Titre: Title:	Optimisation de la recherche combinatoire d'un solveur mêlant programmation par contrainte et belief propagation
Auteur: Author:	Auguste Burlats
Date:	2022
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Burlats, A. (2022). Optimisation de la recherche combinatoire d'un solveur mêlant programmation par contrainte et belief propagation [Master's thesis, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/10444/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/10444/
Directeurs de recherche: Advisors:	Gilles Pesant
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Optimisation de la recherche combinatoire d'un solveur mêlant Programmation Par Contrainte et Belief Propagation

AUGUSTE BURLATS
Département de génie informatique et génie logiciel

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

Août 2022

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Optimisation de la recherche combinatoire d'un solveur mêlant Programmation Par Contrainte et Belief Propagation

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

Guillaume-Alexandre BILODEAU, président
Gilles PESANT, membre et directeur de recherche
Louis-Martin ROUSSEAU, membre

REMERCIEMENTS

Je tiens à remercier mon directeur de recherche Gilles Pesant pour avoir supervisé ce projet de recherche. Il a consacré beaucoup de temps et d'énergie pour me guider dans ma recherche et dans la rédaction de ce mémoire. Son aide a été inestimable.

Même si nous n'avons pas eu beaucoup l'occasion de nous voir en personne, je tiens à remercier les membre du laboratoire QUOSSEÇA pour leur accueil chaleureux.

Je tiens aussi à remercier ma famille qui m'ont beaucoup supporté et encouragé. Je suis heureux de les avoir. Je remercie aussi mes amis. Leur soutien m'a été très précieux.

RÉSUMÉ

La programmation par contraintes est une méthode populaire pour résoudre les problèmes combinatoires. Cette popularité est en grande partie due à sa capacité à exploiter les contraintes pour accélérer fortement la résolution. En effet, les contraintes lui permettent de filtrer les valeurs impossibles au fur et à mesure de la recherche de solution, réduisant ainsi fortement l'espace de recherche. Mais l'amplitude de ce filtrage est fortement impactée par l'ordre dans lequel les variables sont fixées. Pour cette raison, les heuristiques d'ordonnancement de variables robustes et généralisables sont un enjeu crucial pour la programmation par contraintes. L'ajout de la propagation de croyances à un solveur de programmation par contraintes permet de calculer, pour chaque affectation, les distributions marginales. Ces distributions correspondent aux proportions de solutions valides contenant les différentes affectations. Ce sont donc des bases très intéressantes pour une heuristique d'ordonnancement de variables et de valeurs.

Le travail présenté dans ce mémoire a pour but d'améliorer l'usage de la propagation de croyances. Nous présentons tout d'abord deux critères d'arrêt dynamiques pour cette propagation. L'idée est de l'arrêter au moment propice afin d'éviter toute itération inutile. Le premier critère observe la stabilité du meilleur marginal présent dans les distributions. Le deuxième observe l'évolution de l'entropie du modèle. De plus, nous présentons aussi deux manières de pondérer les messages envoyés au cours de la propagation. La première est de pondérer en fonction de l'arité des contraintes, tandis que l'autre est basée sur le nombre d'échecs provoqués par chaque contrainte. Enfin, nous présentons trois nouvelles heuristiques : *min-entropy*, choisissant de fixer la variable avec la plus faible entropie, *impact-entropy* qui choisit les variables à fixer de manière à réduire le plus fortement possible l'entropie du modèle et une combinaison de ces deux heuristiques nommée *impact-min-entropy*.

Nous avons testé nos contributions sur 1474 exemplaires parmi 11 problèmes différents. Les résultats montrent que nos critères d'arrêt dynamiques pour la propagation de croyance sont effectivement capables d'induire une réduction du temps de calcul. De plus, nos schémas de pondération améliorent significativement les performances de notre approche pour certains problèmes. Enfin, nos expériences démontrent que l'heuristique *min-entropy* est une meilleure exploitation des distributions marginales que les autres heuristiques les utilisant. Elle est de plus compétitive avec l'heuristique de l'état de l'art *dom/wdeg*.

ABSTRACT

Constraint Programming is a popular method to solve combinatorial problems. This popularity is mostly due to its capacity to exploit constraints in order to strongly reduce the runtime. Indeed, constraints allow to filter impossible values through the search of a solution, strongly reducing the search space. But the amplitude of this filtering is strongly impacted by the order in which variables are fixed. Therefore, robust and generalizable variable ordering heuristics are a major issue in Constraint Programming. By adding belief propagation to a Constraint Programming solver, it is possible to compute marginal distributions for each assignment. Those distributions represent the proportion of valid solutions that contain the assignments. Therefore they can be very interesting tools for variable and value ordering heuristics.

The work presented in this thesis aims to improve the use of belief propagation. First, we present two dynamic stopping criteria for this propagation. The goal is to stop it at the best moment in order to avoid any useless iteration. The first criterion looks at the stability of the best marginal of all distributions. The second one looks at the variation of the model's entropy. We also present two ways to weight messages during belief propagation. The first one is to weight based on the arity of constraints, the second one uses the number of conflicts created by each constraint. Finally, we present three new variable ordering heuristics : *min-entropy*, which fixes the variables with the lowest entropy, *impact-entropy*, which fixes variables that are supposed to have the strongest impact on the model's entropy, and a combination of these two heuristics named *impact-min-entropy*.

We tested our contributions on 1474 instances from 11 different problems. Results show that our dynamic stopping criteria are able to induce a reduction of runtime. Furthermore our weighing schemes induce a significant improvement of the performance of our approach for several problems. Finally, our experiments show that *min-entropy* is a better way to exploit marginal distributions than other belief-propagation-based heuristics. Moreover it is competitive with state-of-the-art heuristic *dom/wdeg*.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

CSP	Problème de Satisfaction de Contraintes
COP	Problème d'Optimisation de Contraintes
PPC	Programmation Par Contraintes
LDS	Limited Discrepancy Search
DFS	Recherche en profondeur
BP	Propagation de croyance / Belief Propagation

CHAPITRE 1 INTRODUCTION

Un problème combinatoire est un problème consistant à trouver une combinaison de valeurs satisfaisant différents objectifs. Cette définition peut englober énormément de problèmes concrets différents : emploi du temps, budget, planification de tournées, etc. Ces problèmes sont non seulement très variés dans leurs structures, mais aussi chargés d'enjeux, notamment pour l'industrie. Mais une de leurs principales caractéristiques est que l'ensemble des combinaisons possibles est très souvent gigantesque, rendant leur résolution ardue. En effet, la plupart de ces problèmes sont $\mathcal{NP}$-difficiles, et donc, pour ceux-là, aucun algorithme capable de les résoudre en un temps polynomial n'a été trouvé. Trouver une méthode apte à leur trouver des solutions valides tout en étant performante sur une grande variété de problèmes est donc un enjeu crucial. La Programmation par contraintes (*PPC*) est une solution intéressante à ce problème. Le travail présenté dans ce mémoire part d'une approche de *PPC* basée sur un algorithme appelé *Belief Propagation (BP)* ayant donné des performances intéressantes pour la résolution de ces problèmes combinatoires [1, 2]. Le but est d'améliorer cette approche afin d'obtenir de meilleures performances.

