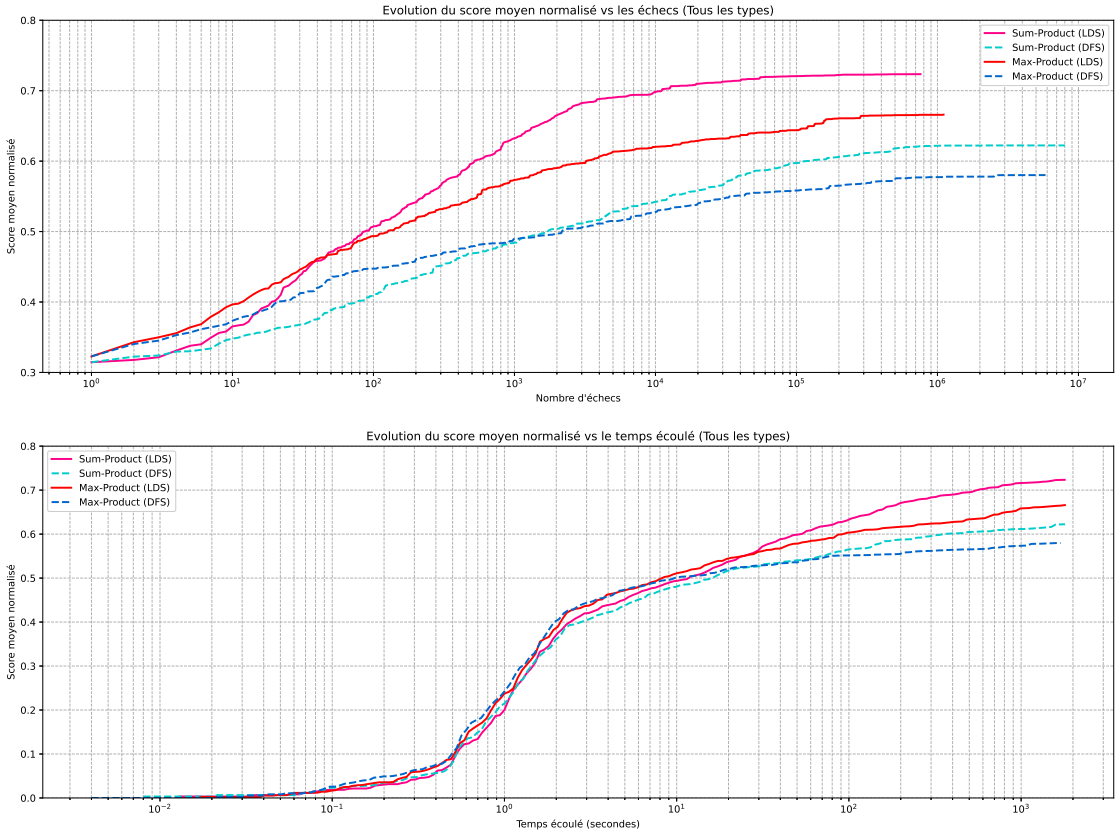

FIGURE 6.1 Évolution du Score Moyen Normalisé avec DFS et LDS

meilleurs scores.

— L'information de l'Oracle doit se propager de proche en proche dans le réseau : elle atteint d'abord la variable objectif, puis les variables connectées à l'objectif par une contrainte... Plusieurs itérations peuvent être nécessaires pour atteindre certaines variables de décision, condition nécessaire pour que ces dernières aient des marginales non uniformes pour le Max-Product, et également utile pour que le Sum-Product bénéficie du biais en faveur des meilleures solutions. Un retour trop fréquent à des variables uniformes peut empêcher cette propagation pour certains exemplaires.

— L'assignation peut également déconnecter certaines parties du graphe de contraintes de la variable objectif et de la contrainte Oracle. En effet, une variable x à laquelle on a assigné la valeur v n'envoie plus qu'un unique message $\mu_{x \to c}(v) = 1$. Si une variable n'est connectée à la variable objectif (et donc à l'Oracle), que par l'intermédiaire de variables déjà assignées, l'information fournie par l'Oracle ne l'atteindra pas. Le cas échéant, il est souhaitable de conserver les marginales : si elles étaient remises à l'uniforme, elles ne pourraient pas bénéficier de l'influence de l'Oracle et resteraient

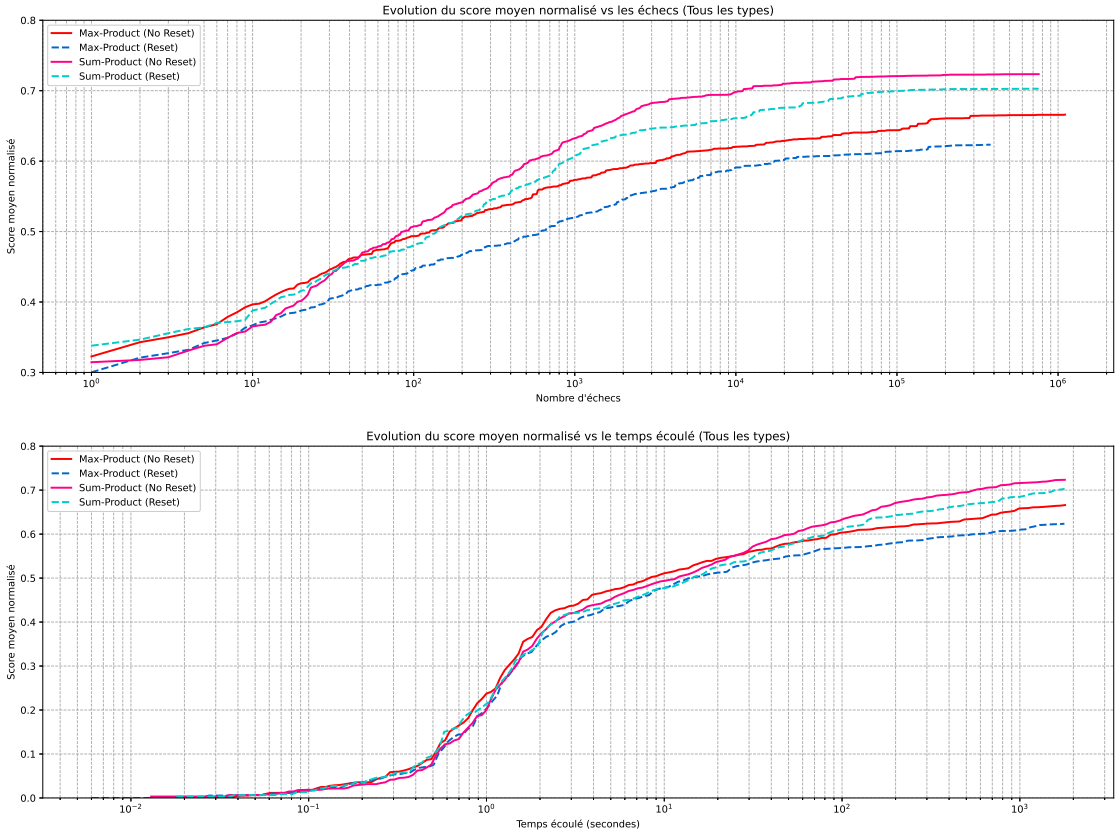

FIGURE 6.2 Évolution du Score Moyen Normalisé avec ou sans réutilisation des marginales

même uniformes dans le cas de Max-Product.

6.2.3 Critère d'arrêt de propagation

Dans un souci d'optimisation, on peut décider d'arrêter la propagation lorsque l'un des critères suivants est rempli (décrits également par l'Algorithme 8) :

— Une variable a une distribution avec une entropie très faible. On a donc une forte certitude sur la valeur à préférer pour cette variable, qui est alors une bonne candidate pour le branchement.

— La dernière itération de propagation n'a pas modifié l'entropie ou l'a très peu réduite, ce qui indique la convergence de l'algorithme.

Si l'on ignore ces considérations, le critère d'arrêt est simplement lorsqu'un nombre maximum d'itérations est atteint.

Sur la Figure 6.3, on voit que ces critères d'interruption anticipée de la propagation (courbes désignées par "Shortcut") améliorent le score à la fois vis-à-vis du nombre d'échecs et du

Algorithm 8: Critère d'arrêt de propagation

Data: Seuil minimal d'entropie $MIN\_VAR\_ENTROPY$, tolérance $ENTROPY\_TOLERANCE$
Result: Arrêt de la propagation des convictions
$iter \leftarrow 1$;
$previousEntropy \leftarrow +\infty$;
repeat
  `// Exécuter une itération de belief propagation`
  ExecuteBPIteration();
  `// Calculer l'entropie courante et minimale`
  $currentEntropy \leftarrow$ CalculerEntropieGlobale();
  $smallEntropy \leftarrow$ CalculerEntropieMinimale();
  `// Vérifier les critères d'arrêt`
  if $smallEntropy \leq MIN\_VAR\_ENTROPY$ **then**
    `// Variable avec forte certitude trouvée`
    break;
  end
  if $iter > 1$ ***and*** $currentEntropy = previousEntropy$ **then**
    `// Convergence détectée`
    break;
  end
  if $iter > 2$ ***and***
   $currentEntropy - previousEntropy > ENTROPY\_TOLERANCE$ **then**
    `// Divergence détectée`
    break;
  end
  $previousEntropy \leftarrow currentEntropy$;
  $iter \leftarrow iter + 1$;
until *critère d'arrêt atteint*;

temps écoulé.

L'amélioration du temps de résolution n'est pas surprenante : l'interruption de la propagation vise à éliminer des itérations superflues.