Dans ce chapitre, nous donnons une brève introduction à la *PPC* et à l'usage de la *BP* pour en améliorer les performances. Les deux dernières sections présentent les objectifs de la recherche et le plan de ce mémoire.

1.1 Programmation par Contraintes

Commençons par formaliser les problèmes dont nous parlons. Un Problème de Satisfaction de Contraintes (*CSP*) est un problème consistant à trouver une combinaison de valeurs satisfaisant une ou plusieurs contraintes.

Un *CSP* est caractérisé par un triplet $< X, D, C >$, où

- $X = \{x_1, x_2, x_3, ..., x_N\}$ est ensemble de variables ;
- $D = \{v_1, v_2, v_3, ..., v_N\}$ est un ensemble de valeurs ;
- $C = \{c_1, c_2, c_3, ..., c_m\}$ est un ensemble de contraintes.

Le sous-ensemble non vide $D(x_i) \subseteq D$ correspond au *domaine* de la variable x_i, c'est-à-dire à l'ensemble des valeurs que peut prendre cette variable.

Une contrainte c_i s'applique au sous-ensemble non vide $X(c_i) \subseteq X$. On appelle ce sous-ensemble la *portée* de c_i et $|X(c_i)|$ est son *arité*. Le rôle des contraintes est des restreindre les combinaisons de variables pour les variables dans leur portée.

Le but est de trouver pour chaque variable une valeur qui n'enfreigne aucune contrainte.

Il est possible d'ajouter une fonction $f(x_1, x_2, ..., x_N)$ à minimiser. Le problème devient alors un Problème d'Optimisation de Contraintes (COP). La fonction est appelée *fonction de coût*.

La Programmation par Contraintes [3] est un paradigme de programmation permettant de résoudre des *CSP*. Ce paradigme présente deux avantages : Le premier est la possibilité de créer des langages descriptifs pour résoudre ces problèmes. Cela permet de résoudre plus aisément de nombreux problèmes, et d'ajouter facilement des contraintes à un modèle. Le deuxième avantage est que la *PPC* permet de faire ressortir et d'exploiter la structure des problèmes à travers ces contraintes. Ainsi la résolution de ces problèmes peut être considérablement accélérée. Nous allons voir comment par la suite.

1.1.1 Modélisation

La modélisation est la formalisation en *CSP* du problème. C'est cette formalisation que l'on va fournir au solveur *PPC*. Modéliser un problème correspond à choisir les variables utilisées et les contraintes à y appliquer.

Intéressons-nous à la modélisation d'un *CSP* à la structure simple. Le *Sudoku* est un jeu dérivé du *Carré Latin* où il faut placer des chiffres entre 1 et 9 dans une grille de dimension 9×9. Mais ils ne doivent pas être placés n'importe comment : un chiffre ne peut apparaître qu'une seule fois par ligne et par colonne. De plus, la grille est divisée en 9 sections carrées de dimensions 3×3. Dans ces carrés, un chiffre ne peut pas être présent plus d'une seule fois.

La modélisation de la grille présentée dans la figure 1.1 est la suivante :

— $X = \{x_{i,j} \quad | \quad i, j \in [1, 9]\}$

— $D = \{D(x_{i,j}) = [1, 9], \quad \forall i, j \in [1, 9]\}$

— $C = \begin{cases} x_{1,3} = 5, x_{1,6} = 3, x_{1,7} = 8, ..., x_{9,9} = 9, & (a) \\ AllDifferent(x_{1,j}, x_{2,j}, ..., x_{9,j}) \quad | \quad j \in [1, 9], & (b) \\ AllDifferent(x_{i,1}, x_{i,2}, ..., x_{i,9}) \quad | \quad i \in [1, 9], & (c) \\ AllDifferent(x_{i,j}, x_{i,j+1}, x_{i,j+2}, x_{i+1,j}, ..., x_{i+2,j+2}) \quad | \quad i, j \in \{1, 4, 7\} & (d) \end{cases}$

Dans cette formulation, les contraintes à la ligne (a) correspondent aux valeurs déjà présentes dans la grille avant la résolution. Les contraintes AllDifferent en (b) empêche les valeurs doublons sur les colonnes. Celles en (c) empêchent les valeurs doublons sur les lignes. Enfin les contraintes AllDifferent en (d) empêchent les doublons dans les carrés.

Pour chaque problème il existe plusieurs modélisations possibles. Faire les bons choix de modélisation est crucial, car cela va fortement impacter les performances de la résolution. Mais intéressons-nous justement à cette résolution.

		5			3	8		
2							6	
	1							2
			4					
8						1	9	
			9					8
		4			8			3
			7		6		5	
7	3		1		4			9

FIGURE 1.1 Grille de Sudoku modélisée dans l'exemple de la section 1.1.1

1.1.2 Filtrage des valeurs

La *PPC* réduit fortement le temps de calcul en exploitant les contraintes afin de filtrer les valeurs ne respectant pas les contraintes. Concrètement, lorsqu'une variable voit son domaine être modifié, par exemple lorsqu'elle fixée, cela va "réveiller" les contraintes concernées. Des algorithmes de filtrage vont être appliqués aux domaines des autres variables afin de retirer les valeurs ne pouvant pas respecter les contraintes. On parle de *propagation de contraintes*. Chaque contrainte dispose d'algorithmes de filtrage spécifiques sur lesquels nous ne reviendrons pas en détail. Lorsqu'au cours de la propagation, une variable voit son domaine être totalement vidé, et donc qu'elle n'a plus de valeur possible, un échec est détecté. On remonte alors au nœud précédent dans l'arbre de recherche.