En revanche, on pourrait s'attendre à ce qu'un nombre moindre d'itérations donne des marginales moins précises et donc des échecs plus nombreux. On obtient le résultat contraire, qui peut s'expliquer de deux manières :

— Un grand nombre d'itérations peut faire apparaître des marginales trop basses (si une valeur doit converger vers zéro), qui peuvent introduire des problèmes de précision numérique, les marginales étant représentées en mémoire comme des `double`.
— Si une variable converge plus rapidement que les autres pour déclencher le critère

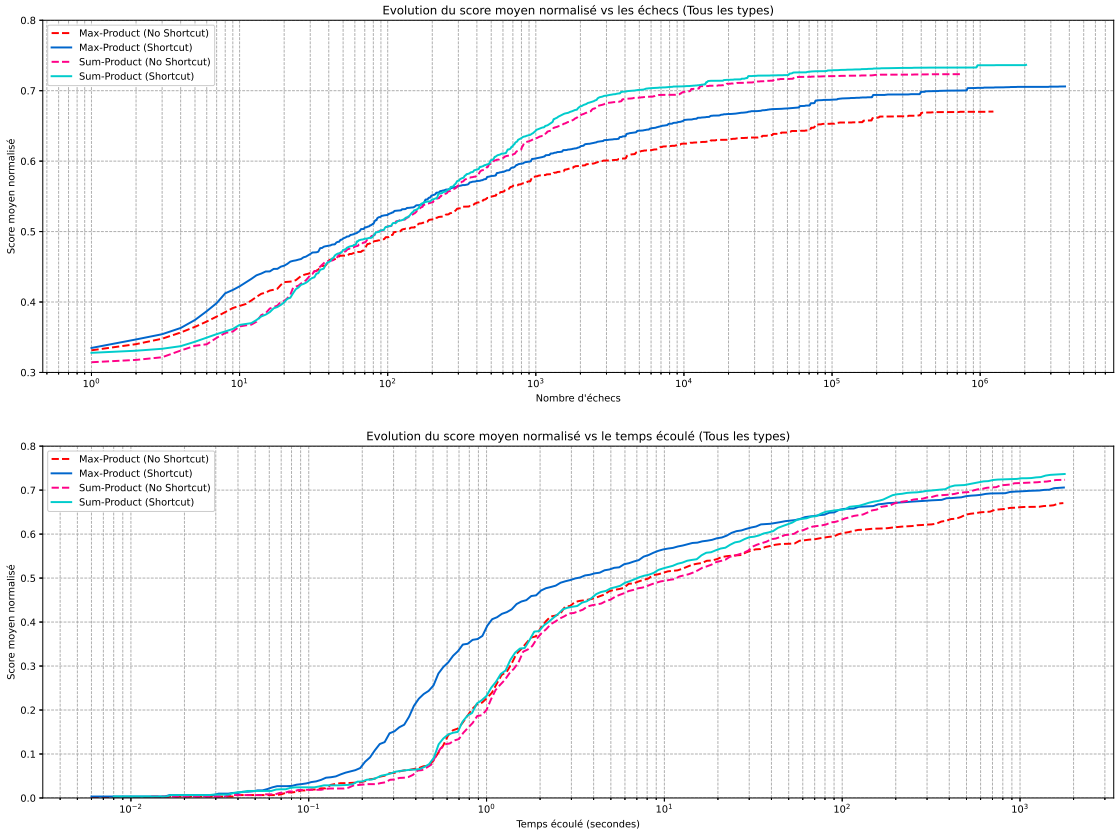

FIGURE 6.3 Évolution du Score Moyen Normalisé avec ou sans interruption de la propagation

d'arrêt, il est possible qu'elle soit une meilleure candidate pour le branchement que d'autres variables qui auraient atteint des distributions similaires en un nombre plus grand d'itérations. L'introduction du critère d'arrêt permet cette discrimination pour le branchement.

Introduire ce critère d'interruption anticipé pour la propagation ne semble pas présenter de désavantage. Toutes les résolutions présentées dans le reste du chapitre utilisent donc ce mécanisme.

6.2.4 Le nombre maximal d'itérations

Il existe également un critère d'arrêt de la propagation basé sur le nombre maximal d'itérations à effectuer à chaque nœud de la recherche.

L'influence de ce critère d'arrêt sur les performances de Max-Product est décrite en Figure 6.4. Configuré avec 20 itérations l'algorithme maintient des performances similaires en termes de temps, en comparaison avec les limites plus basses. Cela suggère qu'il converge

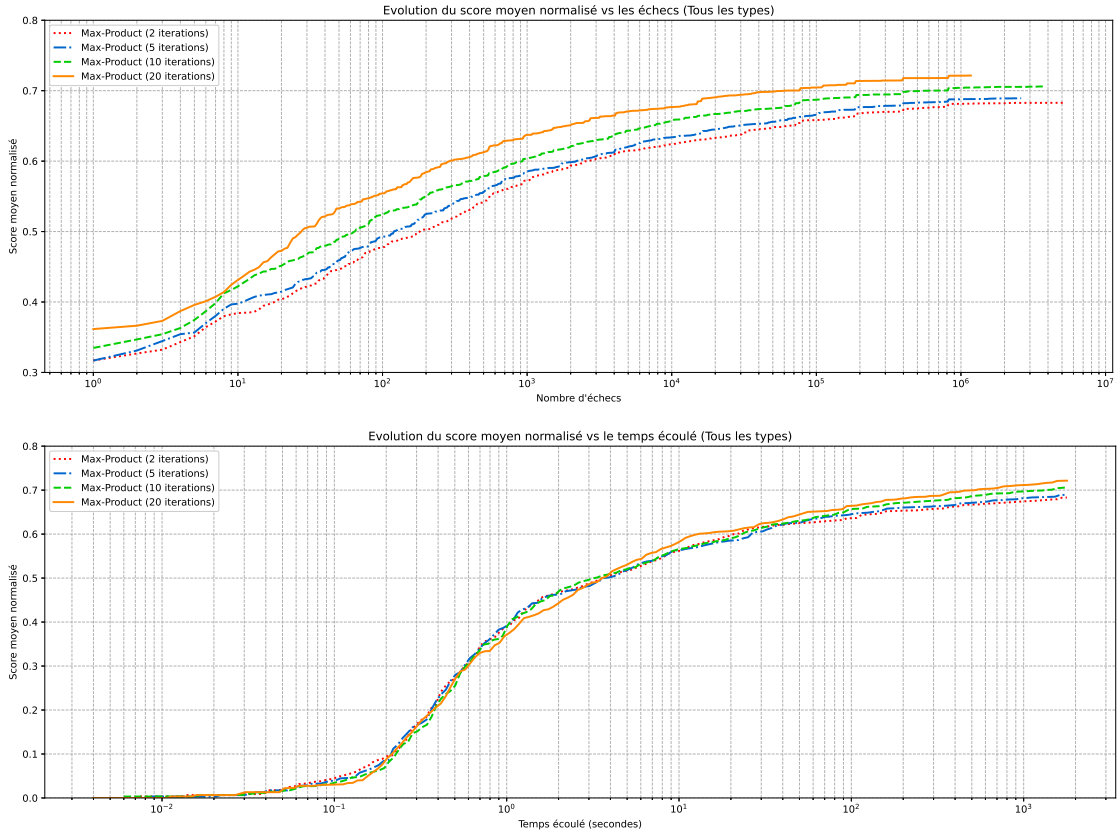

FIGURE 6.4 Influence du nombre maximal d'itérations propagation sur l'évolution du Score Moyen Normalisé avec Max-Product

en réalité en moins d'itérations grâce aux critères d'arrêt basés sur l'entropie, évitant ainsi le surcoût computationnel. Le guidage bénéficie cependant de la possibilité d'un plus grand nombre d'itérations, comme le montre l'augmentation du score plus rapide en fonction du nombre d'échecs. Cela est confirmé en Figure 6.5 qui montre l'influence du nombre maximal d'itérations en l'absence du critère d'arrêt : le score augmente moins rapidement lorsque le nombre d'itérations augmente.

Les performances de Sum-Product sont représentées en Figure 6.6. On constate des scores plus hauts en début de résolution avec une limite de 2 itérations. Quand la résolution progresse, le score est plus élevé avec une limite de 10.

6.2.5 Le poids de l'Oracle

La Section 4.1.2 a mis en évidence que le poids de l'Oracle peut influencer la vitesse de convergence ou si l'algorithme va converger vers les marginales désirées. On évalue différents

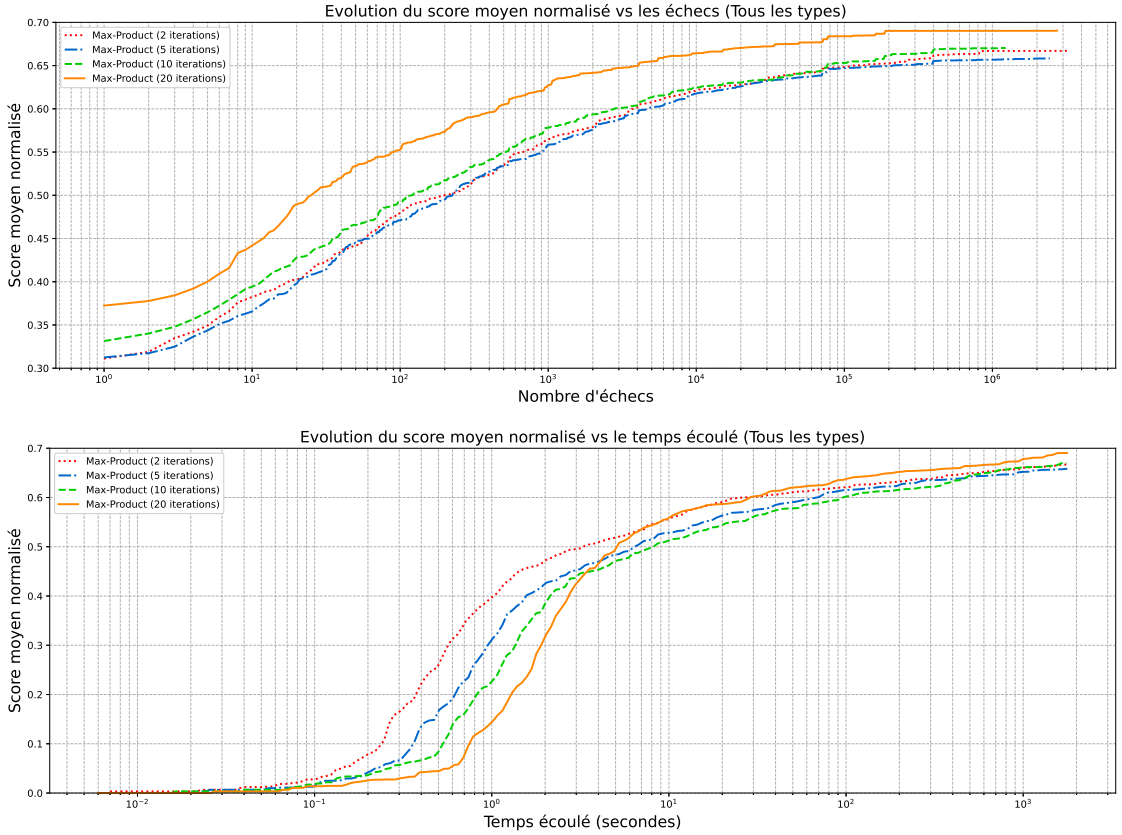

FIGURE 6.5 Influence du nombre maximal d'itérations propagation sur l'évolution du Score Moyen Normalisé avec Max-Product en l'absence de critère d'arrêt anticipé : la rapidité de résolution souffre du nombre plus élevé d'itération

poids d'Oracles pour les algorithmes Max-Product et Sum-Product.