Exemple : Soient $X = \{x_1, x_2, x_3\}$ et $D(x_1) = \{1, 2, 3\}, D(x_2) = \{1, 2, 4\}, D(x_3) = \{1, 2, 4\}$. Les contraintes sont $c_1 : x_1 + x_2 = 5$ et $c_2 : allDifferent(x_1, x_2, x_3)$. Une première propagation va être effectuée avant de commencer la recherche et va supprimer la valeur 1 de $D(x_2)$ et la valeur 2 de $D(x_1)$. En effet, ces valeurs ne peuvent pas respecter c_1 : $D(x_1)$ ne contient pas de valeur v telle que $v + 1 = 5$ et $D(x_2)$ ne contient pas de valeur v' telle que $v' + 2 = 5$ Ensuite, la recherche va commencer. Si nous utilisons un ordre lexicographique pour le choix de valeur et le choix de variable, c'est l'affectation $x_1 = 1$ qui va être essayée en premier. Cette modification du domaine de x_1 va activer la propagation de contraintes. La contrainte c_2 va retirer la valeur 1 de $D(x_3)$. Puis la contrainte c_1 va retirer la valeur 2 de $D(x_2)$, car $1 + 2 \neq 5$. Nous avons donc $x_2 = 4$. Cette modification des domaines va activer une nouvelle propagation des contraintes, et c_2 va permettre de retirer la valeur 4 de $D(x_3)$. On a donc

$x_3 = 2$. Ainsi $(1, 4, 2)$ est une solution possible du problème. La figure 1.2 illustre l'évolution du filtrage.

1.1.3 Résolution

Pour résoudre les problèmes, la Programmation par contraintes utilise une recherche arborescente. À chaque nœud de l'arbre, une variable est choisie et un choix est fait sur son domaine (retirer des valeurs, fixer une valeur, etc.).

Différents types de branchement sont disponibles, mais l'un des plus populaires, et celui utilisé au cours de ce projet, est le suivant : À chaque nœud, deux branches sont créées : dans la première, on affecte à la variable une certaine valeur de son domaine. Dans la deuxième, on retire cette valeur du domaine de la variable.

De même, plusieurs solutions pour le parcours de ce graphe sont possibles. Les deux utilisées dans ce mémoire sont la recherche en profondeur (ou *DFS* pour *Depth-First Search*) et la *Limited discrepancy search* (*LDS*) [4]. La *DFS* va suivre un chemin sans remonter dans l'arbre jusqu'à atteindre un cul-de-sac. Ensuite, elle va remonter dans l'arbre pour prendre la première déviation disponible et redescendre dans l'arbre jusqu'à un autre cul-de-sac. La *LDS* est similaire à la *DFS*, mais ne considère pas les chemins dans le même ordre. Elle ordonne les chemins à explorer selon le nombre de fois qu'elle ne suit pas la recommandation de l'heuristique. Elle commence en suivant un chemin qui ne dévie jamais, puis passe à ceux qui ne dévient qu'une seule fois, puis ceux qui dévient deux fois, etc. La figure 1.3 présente un exemple pour différencier les deux recherches. Nous pouvons constater que les feuilles explorées en 4e et 5e positions sont échangées selon si c'est une *DFS* ou une *LDS*. L'intérêt de la *LDS* est que, étant donné qu'elle ordonne les chemins à prendre en fonction de leur accord avec l'heuristique, elle permet d'exploiter plus fortement une heuristique fiable.

Des relances peuvent être réalisées au cours de la recherche. Cette technique vient des solveurs *SAT*, elle a été introduite par [5] afin de résoudre le problème des *heavy-tailed cost distributions*. Un phénomène rendant les temps de calcul plus imprévisibles : deux exemplaires identiques mais où les variables sont ordonnées différemment peuvent être résolus dans des temps radicalement différents. Ce problème a été résolu par l'ajout d'aléatoire et de relance au cours de la recherche. Les relances permettent à la recherche d'éviter de rester coincée dans une même partie de l'arbre et donc de pouvoir explorer une plus grande variété de branches. C'est une technique très utilisée dans de nombreux solveurs SAT comme *Chaff* [6] ou *PicoSAT* [7]. Le premier relance la recherche après un nombre fixe de conflits rencontrés. Le deuxième commence avec un petit nombre d'échecs avant relance et ce nombre croît géométriquement après chaque redémarrage. Les solveurs *PPC* peuvent bénéficier de ces re-

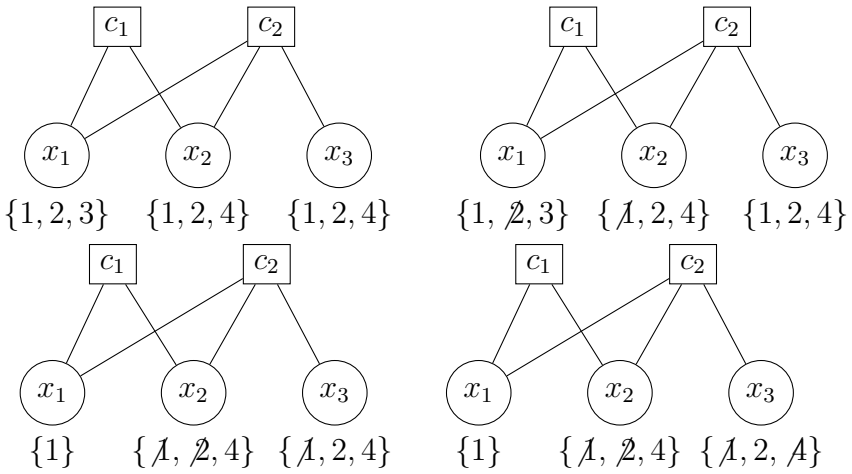

FIGURE 1.2 Évolution des domaines des variables avant les propagations de contraintes (en haut à gauche), après la première propagation de contrainte (en haut à droite), après la deuxième propagation de contrainte (en bas à gauche) et après la troisième propagation de contrainte (en bas à droite) pour l'exemple de la section 1.1.2

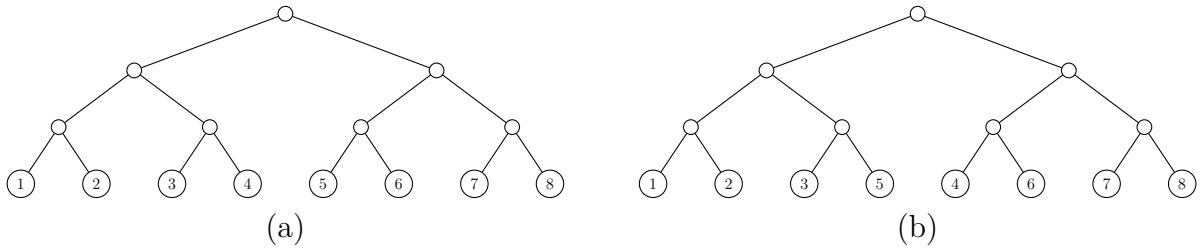

FIGURE 1.3 Schémas présentant l'ordre d'exploration des feuilles d'un arbre de recherche dans le cas d'une *DFS* (a) et d'une *LDS* (b)

démarrages : comme expliqué par la suite dans ce mémoire, certaines stratégies de recherche apprennent pendant cette recherche afin de s'adapter. Dans ce cas, relancer la recherche permet d'exploiter en haut de l'arbre les connaissances les plus récentes. L'usage de relance n'est pertinent que dans le cas d'heuristiques utilisant de l'aléatoire ou d'heuristiques apprenant au cours de la recherche.