Pour Max-Product, présenté Figure 6.7, un Oracle de poids plus élevé semble légèrement préférable, mais l'influence de ce paramètre semble minime.

Pour Sum-Product (Figure 6.8), on compare également avec l'absence de l'Oracle (i.e. un poids de 0). La présence d'un Oracle augmente le score significativement pour les petits nombres d'échecs. L'algorithme sans Oracle atteint toutefois des scores similaires, bien que légèrement inférieurs, à partir de seulement 30 échecs. La valeur exacte du poids ne semble pas avoir d'influence.

En conclusion, l'impact du poids de l'Oracle semble minime dans le cadre de nos expérimentations. Le problème de convergence lié à un mauvais poids d'Oracle mis en évidence à la Section 4.1.2 ne semble pas être un problème récurrent avec les poids et les exemplaires utilisés ici. On conservera un Oracle de poids 1 dans les autres résolutions.

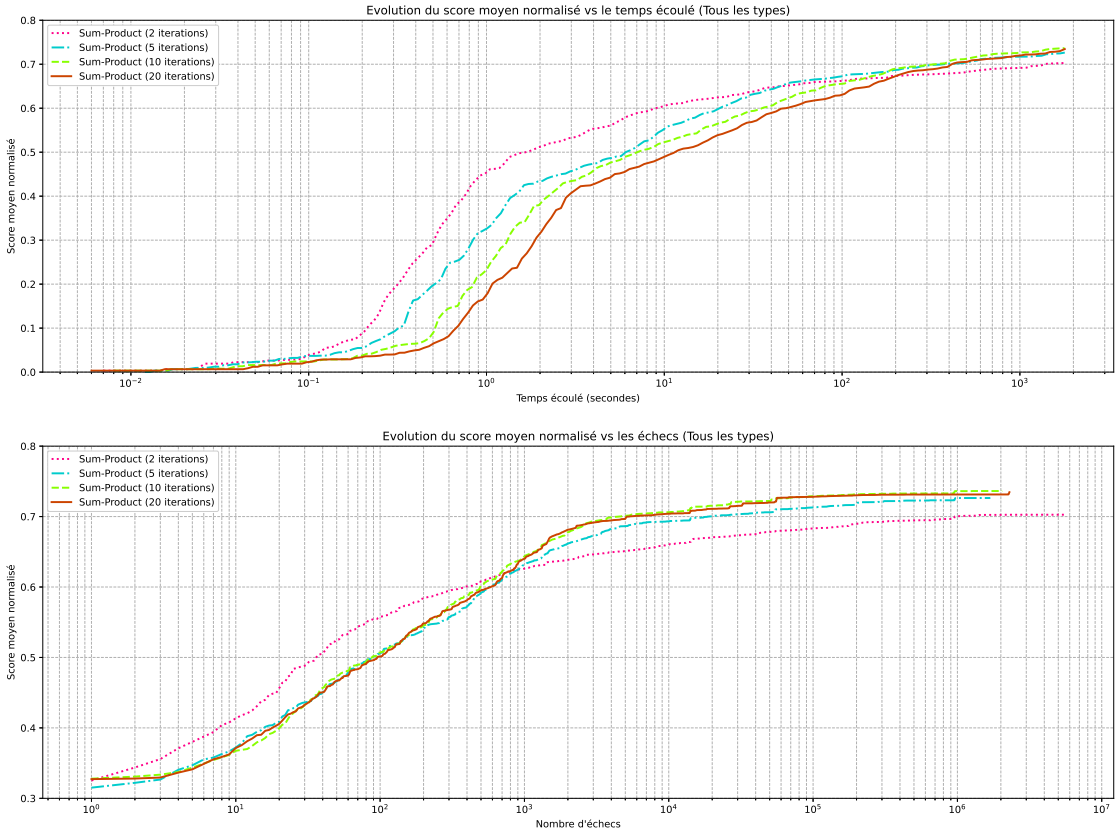

FIGURE 6.6 Influence du nombre maximal d'itérations propagation sur l'évolution du Score Moyen Normalisé avec Sum-Product

6.2.6 DomWDeg si l'entropie est trop élevée

Le critère de l'entropie permet de sélectionner une variable pour laquelle la propagation a produit une forte certitude quant à la variable à choisir. En revanche, dans le cas où toutes les distributions marginales ont des entropies élevées, on va préférer sélectionner la variable sur la base de l'heuristique *DomWDeg* plutôt que des distributions marginales.

Au moment de sélectionner une variable selon sa marginale, on va calculer l'entropie normalisée de sa distribution marginale et si celle-ci dépasse un certain seuil, on va revenir sur ce choix et sélectionner la variable selon *DomWDeg*.

Le choix de la valeur n'est pas inclus dans *DomWDeg*. On va donc se baser sur les marginales et choisir la valeur ayant la probabilité la plus forte.

On évalue ici l'impact du choix du seuil d'entropie au-dessus duquel on va utiliser *DomWDeg*. Si le seuil est t, on utilisera *DomWDeg* si la valeur de l'entropie vérifie $H(x) \geq t$.

L'évolution du score pour Max-Product est présentée à la Figure 6.9.

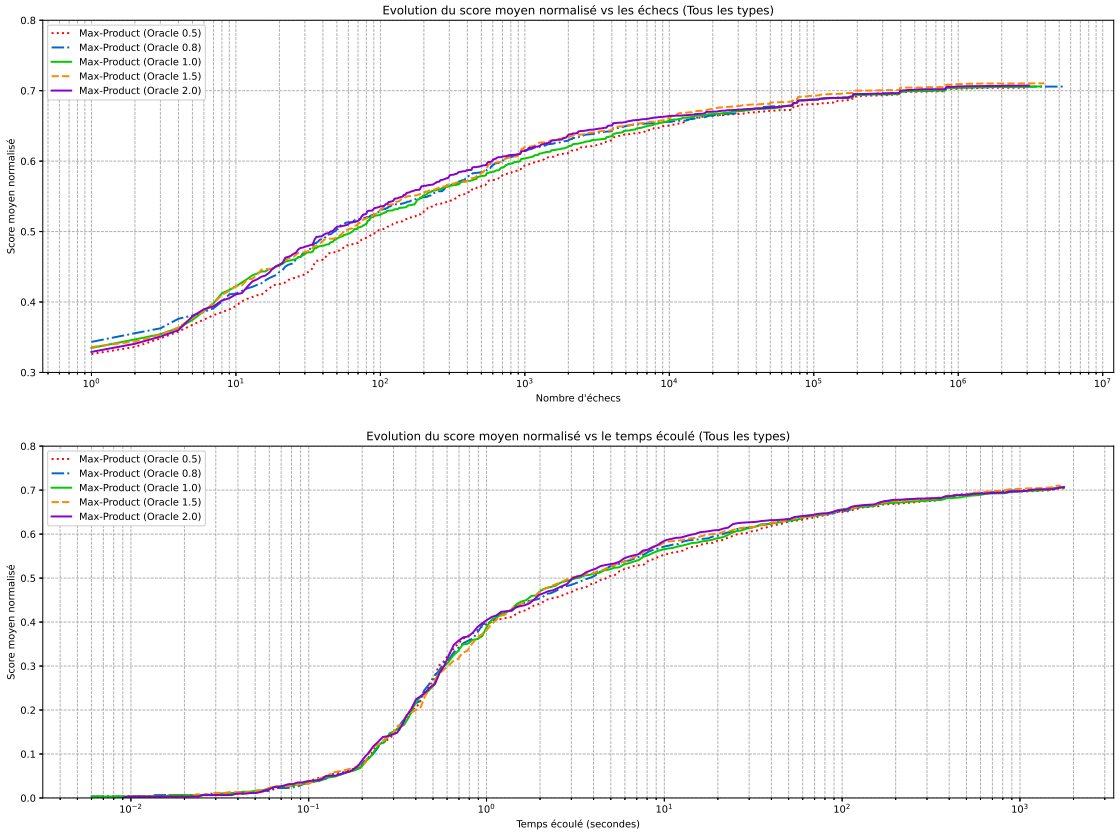

FIGURE 6.7 Influence du poids de l'Oracle sur l'évolution du Score Moyen Normalisé avec Max-Product

— Pendant la première seconde de résolution, tous les seuils ont des résultats équivalents.
— Sur l'intervalle de temps de 1 à 10 secondes, *Min Entropy* montre une légère supériorité sur *DomWDeg + Max Marginal*. Le choix exact du seuil semble peu important : quand Max-Product converge correctement, on a vu qu'il produisait des distributions de basse entropie comme celles présentées dans le Tableau 4.1, à moins qu'il n'existe de nombreuses solutions optimales avec des assignations variées.
— Après une dizaine de secondes, *DomWDeg* va atteindre des scores supérieurs.

L'influence du seuil sur l'évolution du score pour Sum-Product est montrée à la Figure 6.10.

— Les seuils de 0.1, 0.2, 0.5 ont des performances équivalentes du point de vue du temps. Le score est cependant plus élevé en dessous de 10 échecs pour le seuil de 0.5.
— L'usage systématique de *DomWDeg* donne des scores semblables pendant les 10 premières secondes, et dispose ensuite d'un léger avantage.