1.1.4 Heuristiques de branchement

Comme avec toute recherche arborescente, une bonne heuristique de choix de valeur est nécessaire. Il vaut mieux attribuer à chaque variable la bonne valeur le plus tôt possible afin d'éviter à avoir à explorer la majorité de l'arbre. Mais la *PPC*, avec ses algorithmes de filtrage, introduit une nouvelle question en plus de celle de l'ordonnancement des valeurs. En effet, la *PPC* économise du temps de calcul en élaguant l'arbre de recherche. Mais l'amplitude

de cet élagage, et donc l'économie sur le temps de recherche, dépend de l'ordre dans lequel les variables sont fixées. C'est pour cette raison que l'ordonnancement des variables est une question cruciale.

Les choix les plus cruciaux se font au sommet de l'arbre. En effet, ce sont ceux qui vont avoir le plus d'impact sur le nombre de nœuds explorés. Un principe très important en CP, et suivi par de nombreuses stratégies de recherche, est le *Fail first principle* [8]. Ce principe dit qu'il faut prendre les décisions pouvant mener le plus rapidement possible à un échec. En effet, si l'on se trouve dans un sous-arbre ne contenant pas de solutions valides, il vaut mieux en sortir le plus vite possible. Une application directe de ce principe est l'heuristique *dom* [8]. Cette heuristique choisit de fixer, à chaque branchement, la variable présentant le plus petit domaine. Cette approche donnant de bonnes performances et étant simple à implémenter, elle a longtemps été une heuristique de référence. Ses améliorations comme *dom/wdeg* [9] sont toujours très populaires. Mais d'autres approches ont été explorées pour décider de l'ordre de branchement des variables. La section suivante présente celle sur laquelle se base notre travail.

1.2 Belief Propagation et Programmation par contraintes

La *Belief Propagation* (*BP*) a été introduite en 1982 par Pearl [10]. Cet algorithme permet de calculer une approximation des probabilités marginales (*belief*) de variables en inférant à partir de distributions de probabilités jointes. Cet algorithme s'applique à des modèles graphiques tels que des réseaux bayésiens. L'inférence est réalisée grâce à des échanges de messages entre les nœuds composant le réseau.

Un *CSP* peut être représenté comme un *factor graph* : les nœuds facteurs correspondent alors aux contraintes, et les nœuds variables correspondent aux variables. Dans le cas de ce type de graphe, les messages envoyés par les contraintes et par les variables sont différents.

Le message envoyé par la variable x à la contrainte c à propos de la valeur $v \in D(x)$ est :

$$\mu_{x \to c}(v) = \prod_{c' \in N(x) \setminus \{c\}} \mu_{c' \to x}(v) \tag{1.1}$$

Le message envoyé par la contrainte c à la variable x à propos de la valeur v est :

$$\mu_{c \to x}(v) = \sum_{\mathbf{v} : \mathbf{v}[x] = v} f_c(\mathbf{v}) \prod_{x' \in X(c) \setminus \{x\}} \mu_{x' \to c}(\mathbf{v}[x']) \tag{1.2}$$

Dans ces formules, $N(x)$ est le voisinage de la variable x, c'est-à-dire l'ensemble des contraintes

qui lui sont appliquées, $X(c)$ est le voisinage de la contrainte c, son scope, $\mathbf{v}$ est un tuple issu du produit cartésien de toutes les variables dans la portée de c, $\mathbf{v}[x]$ est la valeur prise par x dans $\mathbf{v}$ et $f_c(\mathbf{v})$ est une fonction retournant 1 si le tuple $\mathbf{v}$ satisfait la contrainte c et 0 sinon.

Calculer $\sum_{\mathbf{v}:\mathbf{v}[x]=v} f_c(\mathbf{v})$ équivaut à compter les solutions (locales à c) où $\mathbf{v}[x] = v$. De ce fait, calculer les messages venant des contraintes revient à faire du comptage de solutions pondéré. Pesant [1] a fourni des algorithmes dédiés efficaces pour performer ce type de comptage sur plusieurs contraintes.

Grâce aux messages, il est possible de calculer la probabilité marginale pour la valeur $v \in D(x)$:

$$\theta_x(v) = \prod_{c \in N(x)} \mu_{c \to x}(v) \tag{1.3}$$

Ce marginal représente la probabilité qu'a la valeur v d'être dans la solution finale. En d'autres termes, $\theta_x(v)$ est une estimation de la proportion de solutions valides où $x = v$.

Prenons un exemple tiré de [1] pour illustrer le comportement des marginaux : Considérons quatre variables a, b, c et d, avec $D(a) = D(b) = D(c) = D(d) = \{1, 2, 3, 4\}$ et soumises aux contraintes suivantes :

- $alldifferent(a, b, c)$
- $a + b + c + d = 7$
- $c \leq d$

Ce *CSP* a deux solutions : $(a = 2, b = 3, c = 1, d = 1)$ et $(a = 3, b = 2, c = 1, d = 1)$. Si nous nous intéressons à la variable a, nous pouvons constater que l'affectation $a = 2$ est présente dans une solution valide, soit la moitié de ces solutions. La probabilité marginale de cette affectation est donc $\theta_a(2) = 0.5$. Il n'y a aucune solution contenant $a = 1$, ainsi $\theta_a(1) = 0$. Enfin, nous pouvons constater, en regardant la variable c, que l'affectation $c = 1$ est présente dans la totalité des solutions valides, on a donc $\theta_c(1) = 1$. En appliquant la même logique aux autres affectations, nous obtenons les distributions du premier tableau du tableau 1.1. La *BP* part de distributions uniformes pour chacune des variables : $\theta_{x_i}(v) = 1/|D(x_i)|, \forall v \in D(x_i), \forall x_i \in X$. Et comme nous pouvons l'observer dans le tableau 1.1, les distributions finissent par converger vers les distributions réelles.