On observe toutefois que cette fois, un seuil bas de 0.5 augmente significativement les performances. On en déduit que Sum-Product produit des distributions d'entropies variées et que

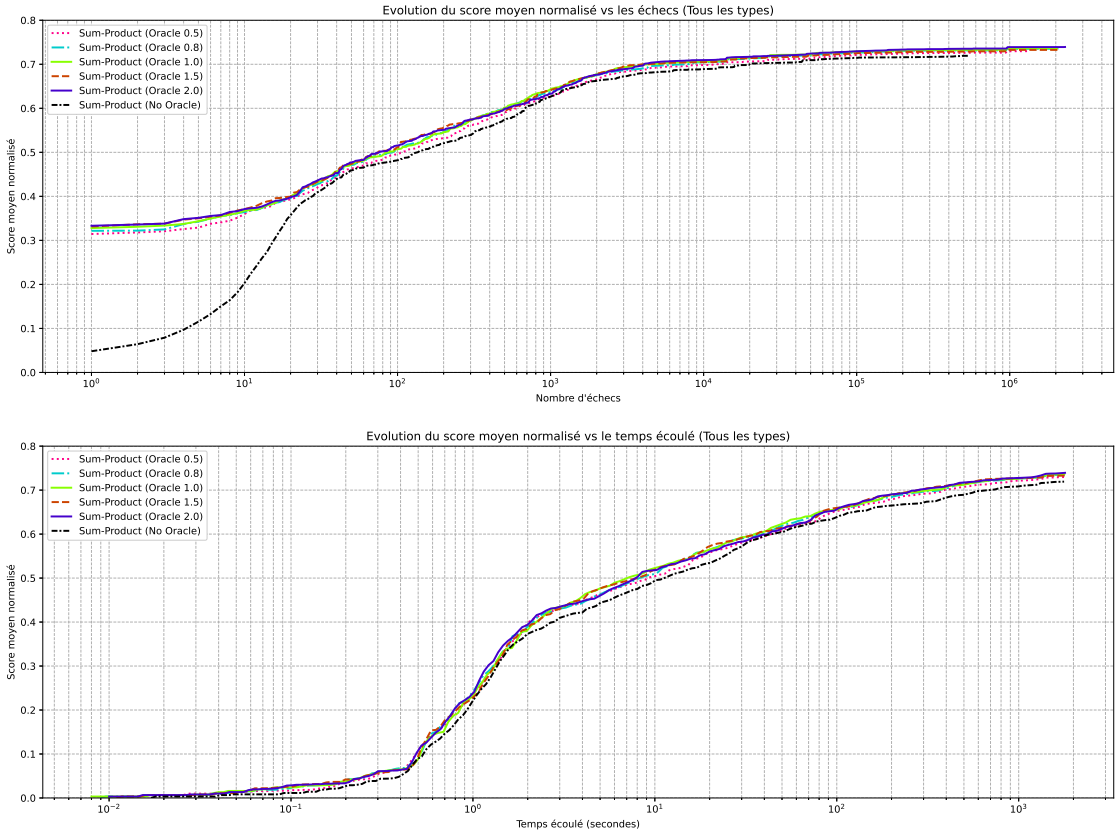

FIGURE 6.8 Influence du poids de l'Oracle sur l'évolution du Score Moyen Normalisé avec Sum-Product

le seuil est utile pour se focaliser sur celles apportant le plus de certitudes.

Un seuil de 1.0, i.e. n'utilisant *DomWDeg* que si la marginale est exactement uniforme, n'a pas d'impact, et les performances sont sensiblement les mêmes qu'en l'absence du mécanisme. Avec un tel seuil, l'usage de *DomWDeg* n'est presque jamais déclenché : non seulement il est rare que les distributions calculées soient uniformes, mais le test d'égalité $H \geq 1.0$ peut aussi être négatif du fait de la précision numérique limitée.

6.2.7 Heuristique de branchement

La Section 2.3.5 a introduit plusieurs heuristiques pour exploiter les marginales résultant de la propagation. Celles-ci se basent sur différentes métriques relatives aux distributions : la marginale maximale, le regret, la force, l'entropie (normalisée ou non).

Avant de nous pencher sur les performances empiriques de ces heuristiques, comparons brièvement les métriques qui leur sont associées. On a généré de nombreuses distributions pos-

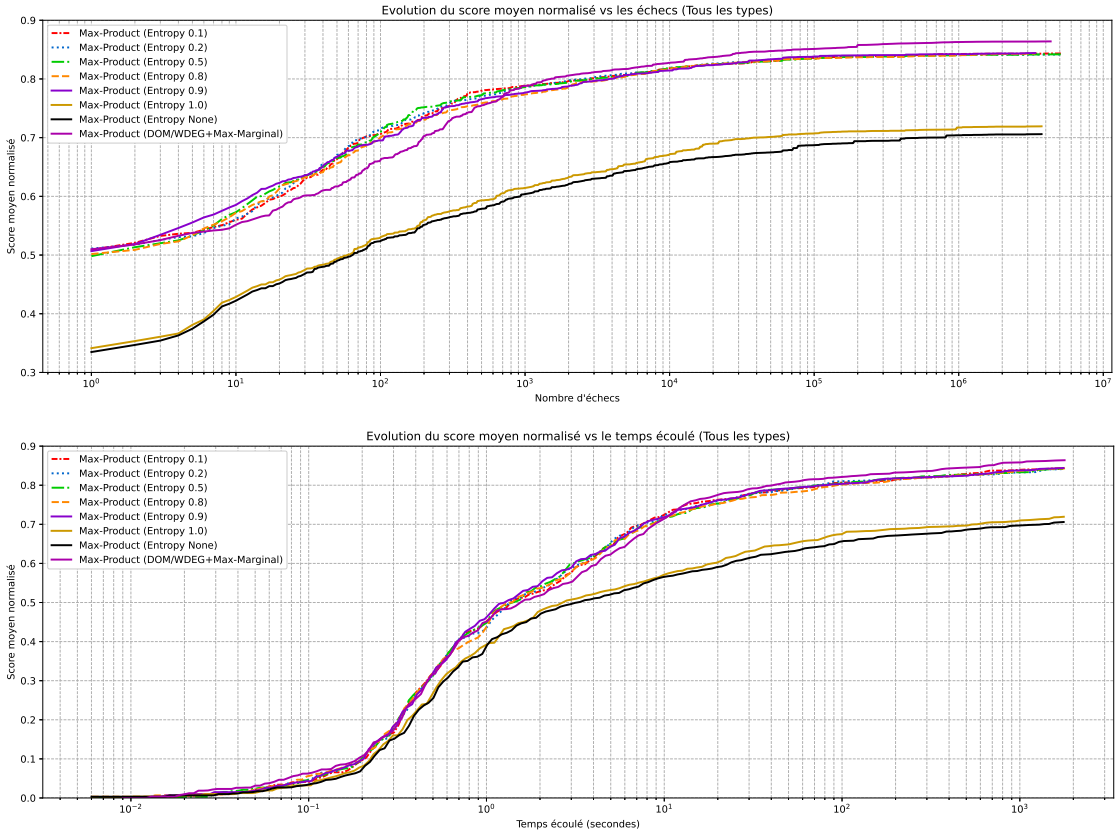

FIGURE 6.9 Influence du seuil d'entropie normalisée au-dessus duquel on va sélectionner la variable avec *DomWDeg* (Max-Product)

sédant des niveaux variés d'entropie et représenté chacune des autres métriques en fonction de l'entropie normalisée à la Figure 6.11.

On constate une forte corrélation entre une entropie basse et un poids, force ou regret élevé. Cependant, on observe également une forte variance, qui justifie d'évaluer les performances de chaque métrique et heuristique associée.

On effectue cette comparaison en Figure 6.12 pour Max-Product et en Figure 6.13 pour Sum-Product. Les résultats sont similaires pour les deux algorithmes : l'heuristique dominante est *DomWDeg-Max Marginal* qui sélectionne la variable via *DomWDeg* et la valeur ayant la plus haute marginale. Les heuristiques *Max Marginal* et *Min Entropy* sont les deuxième et troisième meilleures heuristiques et ont des performances équivalentes.

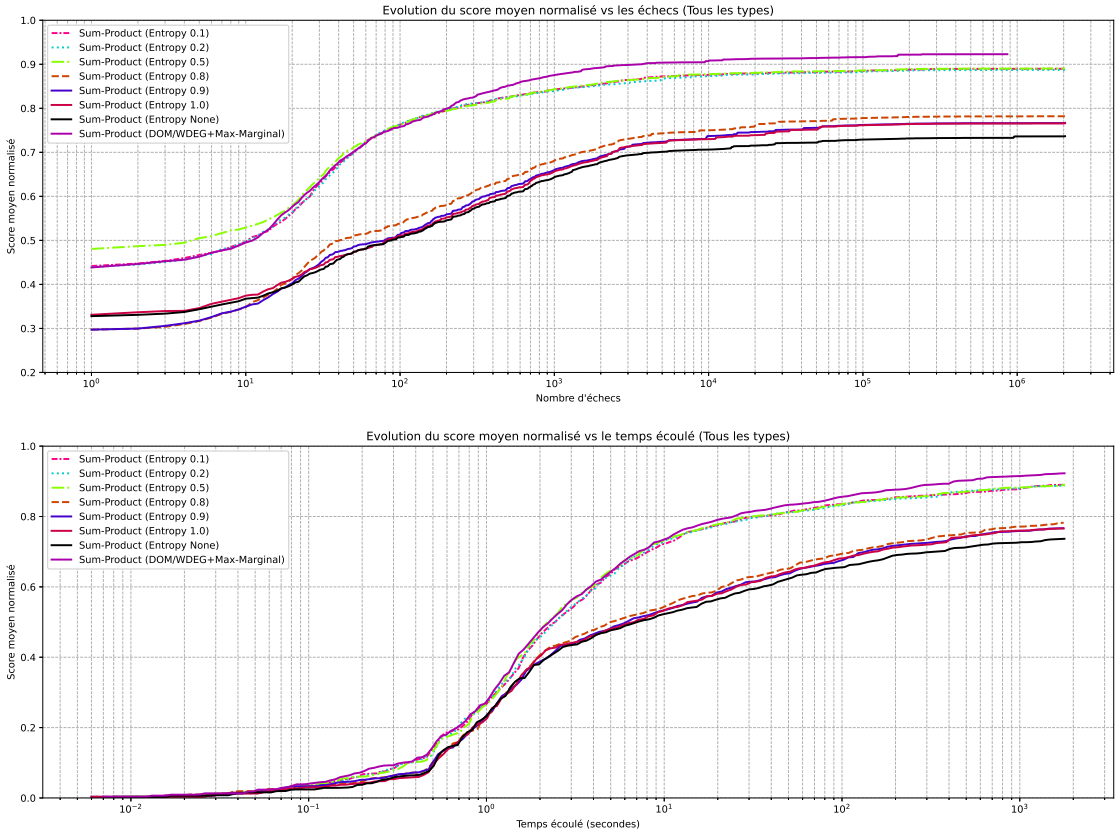

FIGURE 6.10 Influence du seuil d'entropie normalisée au-dessus duquel on va sélectionner la variable avec *DomWDeg* (Sum-Product)