Ainsi, ces probabilités marginales sont potentiellement un excellent support pour choisir les variables et les valeurs à fixer. Les exploiter pour mettre au point une heuristique de branchement peut donner de bons résultats. C'est notamment le cas avec l'heuristique *max-marginal* [2]. Cette heuristique choisit de fixer la paire variable-valeur possédant le marginal le plus élevé. Autrement dit, l'heuristique branche en priorité sur les paires les plus susceptibles

TABLEAU 1.1 Distributions marginales réelles (en haut à gauche), distributions initiales (en haut à droite), distributions après une itération (en bas à gauche) et après 10 itérations (en bas à droite) pour l'exemple dans la section 1.2

	1	2	3	4
θ_a	0	.50	.50	0
θ_b	0	.50	.50	0
θ_c	1	0	0	0
θ_d	1	0	0	0

	1	2	3	4
θ_a	.25	.25	.25	.25
θ_b	.25	.25	.25	.25
θ_c	.25	.25	.25	.25
θ_d	.25	.25	.25	.25

	1	2	3	4
θ_a	.50	.30	.15	.05
θ_b	.50	.30	.15	.05
θ_c	.62	.28	.09	.01
θ_d	.29	.34	.26	.11

	1	2	3	4
θ_a	.01	.52	.46	.01
θ_b	.01	.52	.46	.01
θ_c	.98	.02	.00	.00
θ_d	.90	.10	.00	.00

d'être présentes dans la solution finale. Cette approche a montré des résultats intéressants mais certaines pistes peuvent être explorées pour tenter de l'améliorer. C'est à cela que se consacre le travail présenté dans ce mémoire.

1.3 Objectifs de la Recherche

Le but de ce travail de recherche est d'améliorer l'usage de la *BP* dans le cadre de la *PPC*. Nous avons identifié trois axes d'amélioration potentiels :

Tout d'abord, il est possible d'utiliser les marginaux différemment. En nous basant sur la définition de l'entropie de Shannon [11], ils nous permettent de calculer l'entropie de chaque variable. Originellement l'entropie de Shannon est utilisée dans les télécommunications. Elle indique quelle quantité d'information une source doit envoyer afin qu'il n'y ait pas d'ambiguïté sur le message reçu par le récepteur. Autrement dit, elle quantifie l'incertitude qu'il y a sur un message reçu. Il est donc possible de l'utiliser dans notre cas afin d'estimer l'incertitude que nous avons à propos d'une variable. Nous proposons trois nouvelles heuristiques basées sur l'entropie. Ainsi notre première question de recherche est :
Q1 : Est-ce qu'exploiter l'entropie est une approche intéressante pour exploiter les distributions marginales ?

Ensuite, dans l'état actuel des choses le nombre d'itérations de *BP* avant chaque branchement est fixe. Cette situation n'est pas avantageuse, nous proposons donc des critères d'arrêts permettant de décider dynamiquement à chaque nœud quand stopper la *BP*. Notre seconde question de recherche est :
Q2 : Est-il possible de décider dynamiquement quand stopper les itérations de *BP* afin de

gagner du temps de calcul ?

Enfin, au cours des itérations de *BP* tous les messages ont le même poids. Donner plus d'importance aux messages venant de certaines contraintes ou variables pourrait accélérer la convergence des marginaux. Nous proposons deux schémas de pondération se concentrant sur les messages venant des contraintes. Notre dernière question est donc :
Q3 : Pondérer les messages des contraintes peut-il permettre d'obtenir des distributions plus proches des distributions réelles ?

1.4 Plan du Mémoire

Ce mémoire est organisé comme suit :

Le chapitre 2 contient la revue de littérature, consacrée aux principales heuristiques de branchement utilisées en *PPC*.

Le chapitre 3 présente la contribution principale de ce travail de recherche. La première partie est consacrée aux solutions explorées pour ajuster dynamiquement le nombre d'itérations de *BP*. Les deux premières solutions observent les variations des marginaux afin de déterminer si les variables sont stables. La troisième observe les variations de l'entropie des variables. La seconde partie explique les schémas utilisés pour pondérer les messages lors de la *BP*. Ces schémas sont basés sur l'arité des contraintes et sur les conflits que ces dernières provoquent. Enfin, la troisième partie présente trois nouvelles heuristiques. Elles exploitent d'une nouvelle manière les marginaux en les utilisant pour calculer l'entropie des variables. La première choisit de brancher en priorité les variables ayant une faible entropie, tandis que la seconde branche de manière à réduire le plus possible l'entropie du modèle. La dernière est une combinaison des deux autres : elle passe de la seconde à la première lorsque l'entropie du modèle descend en dessous d'un certain seuil.

Les résultats expérimentaux sont présentés dans le chapitre 4. Ils montrent que les critères d'arrêt dynamiques peuvent induire une réduction significative du temps de calcul. Ils montrent aussi que nos schémas de pondérations permettent de réduire significativement le nombre d'échecs rencontrés au cours de la recherche pour certains problèmes. Finalement, ils démontrent que choisir la variable avec la plus faible entropie est une meilleure exploitation des distributions marginales que *max-marginal*. De plus, cela permet à notre approche d'être compétitive avec l'heuristique de l'état de l'art *dom/wdeg*.

Enfin, le cinquième et dernier chapitre contient la synthèse des travaux réalisés, leurs limitations et une ouverture sur les possibles améliorations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

La programmation par contraintes permet d'exploiter la structure du problème afin de réduire considérablement l'espace de recherche. Concrètement, à chaque branchement effectué dans l'arbre de recherche, les contraintes sont propagées afin de filtrer, dans les domaines de chacune des variables, les valeurs impossibles. Mais l'ampleur de ce filtrage, et donc les performances de la recherche, est grandement impactée par l'ordre dans lequel sont fixées les variables. Donc la stratégie de recherche est très importante. Les heuristiques d'ordonnancement de variables peuvent être classées en deux catégories : les stratégies statiques et les stratégies dynamiques. Les premières ordonnent les valeurs selon des valeurs qui ne changeront pas au cours de la recherche. L'ordre des valeurs y est donc prédéfini. Parmi elles, on peut noter l'heuristique *minimum width* (*minw*) [12] qui ordonnance les variables de manière à minimiser la largeur du graphe des contraintes. Ou encore, l'heuristique *maximum degree* (*deg*) [13] qui ordonne les variables de manière décroissante par rapport à leur degré. Le degré d'une variable étant son nombre de voisins dans le graphe des contraintes, i.e. le nombre de variables avec lesquelles elle a au moins une contrainte en commun. Les stratégies dynamiques, quant à elles, choisissent dynamiquement la prochaine variable à fixer à chaque nœud de l'arbre de recherche. L'heuristique *dom* décrite dans la section 1.1.4 appartient à cette catégorie. Cette revue de littérature présente différentes heuristiques de l'état de l'art et se concentre sur les heuristiques dynamiques.