6.2.8 Comparaison des meilleures configuration

Sélection des meilleurs paramètres

Sur la base des évaluations précédentes, nous avons sélectionné un ensemble de configurations représentatives intégrant les meilleurs paramètres identifiés :

- **Paramètres communs optimaux** : recherche LDS, réutilisation des marginales, interruption anticipée de la propagation en fonction de l'entropie, et poids d'Oracle de 1.
- **Sum-Product avec DomWDeg + Max Marginal** : l'heuristique dominante identifiée, évaluée avec 2 et 10 itérations, configurations qui dominent respectivement au début et à la fin de la résolution.
- **Sum-Product avec Min-Entropy** : conservée comme représentante des heuristiques de sélection de variable basées sur la propagation. Elle est utilisée quand l'entropie normalisée est sous un seuil de 0.5, ce qui semble optimal. Le nombre maximal d'ité-

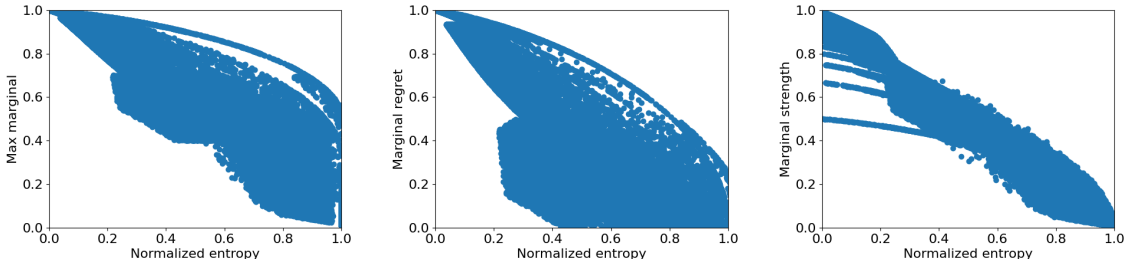

FIGURE 6.11 Corrélation des métriques sur les distributions marginales et de leur entropie normalisée. De gauche à droite : marginale max, regret, force

rations est 2.

— **Max-Product avec DomWdeg + Max Marginal** avec un maximum de 20 itérations.

— **Max-Product avec Min Entropy** avec 20 itérations au maximum, et un seuil d'entropie de 0.9 pour avoir recours à *DomWDeg*.

— **Configuration de référence** : résolution sans passage de messages et utilisant *DomWDeg + Min Value* pour établir une base de comparaison.

Performances sur l'ensemble du corpus

Les résultats présentés en Figure 6.14 confirment la pertinence de la propagation pour l'optimisation. Les configurations utilisant la propagation améliorent le score moyen d'environ 15% par rapport à la résolution sans propagation à partir de 10 secondes de résolution.

L'analyse révèle des profils de performance distincts selon l'horizon temporel :

— **Résolutions courtes** (moins d'une minute) : Sum-Product avec un faible nombre d'itérations (2) et Max-Product montrent les meilleures performances initiales, convergeant rapidement vers des solutions de qualité.

— **Résolutions longues** : Sum-Product avec 10 itérations démontre son avantage sur les résolutions longues, probablement en raison de sa capacité à guider efficacement la recherche lorsque les solutions satisfaisantes se raréfient et que la satisfaction devient un enjeu majeur.

Sum-Product semble un choix préférable à Max-Product, à condition de choisir une configuration adaptée à la durée de résolution pour laquelle on veut optimiser. Max-Product pourrait être un compromis dans le cas où les besoins de la résolution sont ambigus de ce point de vue.

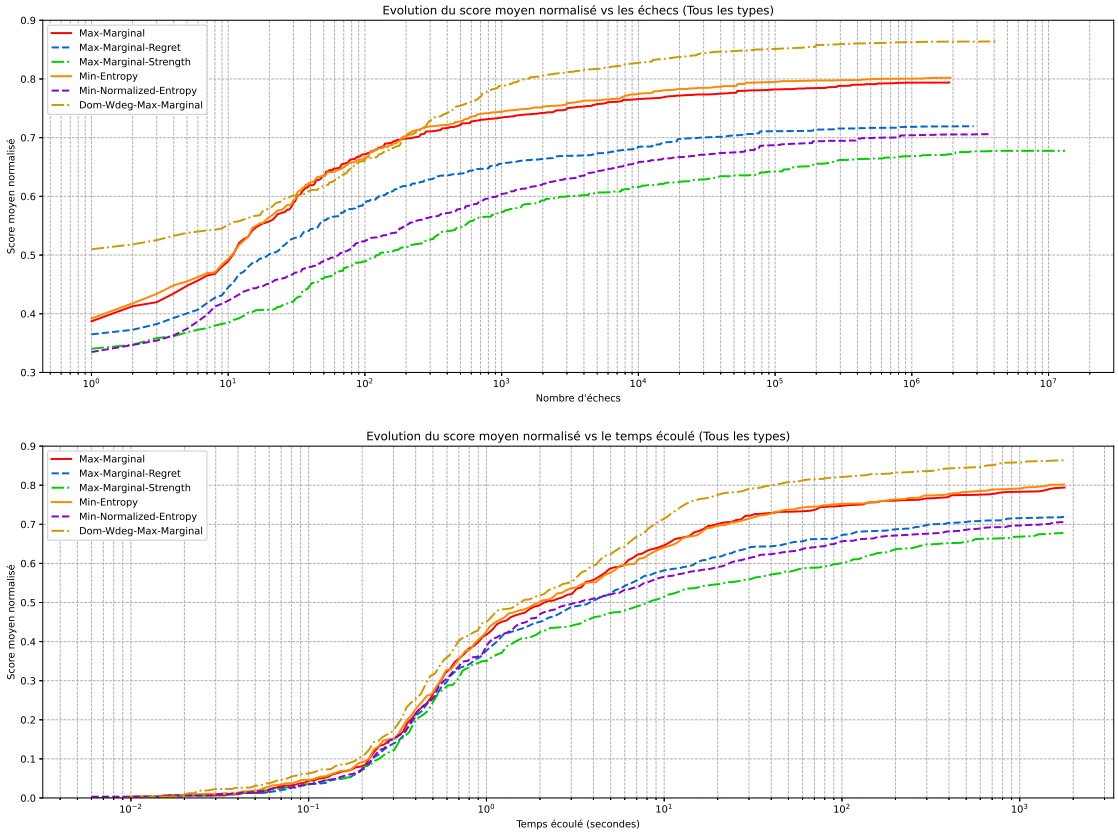

FIGURE 6.12 Sélection d'une heuristique de branchement pour Max-Product

On note aussi une différence de maturité entre les implémentations. Sum-Product a bénéficié d'optimisations sur quelques années, tandis que Max-Product a été introduit dans le solveur dans le cadre de ce mémoire. Des optimisations pourraient rendre Max-Product plus compétitif à l'avenir. À titre d'exemple, si une contrainte a établi la cohérence d'arc généralisée et que les variables impliquées sont toutes uniformes, la contrainte pourrait envoyer immédiatement des messages uniformes (voir Section 2.3.2).

Performances sur différents problèmes

On considère en Figure 6.15, pour chacun des problèmes, l'évolution du score des résolutions Sum-Product, Max-Product et sans passage de messages.

On constate une grande disparité des performances en fonction du problème traité. Pour les trois approches, certains problèmes sont bien résolus (*Drinking*, *Knapsack*, *LowAutocorrelation*) tandis que d'autres (*Litpuzzle*, *MaximumDensityOscillatingLife*) présentent une grande difficulté. On considérera cette difficulté intrinsèque aux exemplaires associés aux problèmes,

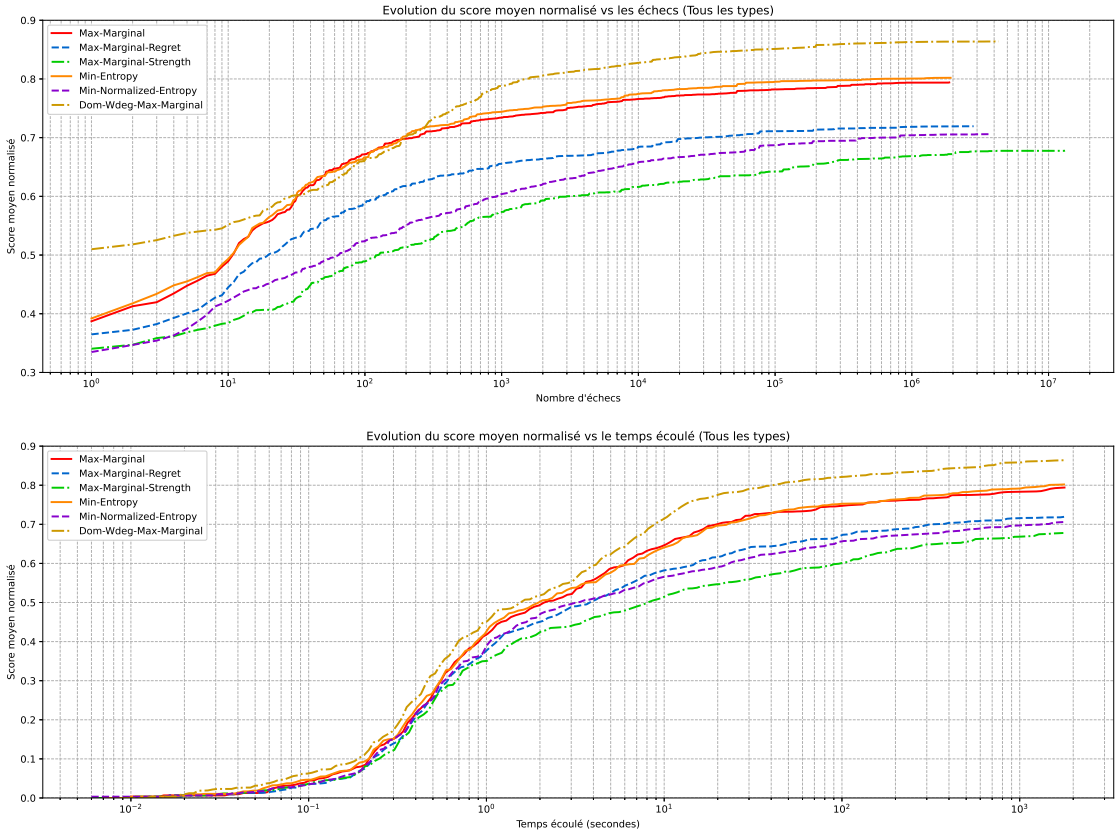

FIGURE 6.13 Sélection d'une heuristique de branchement pour Sum-Product

puisque les performances des trois approches sont similaires.