2.1 Améliorations de *dom*

Au cours du chapitre 1, nous avons évoqué l'heuristique *dom*. Si cette heuristique est moins utilisée, elle a quand même eu quelques améliorations en la combinant avec d'autres heuristiques. Par exemple, *dom/deg* [14] choisit la variable qui minimise le ratio entre la taille de son domaine et son degré. Le degré d'une variable est le nombre de contraintes qui lui sont appliquées.

Mais la meilleure combinaison est *dom/wdeg* [9], qui ajoute la prise en compte des conflits. L'heuristique garde en mémoire pour chaque contrainte un compteur du nombre de fois où celle-ci a été à l'origine d'un échec. On calcule alors pour les variables un degré pondéré *wdeg*. Ce nouveau degré est la somme des compteurs de chaque contrainte appliquée à la variable. Ainsi, les contraintes à l'origine du plus d'échecs ont un poids plus important que les autres. *Dom/wdeg* choisit la variable minimisant le ratio entre la taille de son domaine et son degré pondéré. Ce *weighted degree* a permis de donner de meilleures performances que *dom*. En plus

des bonnes performances, cette heuristique est simple à implémenter dans un solveur. Ainsi, *dom/wdeg* est une heuristique très importante dans l'état de l'art et très souvent utilisée comme référence dans des articles ou comme heuristique par défaut dans plusieurs solveurs (par exemple les solveurs Choco [1] et Abscon [2]).

2.2 Impact-Based Search

Impact-Based Search [15] (*IBS*) choisit la prochaine variable à fixer en se basant sur une estimation de l'impact qu'elle aura sur l'espace de recherche. Cette estimation est calculée à partir des impacts mesurés au cours des branchements précédents. À chaque affectation $x_i = a$, on calcule le taux de réduction pour cette affectation :

$$I(x_i = a) = 1 - \frac{P_{after}}{P_{before}} \tag{2.1}$$

avec P_{after} le produit des tailles des domaines de toutes les variables après l'affectation, et P_{before} le même produit avant l'affectation.

Pour chaque paire (x_i, a) on calcule la moyenne des taux de réduction mesurés afin d'avoir une meilleure estimation de l'impact :

$$\bar{I}(x_i = a) = \frac{\sum_{k \in K} I^k(x_i = a)}{|K|} \tag{2.2}$$

avec K l'ensemble des indices des impacts mesurés pour la paire (x_i, a).

Finalement on peut calculer l'impact de la variable x_i :

$$I(x_i) = \sum_{a \in D'_{x_i}} 1 - \bar{I}(x_i = a) \tag{2.3}$$

avec D'_{x_i} le domaine actuel de la variable x_i.

Pour suivre le *Fail-First Principle*, *IBS* choisit la variable x_i qui maximise $I(x_i)$. Et pour cette variable, l'heuristique choisit la valeur a qui minimise $\bar{I}(x_i = a)$.

Cette heuristique "apprenant" les impacts au cours de la recherche, elle nécessite d'utiliser des relances afin d'obtenir de bonnes performances. De même, initialiser les impacts avant la recherche améliore de manière significative l'efficacité de l'heuristique. Pour ce faire, le

1. Disponible à https ://github.com/chocoteam/choco-solver/releases/tag/4.10.2
2. Disponible à https ://github.com/xcsp3team/ace

domaine D_{x_i} de chaque variable x_i est divisé en k sous-domaines. L'impact d'une seule valeur par sous-domaine est alors calculé. Pour chaque valeur a appartenant à ce sous-domaine, l'impact est :

$$I(x_i = a) = 1 - \frac{1 - I(x_i \in D^w_{x_i})}{D^w_{x_i}} \; \forall a \in D^w_{x_i} \tag{2.4}$$

Ainsi, il n'est pas nécessaire de calculer l'impact de chaque valeur de chaque variable, évitant ainsi un surcoût en temps de calcul trop important.

Du fait de ses très bonnes performances *IBS* a été pendant longtemps une heuristique importante de l'état de l'art.

2.3 Counting-Based Search

Certaines affectations sont dans une proportion plus importante de solutions valides que d'autres. Calculer cette proportion permettrait d'obtenir une puissante information pour les décisions de branchement. C'est sur cette idée que repose la *Counting-Based Search* [16]. Cette approche choisit les paires *(variable, valeur)* en fonction d'une mesure appelée *densité de solution*. Elle est définie de la manière suivante :

$$\sigma(x_i, d, c) = \frac{\#c(x_1, ..., x_{i-1}, d, x_{i+1}, ..., x_n)}{\#c(x_1, ..., x_n)} \tag{2.5}$$

avec x_i une variable, d une valeur, c une contrainte et $\#c(x_1, ..., x_n)$ le nombre de *n-tuplets* satisfaisant c. La densité de solution $\sigma(x_i, d, c)$ représente donc la proportion de solutions à la contrainte c contenant l'affectation $x_i = d$. Comme expliqué dans le chapitre 1, les probabilités marginales calculées grâce à la *BP* représentent pour chaque affectation la proportion de solutions valides contenant cette affectation. Ainsi, ces marginaux sont une version généralisée de la densité de solution, dans le sens où ils ne sont plus spécifiques à une contrainte, mais au problème dans sa globalité. Le travail présenté dans ce mémoire s'inscrit donc dans la continuité de la *Couting-Based Search*.

Calculer ces densités de solutions requiert des algorithmes spécifiques à chaque contrainte. De tels algorithmes ont déjà été développés pour plusieurs contraintes [16–20], il est donc possible d'exploiter ces densités pour un large panel de problèmes différents. Plusieurs heuristiques ont été conçues dans ce but, mais celle donnant les meilleurs résultats est *maxSD*. Cette heuristique choisit la paire (x_i, d) présentant pour une contrainte la plus grande densité de solution, toutes contraintes confondues. Elle a démontré des performances supérieures à *IBS*

et *dom/wdeg* sur plusieurs problèmes différents et calculer les densités de solutions ne semble pas introduire de surcoût conséquent sur le temps de calcul. Mais parce cette approche est axée sur la faisabilité, elle ne performe pas aussi bien sur les problèmes d'optimisation. Pour répondre à cela, [21] introduit une nouvelle définition de la densité de solution basée sur le coût et des algorithmes pour la calculer pour plusieurs contraintes. Avec cette approche, on ne calcule plus la densité sur toutes les solutions valides, mais seulement sur les solutions dont le coût est suffisamment proche de l'optimal. *maxSD**, la version de *maxSD* utilisant cette nouvelle densité, a démontré de très bonnes performances en trouvant plus de solutions optimales que *IBS* et *dom* sur 141 instances de trois problèmes d'optimisations différentes. Par contre, cette approche a pour défaut de demander un effort plus grand pour l'implémenter dans un solveur à cause des algorithmes de dénombrement spécifique à chaque contrainte.