En revanche, *Knapsack* et *LowAutocorrelation* sont résolus plus rapidement que *Drinking* avec les approches de passage de messages, tandis qu'en l'absence de passage de messages, c'est l'inverse qui se produit.

Une possible explication est la différence dans la taille des portées des contraintes impliquées. Une contrainte de grande arité peut calculer une information locale couvrant une part plus grande du réseau et de l'ensemble des solutions, ces messages seront les plus pertinents. On considère le rapport moyen entre la taille de la portée de chaque variable et le nombre de variables de décision (i.e. celles que l'on souhaite fixer, par opposition aux variables intermédiaires d'un modèle). Si l'on note $X_d \subset X$ l'ensemble des variables de décision, on peut définir le rapport comme :

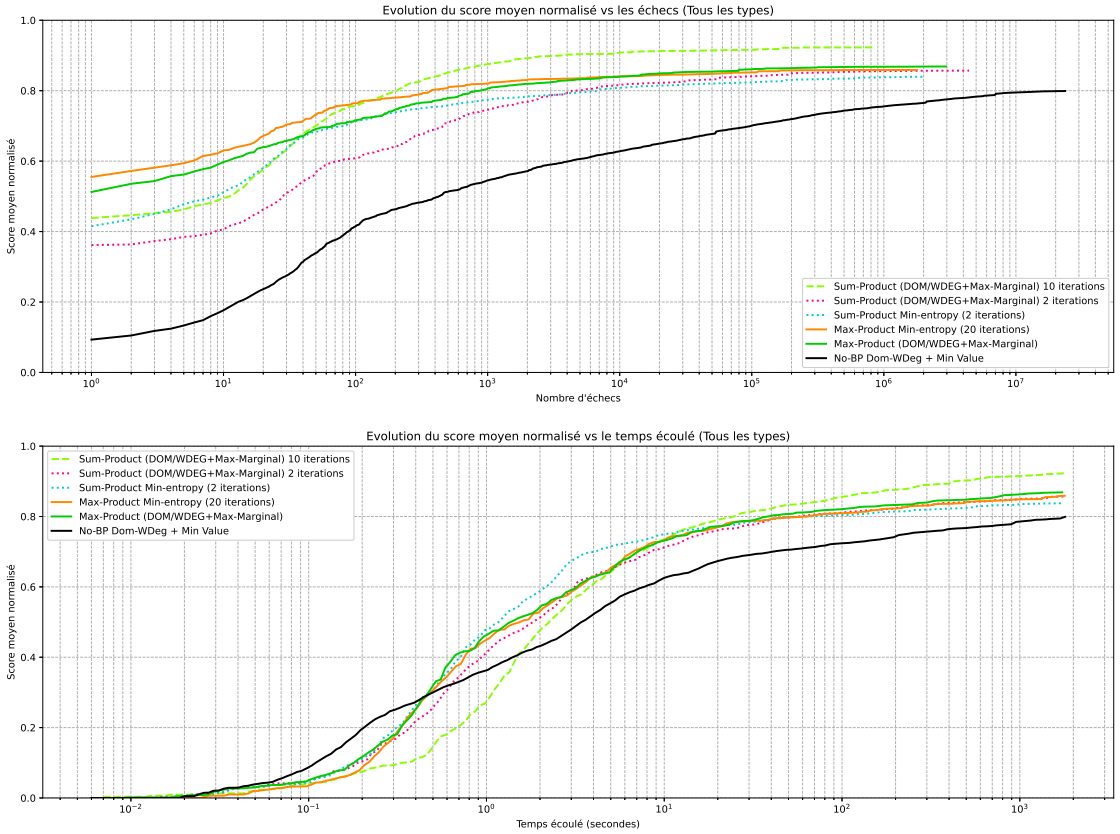

FIGURE 6.14 Évolution du Score Moyen Normalisé pour les configurations sélectionnées

$$rapportPortee(c, X_d) = \frac{|S(c)|}{|X_d|}$$

$$rapportPorteeMoyen(X_d, C) = \frac{\sum_{c \in C} scopeRatio(c, X_d)}{|C|}$$

Puisqu'il existe des variables intermédiaires impliquées dans les contraintes, ce rapport peut être supérieur à 1.

On présente la moyenne de ces rapports sur les exemplaires de ces trois types de problèmes dans le Tableau 6.1. *Knapsack* et *LowAutocorrelation* possèdent des rapports plus élevés et donc des contraintes ayant des portées de tailles supérieures. Ceci pourrait expliquer les meilleures performances de la propagation sur ces deux types de problèmes.

Tableau 6.1 Rapport de la portée moyenne sur l'ensemble des exemplaires des problèmes *Drinking*, *Knapsack*, *LowAutocorrelation*

Problème	**Rapport de portée moyen**
Drinking	0.030
Knapsack	0.667
LowAutocorrelation	2.511

6.3 Conclusion

6.3.1 Impact des paramètres

Cette évaluation empirique a permis d'identifier l'impact de chaque paramètre sur l'efficacité des heuristiques basées sur la propagation pour l'optimisation.

Gestion des marginales : la réutilisation des marginales entre les phases de propagation améliore significativement les performances. Cette approche préserve l'information de l'Oracle propagée dans le réseau et maintient la connectivité aux variables déconnectées, compensant les risques théoriques de convergence vers des marginales inexactes.

Optimisation de la propagation : l'interruption anticipée de la propagation basée sur l'entropie des variables et la convergence améliore à la fois l'efficacité temporelle et la qualité des décisions de branchement, en évitant les itérations superflues et les problèmes de précision numérique.

Stratégie de recherche : la recherche à divergence limitée (LDS) surpasse systématiquement la recherche en profondeur (DFS) en limitant l'impact des décisions précoces, particulièrement critiques et sujettes à l'erreur car les domaines sont alors larges et les marginales approximatives. L'usage de *DomWDeg* suggérerait toutefois d'expérimenter avec l'introduction de relances pour exploiter l'apprentissage des poids.

Nombre maximal d'itérations : Pour Max-Product, une limite élevée (20 itérations) améliore la qualité du guidage sans surcoût temporel grâce aux critères d'arrêt anticipé. Pour Sum-Product, un faible nombre d'itérations (ici, 2) favorise les performances initiales, tandis qu'une limite plus élevée devient avantageuse sur les résolutions prolongées.

Paramétrage de l'Oracle : un poids d'Oracle de 1 est adéquat pour les deux algorithmes. Pour Sum-Product, la présence de l'Oracle améliore drastiquement les performances initiales, permettant d'atteindre des scores plus élevés avec moins d'échecs. Pour Max-Product, l'Oracle est indispensable pour obtenir une information au-delà de la cohérence d'arc géné-

ralisée.

Seuils d'entropie : le mécanisme de basculement vers *DomWDeg* améliore les performances, ce qui est à prévoir dans la mesure où *DomWDeg* est l'heuristique la plus performante. Le choix du seuil est variable selon l'algorithme. Pour Sum-Product, un seuil de 0.5 améliore les performances en exploitant les distributions à forte certitude. Pour Max-Product, l'impact du choix du seuil reste marginal car l'algorithme produit naturellement des distributions de basse entropie.

Heuristiques de branchement : *DomWDeg + Max Marginal* émerge comme l'heuristique dominante, combinant la robustesse de *DomWDeg* pour la sélection de variable et l'information des marginales pour le choix de valeur. *Min Entropy* constitue la meilleure alternative purement basée sur la propagation.

6.3.2 L'approche de passage de messages en optimisation

Les heuristiques basées sur le passage de message apportent un gain substantiel de performance dans la résolution de problèmes d'optimisation par rapport à une heuristique *DomWDeg* représentant l'état de l'art en l'absence de passage de message. Les paramètres pour la résolution sont nombreux, mais on a montré qu'une configuration adaptée peut exploiter efficacement l'information probabiliste pour guider la recherche.

Différentes approches propagation sont à préférer si l'on souhaite arrêter la résolution rapidement ou lui accorder beaucoup de temps : Sum-Product avec un nombre réduit d'itérations optimise pour un score plus élevé sur les résolutions courtes, tandis qu'un nombre plus élevé d'itérations finira par produire de meilleures solutions sur les résolutions plus longues. Dans le futur, on pourra considérer un nombre d'itérations changeant dynamiquement au cours de la résolution pour tirer profit de ces deux profils de performance.

L'algorithme Sum-Product, associé à un choix judicieux de paramètres, semble plus performant que Max-Product, mais des perspectives sont ouvertes pour rendre Max-Product plus compétitif.

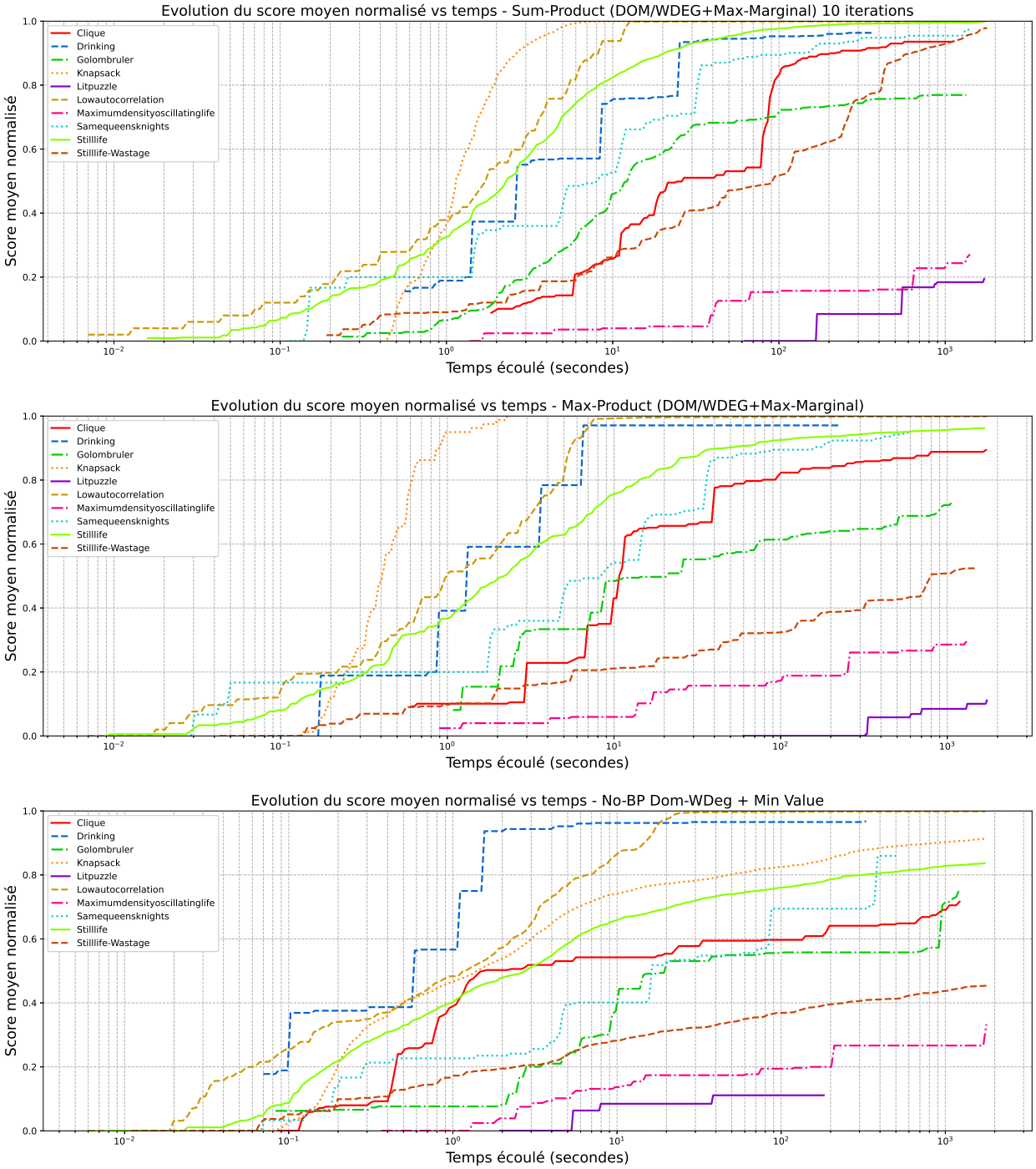

FIGURE 6.15 Évolution du Score Moyen Normalisé en fonction du temps pour chacun des problèmes

CHAPITRE 7 CONCLUSION

Ce mémoire a permis d'étendre l'applicabilité du passage de messages, une méthode d'inférence avancée complémentaire au filtrage dans la CP, de la résolution de CSPs à celle des COPs.

7.1 Synthèse des travaux

Pour la résolution de COPs, nous avons introduit l'algorithme de passage de messages Max-Product dans le solveur MiniCPBP, dans l'espoir que les marginales résultantes désignent les valeurs participant aux solutions optimales.

Afin de représenter la fonction objectif au niveau des messages et marginales, nous avons proposé l'usage d'une contrainte Oracle sur la variable objectif. Dans un problème de maximisation, celle-ci envoie un message proportionnel à chaque valeur pour biaiser les distributions en faveur des solutions de score plus élevé. Le principe peut-être adapté aux problèmes de minimisation.

La présence de la contrainte Oracle s'avère avoir une influence positive sur les scores obtenus par la recherche guidée par Sum-Product. Elle est même indispensable lorsque Max-Product est utilisé, afin de produire des marginales présentant un intérêt par rapport à la cohérence d'arc généralisée.

De nouveaux algorithmes sont présentés pour le calcul des messages Max-Product pour chacune des contraintes AllDifferent, Table, Sum et Regular.

Le calcul d'un message Max-Product pour AllDifferent se réduit au calcul d'un couplage de poids minimal. Ce problème est dans la classe de complexité P, tandis que le message Sum-Product correspond au permanent d'une matrice d'adjacence, problème appartenant à $\#P$. Malgré cette complexité théorique plus haute, le calcul du Sum-Product comporte plusieurs optimisations qui le rendent plus rapide que celui du Max-Product. Nous avons donc conçu un algorithme Max-Product optimisé réutilisant des résultats intermédiaires entre les messages d'une même variable. Cet algorithme basé sur la construction de plus courts chemins augmentants a permis d'être compétitif avec Sum-Product.

Les messages des contraintes Table, Sum et Regular pour Max-Product s'appuient sur les mêmes algorithmes et structures de données que leurs homologues Sum-Product. L'accumulation par somme y est toutefois remplacée par un maximum.

Une évaluation empirique sur un ensemble d'exemplaires issus du corpus XCSP a comparé l'évolution du score normalisé moyen pour un grand nombre de résolutions. Cela a permis d'identifier des options avantageuses pour les nombreux paramètres de la résolution CPBP : nombre d'itérations, poids de l'Oracle, heuristiques de branchement...

On a constaté qu'une sélection de variable avec l'heuristique *DomWDeg* complétée par une sélection de valeur basée sur les marginales était la plus efficace parmi les approches évaluées.

La CPBP basée sur le Sum-Product et un nombre d'itérations bas (ici, 2) obtient de meilleurs scores pour des résolutions courtes. Pour les résolutions plus longues, le Sum-Product voit son score monter plus haut si on lui accorde un nombre élevé d'itérations (20, dans notre cas). Max-Product, s'il est dominé par ces deux dernières configurations, peut représenter un compromis intéressant si le temps alloué à la résolution n'est pas décidé d'avance, ou un ajout dans un portfolio d'algorithme.

L'ensemble des résolutions basées sur la BPBP présente de meilleures performances que celui des résolutions avec l'heuristique de référence *DomWDeg + Min Value*. Le score normalisé moyen est environ 15% plus haut pendant la résolution. Ceci met en évidence l'intérêt de la CPBP pour la résolution de COPs, intérêt qui n'avait été montré que pour les CSPs jusqu'ici. L'approche conserve également les avantages de la CP : la génération de solutions intermédiaires tout au long de la résolution, ainsi que la preuve d'optimalité si l'on accorde assez de temps pour une recherche exhaustive.

7.2 Limitations de la solution proposée

Des limitations et difficultés persistent pour appliquer la CPBP à la résolution de COPs.

Le passage de messages est un processus complexe qui présente des comportements qui sont parfois surprenants. Il est difficile de garantir la bonne convergence des marginales quand le réseau de contraintes comporte des cycles, ce qui est le cas dans l'écrasante majorité des applications. Par exemple, l'influence du poids de l'Oracle sur la convergence aurait besoin d'être explorée.

L'approche a aussi des performances très variables selon le type d'exemplaire abordé (*Knapsack*, *Drinking*...). La topologie du réseau de contraintes semble jouer un rôle important et si un exemple à été donné dans ce mémoire, son rôle mérite d'être étudié systématiquement.

Le coût computationnel du calcul des messages reste un obstacle à l'applicabilité plus large de la CPBP. Le gain en efficacité pour le guidage de la recherche compense cet investissement, mais le temps considérable consacré à chaque nœud reste un enjeu majeur.

7.3 Améliorations futures

Si le présent mémoire indique un intérêt du passage de messages pour guider la recherche en CP, il ouvre aussi de nombreux questionnements et opportunités d'améliorations.

Le passage de messages et la convergence des marginales présentant des comportements complexes, instrumenter le solveur pour produire une étude plus détaillée de l'évolution des marginales pourrait fournir des renseignements précieux. On pourrait alors mieux caractériser l'influence de facteurs tels que la topologie du réseau de contrainte, le poids de l'Oracle, etc...

Appliquer l'approche Max-Product à des problèmes plus variés nécessite l'ajout d'algorithmes pour pouvoir calculer les messages de contraintes globales plus nombreuses. Les algorithmes et structures de données associés à Sum-Product fournissent souvent un excellent point de départ pour la conception des algorithmes Max-Product.

Max-Product peut sans doute bénéficier d'une implémentation plus optimisée. Par exemple, une contrainte pourrait envoyer des messages uniformes sans avoir à les calculer réellement si les variables qui y participent ont toutes des marginales uniformes. De telles améliorations pourraient rendre Max-Product plus compétitif avec l'implémentation plus mature de Sum-Product présente dans le solveur MiniCPBP.

Enfin, de nombreux travaux portent sur l'amélioration des algorithmes de passage de messages. Ceux-ci peuvent viser à accélérer la convergence [33], augmenter la résilience à la présence de cycles [34], introduire du parallélisme [35]... Introduire ces améliorations dans le solveur MiniCPBP pourrait mitiger le coût de l'inférence à chaque nœud, ou produire des marginales plus utiles dans le guidage de la recherche. Ceci pourrait rendre l'approche viable pour une gamme plus large d'applications.

7.4 Conclusion

Ce mémoire présente une extension de l'inférence par passage de messages dans la CP des problèmes de satisfaction aux problèmes d'optimisation, via la contrainte Oracle et l'algorithme Max-Product. Nous avons pu montrer que le passage de messages et les distributions qui en résultent permettent de guider la recherche et d'arriver plus rapidement à des solutions de meilleure qualité. Cette nouvelle méthode pourrait à terme permettre à la CP d'être compétitive sur un plus large éventail de problèmes d'optimisation.