2.4 Activity-Based Search

Activity-Based Search [22] (*ABS*) examine l'activité des variables au cours de la recherche. On entend par activité, la fréquence à laquelle une variable voit son domaine être réduit lorsque d'autres variables sont fixées. À chaque branchement au cours de la recherche, l'activité de chaque variable x est mise à jour selon les deux règles suivantes :

$$\begin{cases} \forall x \in X s.t. |D(x)| > 1 : A(x) = A(x).\gamma \\ \forall x \in X' : A(x) = A(x) + 1 \end{cases} \tag{2.6}$$

où $A(x)$ est l'activité de la variable x, X est l'ensemble des variables, X' est le sous-ensemble de X contenant les variables dont le domaine a été réduit et γ est un paramètre de décrépissage avec $0 \leq \gamma \leq 1$. Le décrépissage ne s'applique qu'aux variables qui ne sont pas fixées, car les variables ayant été fixées tôt verraient leur activité tomber rapidement à 0.

ABS sélectionne la variable x maximisant le rapport $\frac{A(x)}{|D(x)|}$. Ce rapport permet d'éviter de favoriser les variables ayant un grand domaine de valeurs comme le ferait la valeur $A(x)$ seule.

ABS permet aussi de sélectionner l'affectation $x = a$ en plus de la variable x. Pour cela il faut introduire l'activité d'une affectation :

$$A_k(x = a) = |X'| \tag{2.7}$$

Cette activité est spécifique au nœud k de l'arbre de recherche. Elle correspond donc simplement au nombre de variables affectées par l'affectation $x = a$. Il est alors possible de calculer une estimation de l'activité de l'affectation pour tout l'arbre de recherche $\tilde{A}(x = a)$. Plutôt

que d'utiliser une moyenne, c'est une somme pondérée qui est utilisée :

$$\tilde{A}_1(x = a) = \frac{\tilde{A}_0(x = a).(\alpha - 1) + A_k(x = a)}{\alpha} \tag{2.8}$$

où α est un paramètre permettant de choisir le poids qu'à la nouvelle observation $A_k(x = a)$.

D'une manière similaire à IBS, ABS choisit la valeur avec la plus faible activité.

Sur un banc d'essai contenant 5 types de problèmes différents, ABS a démontré être bien plus robuste que IBS tout en apportant une amélioration des performances.

2.5 Conflict History Based Search

L'idée de la *Conflict History Based Search* [23] est de s'intéresser à l'historique des contraintes impliquées dans des conflits.

À chaque contrainte c_i correspond une valeur récompense $r(c_i)$. Elle est donnée par la formule :

$$r(c_i) = \frac{1}{\#Conflicts - Conflicts(c_i) + 1} \tag{2.9}$$

où $\#Conflicts$ est le nombre de conflits enregistrés depuis le début de la recherche et $Conflicts(c_i)$ correspond à la valeur de $\#Conflicts$ la dernière fois que c_i a été à l'origine d'un échec. Ainsi, une contrainte qui échoue régulièrement sur de petites périodes va obtenir une récompense plus élevée.

Cette récompense est utilisée dans le calcul du score $q(c_i)$ de chaque contrainte. À chaque fois qu'une variable est à l'origine d'un échec, son score est incrémenté selon la formule suivante :

$$q(c_i) = (1 - \alpha) \times q(c_i) + \alpha \times r(c_i) \tag{2.10}$$

avec $0 < \alpha < 1$ et $r(c_i)$ la valeur récompense. Afin d'accorder plus d'importance aux échecs récents, la valeur de α est décrémentée au fur et à mesure de la recherche.

$$chv(x_i) = \frac{\sum_{c_j \in C | x_i \in X(c_j) \wedge |Uvars(c_i)|>1} (q(c_j) + \delta)}{|D_i|} \tag{2.11}$$

avec $X(c_j)$ l'ensemble des variables soumises à c_j, $Uvars(c_j)$ l'ensemble des variables non

fixées de $X(c_j)$ et δ un entier positif proche de 0. L'heuristique CHS va choisir la variable x_i qui maximise $chv(x_i)$, et donc choisir les variables liées aux contraintes le plus fréquemment impliquées dans des conflits. Le dénominateur de cette formule permet de privilégier les variables avec des domaines de petite taille en accord avec le *Fail-First Principle*. La valeur δ est utile pour le début de la recherche, lorsque toutes les variables ont un score de 0. Grâce à cette valeur, le premier branchement est basé sur le degré de la variable et non sur un choix aléatoire.

Comme *Impact-Based Search* et *Activity-Based Search*, cette heuristique nécessite l'utilisation de relances au cours de la recherche. Lors des relances, la valeur de α est réinitialisée, contrairement aux valeurs $Conflicts(c_i)$.

CHS a résolu plus d'exemplaires que *dom/wdeg* et *ABS* sur un banc d'essai contenant 14 types de problèmes différents. Mais avec un temps de calcul bien plus élevé pour certains problèmes.

2.6 Failure-Directed-Search

La *Failure-Directed-Search* [24] part du principe qu'il n'y a pas de solution, ou qu'elle est au moins très difficile à trouver. En partant de ce postulat, elle ne se concentre que sur le fait d'échouer le plus rapidement possible.

À chaque noeud de l'arbre de recherche cette heuristique ne va pas soit fixer une variable sur une valeur, soit retirer cette valeur du domaine de la variable, contrairement aux stratégies vues jusqu'ici. À la place, elle va séparer le domaine de la variable en deux intervalles disjoints. Cet approche a été pensée avec les problèmes de planification en tête. Dans ce type de problèmes, la plupart des contraintes utilisent des cohérences de bornes. Ainsi, retirer une valeur d'un domaine n'engendrera pas de filtrage conséquent. En séparant le domaine en deux intervalles, on résout ce problème.