RÉFÉRENCES

[1] R. Dechter, *Constraint processing*. Elsevier Morgan Kaufmann, 2003. [En ligne]. Disponible : http://www.elsevier.com/wps/find/bookdescription.agents/678024/description

[2] C. Lecoutre, "Restarts and Nogood Recording," dans *Constraint Networks : Techniques and Algorithms*. Wiley-IEEE Press, août 2009.

[3] P. J. Stuckey, "Lazy clause generation : Combining the power of SAT and CP (and mip ?) solving," dans *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, ser. Lecture Notes in Computer Science, A. Lodi, M. Milano et P. Toth, édit., vol. 6140. Springer, 2010, p. 5–9. [En ligne]. Disponible : https://doi.org/10.1007/978-3-642-13520-0_3

[4] C. Lecoutre, "Backtrack Search," dans *Constraint Networks : Techniques and Algorithms*. Wiley-IEEE Press, août 2009.

[5] G. Pesant, "A regular language membership constraint for finite sequences of variables," dans *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, ser. Lecture Notes in Computer Science, M. Wallace, édit., vol. 3258. Springer, 2004, p. 482–495. [En ligne]. Disponible : https://doi.org/10.1007/978-3-540-30201-8_36

[6] M. A. Trick, "A dynamic programming approach for consistency and propagation for knapsack constraints," *Ann. Oper. Res.*, vol. 118, n°. 1-4, p. 73–84, 2003. [En ligne]. Disponible : https://doi.org/10.1023/A:1021801522545

[7] G. Pesant, "From support propagation to belief propagation in constraint programming," *J. Artif. Intell. Res.*, vol. 66, p. 123–150, 2019. [En ligne]. Disponible : https://doi.org/10.1613/jair.1.11487

[8] ——, "Counting-based search for constraint optimization problems," dans *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, D. Schuurmans et M. P. Wellman, édit. AAAI Press, 2016, p. 3441–3448. [En ligne]. Disponible : https://doi.org/10.1609/aaai.v30i1.10433

[9] J. Régin, "A filtering algorithm for constraints of difference in csps," dans *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1*, B. Hayes-Roth et R. E. Korf, édit. AAAI Press / The MIT Press, 1994, p. 362–367. [En ligne]. Disponible : http://www.aaai.org/Library/AAAI/1994/aaai94-055.php

[10] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J. Régin et P. Schaus, "Compact-table : Efficiently filtering table constraints with reversible sparse bit-sets," dans *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, ser. Lecture Notes in Computer Science, M. Rueher, édit., vol. 9892. Springer, 2016, p. 207–223. [En ligne]. Disponible : https://doi.org/10.1007/978-3-319-44953-1_14

[11] F. Boussemart, F. Hemery, C. Lecoutre et L. Sais, "Boosting systematic search by weighting constraints," dans *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, R. L. de Mántaras et L. Saitta, édit. IOS Press, 2004, p. 146–150.

[12] B. Babaki, B. Omrani et G. Pesant, "Combinatorial search in cp-based iterated belief propagation," dans *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, ser. Lecture Notes in Computer Science, H. Simonis, édit., vol. 12333. Springer, 2020, p. 21–36. [En ligne]. Disponible : https://doi.org/10.1007/978-3-030-58475-7_2

[13] G. W. Soules, "New permanental upper bounds for nonnegative matrices," *Linear and Multilinear Algebra*, vol. 51, n°. 4, p. 319–337, déc. 2003, publisher : Taylor & Francis _eprint : https ://doi.org/10.1080/0308108031000098450.

[14] S. Parsons, "*Probabilistic Graphical Models : Principles and Techniques* by daphne koller and nir friedman, MIT press, 1231 pp., $95.00, ISBN 0-262-01319-3," *Knowl. Eng. Rev.*, vol. 26, n°. 2, p. 237–238, 2011. [En ligne]. Disponible : https://doi.org/10.1017/S0269888910000275

[15] R. Dechter, B. Bidyuk, R. Mateescu et E. Rollon, *On the Power of Belief Propagation : A Constraint Propagation Perspective*. College Publications, January 2010, p. 137–158. [En ligne]. Disponible : https://www.microsoft.com/en-us/research/publication/on-the-power-of-belief-propagation-a-constraint-propagation-perspective/

[16] J. Pearl, "Belief Updating by Network Propagation," dans *Probabilistic Reasoning in Intelligent Systems*, J. Pearl, édit. San Francisco (CA) : Morgan Kaufmann, janv. 1988, p. 143–237. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/B9780080514895500102

[17] J. S. Yedidia, W. T. Freeman et Y. Weiss, "Generalized belief propagation," dans *Advances in Neural Information Processing Systems 13, Papers from Neural Information Processing Systems (NIPS) 2000, Denver, CO, USA*, T. K. Leen, T. G. Dietterich et V. Tresp, édit. MIT Press, 2000, p.

689–695. [En ligne]. Disponible : https://proceedings.neurips.cc/paper/2000/hash/61b1fb3f59e28c67f3925f3c79be81a1-Abstract.html

[18] A. Burlats et G. Pesant, "Exploiting entropy in constraint programming," dans *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 20th International Conference, CPAIOR 2023, Nice, France, May 29 - June 1, 2023, Proceedings*, ser. Lecture Notes in Computer Science, A. A. Ciré, édit., vol. 13884. Springer, 2023, p. 320–335. [En ligne]. Disponible : https://doi.org/10.1007/978-3-031-33271-5_21

[19] "2.7. Effective Modelling Practices in MiniZinc — The MiniZinc Handbook 2.9.3." [En ligne]. Disponible : https://docs.minizinc.dev/en/stable/efficient.html#modelling-choices

[20] R. Dechter, *Reasoning with Probabilistic and Deterministic Graphical Models : Exact Algorithms, Second Edition*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. [En ligne]. Disponible : https://doi.org/10.2200/S00893ED2V01Y201901AIM041

[21] G. Pesant, "Counting solutions of csps : A structural approach," dans *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, L. P. Kaelbling et A. Saffiotti, édit. Professional Book Center, 2005, p. 260–265. [En ligne]. Disponible : http://ijcai.org/Proceedings/05/Papers/0624.pdf

[22] ——, "Getting more out of the exposed structure in constraint programming models of combinatorial problems," dans *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, S. Singh et S. Markovitch, édit. AAAI Press, 2017, p. 4846–4851. [En ligne]. Disponible : https://doi.org/10.1609/aaai.v31i1.11143

[23] G. Pesant, C. Quimper et A. Zanarini, "Counting-based search : Branching heuristics for constraint satisfaction problems," *J. Artif. Intell. Res.*, vol. 43, p. 173–210, 2012. [En ligne]. Disponible : https://doi.org/10.1613/jair.3463

[24] A. Rogers, A. Farinelli, R. Stranders et N. R. Jennings, "Bounded approximate decentralised coordination via the max-sum algorithm," *Artif. Intell.*, vol. 175, n°. 2, p. 730–759, 2011. [En ligne]. Disponible : https://doi.org/10.1016/j.artint.2010.11.001

[25] A. Farinelli, A. Rogers et N. R. Jennings, "Agent-based decentralised coordination for sensor networks using the max-sum algorithm," *Auton. Agents Multi Agent Syst.*, vol. 28, n°. 3, p. 337–380, 2014. [En ligne]. Disponible : https://doi.org/10.1007/s10458-013-9225-1

[26] M. C. Cooper, S. de Givry, M. Sánchez-Fibla, T. Schiex, M. Zytnicki et T. Werner, "Soft arc consistency revisited," *Artif. Intell.*, vol. 174, n°. 7-8, p. 449–478, 2010. [En ligne]. Disponible : https://doi.org/10.1016/j.artint.2010.02.001

[27] V. Manibod, "Ajout de structure aux modèles génératifs de séquences avec la programmation par contraintes," Mémoire de maîtrise, Polytechnique Montréal, août 2022. [En ligne]. Disponible : https://publications.polymtl.ca/10495/

[28] C. Gomes et D. Shmoys, "Completing Quasigroups or Latin Squares : A Structured Graph Coloring Problem," 2002. [En ligne]. Disponible : https://www.semanticscholar.org/paper/Completing-Quasigroups-or-Latin-Squares%3A-A-Graph-Gomes-Shmoys/df5e4a0b3093c1f399668fbf419d9a9efa5befae

[29] R. E. Burkard, M. Dell'Amico et S. Martello, *Assignment Problems*. SIAM, 2009. [En ligne]. Disponible : https://doi.org/10.1137/1.9781611972238

[30] J. Munkres, "Algorithms for the Assignment and Transportation Problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 5, n°. 1, p. 32–38, mars 1957, publisher : Society for Industrial & Applied Mathematics (SIAM). [En ligne]. Disponible : http://epubs.siam.org/doi/10.1137/0105003

[31] D. F. Crouse, "On implementing 2d rectangular assignment algorithms," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 52, n°. 4, p. 1679–1696, 2016. [En ligne]. Disponible : https://doi.org/10.1109/TAES.2016.140952

[32] F. Boussemart, C. Lecoutre et C. Piette, "XCSP3 : an integrated format for benchmarking combinatorial constrained problems," *CoRR*, vol. abs/1611.03398, 2016. [En ligne]. Disponible : http://arxiv.org/abs/1611.03398

[33] S. Tong, B. Bai et X. Wang, "Convergence rates comparison of sum-product decoding of RA codes under different message-passing schedules," *IEEE Commun. Lett.*, vol. 9, n°. 6, p. 543–545, 2005. [En ligne]. Disponible : https://doi.org/10.1109/LCOMM.2005.1437365

[34] A. Kirkley, G. T. Cantwell et M. E. J. Newman, "Message passing for probabilistic models on networks with loops," *CoRR*, vol. abs/2009.12246, 2020. [En ligne]. Disponible : https://arxiv.org/abs/2009.12246

[35] A. Mendiburu, R. Santana, J. A. Lozano et E. Bengoetxea, "A parallel framework for loopy belief propagation," dans *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007, Companion Material*, D. Thierens, édit. ACM, 2007, p. 2843–2850. [En ligne]. Disponible : https://doi.org/10.1145/1274000.1274084