Chaque choix de branchement c possible dispose de deux branches. Étant donné qu'il n'y a pas d'ordre déterminé à l'avance dans lequel il faudrait exploré ces deux branches, on ne parle pas de branche gauche et de branche droite, mais plutôt de branche positive et de branche négative.

Chaque branche dispose d'une note qui va servir à guider l'heuristique :

- rating$^+[c]$ pour la branche positive
- rating$^-[c]$ pour la branche négative

À l'instar d'heuristique comme *IBS* ou *ABS*, à chaque fois qu'une branche est explorée, sa note va être mise à jour. Tout d'abord une note locale va être calculée :

$$\text{localRating} := \begin{cases} 0 & \text{si la branche échoue directement} \\ 1 + R & \text{sinon} \end{cases} \tag{2.12}$$

où R est une valeur quantifiant la réduction de l'espace de recherche. Il est par exemple possible d'utiliser :

$$R = \frac{P_{after}}{P_{before}} \tag{2.13}$$

avec P_{after} le produit des tailles des domaines de toutes les variables après l'affectation, et P_{before} le même produit avant l'affectation. Grâce à cette note locale, la note de la branche explorée va être mise à jour selon la formule suivante :

$$\text{rating}^{+/-}[c] := \alpha \cdot \text{rating}^{+/-}[c] + (1 - \alpha) \cdot \frac{\text{localRating}}{\text{avgRating}[d]} \tag{2.14}$$

où α est un paramètre souvent compris entre 0.9 et 0.99 et avgRating[d] est la moyenne des notes des choix possibles à la profondeur d de l'arbre de recherche. Avec les notes de chaque branche d'un choix c, il est possible de calculer sa note :

$$\text{rating}[c] = \text{rating}^{+}[c] + \text{rating}^{-}[c] \tag{2.15}$$

Avec cette définition, plus rating[c] est faible, plus ce choix a de chances de nous conduire à des échecs. Ainsi, l'heuristique va choisir les choix minimisant cette note.

Cette heuristique n'est pas faite pour être utilisée seule, mais plutôt lorsqu'une autre stratégie n'est plus capable d'améliorer une solution. Elle donne par exemple de très bon résultats lorsqu'elle est combinée à une *Large Neighborhood Search* auto-adaptative [25].

CHAPITRE 3 AMÉLIORATION DE L'EXPLOITATION DE LA PROPAGATION DE CROYANCES

Ce chapitre détaille la contribution apportée par ce mémoire. Dans un premier temps, nous présentons deux critères ayant pour but de déterminer quand arrêter les itérations de *BP*. Le premier observe la stabilité du plus grand marginal et le deuxième observe les variations de l'entropie. Puis nous présentons des approches pour pondérer les messages des contraintes lors de la *BP*, afin d'obtenir des distributions plus proches des distributions réelles. Nos approches sont basées sur l'arité des contraintes et sur le nombre de conflits dont elles sont à l'origine. Enfin nous présentons deux nouvelles heuristiques. La première sélectionne la variable possédant la plus faible entropie. La seconde fixe les variables étant censées réduire le plus l'entropie du modèle.

3.1 Critères d'arrêt dynamiques

Dans la situation actuelle du système, la *BP* est répétée un nombre fixe d'itérations. Cette situation n'est pas avantageuse : un même nombre fixe ne sera pas forcément optimal d'un problème à un autre. Ainsi, selon le problème, il faudra potentiellement faire plusieurs tests afin de trouver une valeur donnant de bons résultats. De plus, au fur et à mesure de la recherche, le graphe du problème change fortement. Ainsi, les marginales ne convergeront pas à la même vitesse. On risque alors de brancher trop tôt, et donc de faire de moins bons branchements, ou de brancher trop tard, ce qui n'est pas optimal en termes de temps de calcul. Il est donc important de trouver un critère afin de déterminer dynamiquement quand il est bon de stopper la *BP* et de brancher.

3.1.1 Critère basé sur la stabilité du meilleur marginal

Instinctivement, nous pouvons considérer qu'il est bon d'arrêter la *BP* lorsque toutes les marginales ont convergé. Mais ce n'est pas forcément nécessaire : l'heuristique *max-marginal* sélectionne la paire variable-valeur présentant le marginal le plus élevé, peu importe sa valeur. Ainsi, nous pouvons considérer qu'il est bon de brancher lorsque la paire à sélectionner ne fait plus de doute. Si pendant plusieurs itérations d'affilée, la paire présentant le plus haut marginal reste la même, il est très probable que le marginal en question reste le meilleur, ou au moins l'un des plus élevés, pendant les itérations suivantes. Sur cette intuition, nous avons conçu le critère suivant : Après chaque itération nous observons quelle est l'affectation qui

possède le plus grand marginal, que nous noterons θ^*. Si, pendant un nombre n d'itérations consécutives de BP, c'est la même paire qui possède θ^*, alors la BP est stoppée et une décision de branchement est prise.

Le problème de cette approche est qu'elle est trop rigide. Si au cours d'une des n itérations consécutives, la paire en question voit son marginal devenir le deuxième plus grand même en ayant un écart très faible avec θ^*, la BP ne sera pas stoppée alors qu'il pourrait être intéressant de fixer cette paire. Il est donc bon de ne pas considérer uniquement le meilleur marginal, mais toutes les meilleures marginales dont la valeur est suffisamment proche de θ^*. Le critère d'arrêt est donc de stopper la BP après l'itération i si $\exists x \in X, v \in V$ tel que pendant n itérations consécutives on a :

$$\theta_x(v) \geq \gamma \times \theta^* \tag{3.1}$$

avec $\gamma \in [0, 1]$. Lorsque $\gamma = 1$, il faut que $\theta_x(v)$ soit le plus grand marginal pendant les n itérations consécutives. À l'inverse, lorsque $\gamma = 0$, la BP est stoppée après n itérations car toutes les marginales respectent le critère.

Une variable qui verrait son meilleur marginal grimper rapidement et dépasser les autres marginales à la dernière itération ne peut pas être considérée comme stable et il ne faudrait pas la fixer. Mais dans ce cas, si le critère a décidé l'arrêt de la BP, c'est qu'il y a tout de même une variable qu'il pourrait être bon de fixer. Par exemple, une variable dont le meilleur marginal était θ^* pendant $n - 1$ itérations et qui devient deuxième à l'itération n, tout en ayant un écart minime avec le plus élevé. Pour prendre en compte ce cas de figure, nous restreignons la liste de variables candidates au branchement aux seules variables dont l'une des probabilités marginales respecte le critère.

Pour ne pas rester coincés dans le cas où aucune variable ne réussit à satisfaire le critère pendant un nombre élevé d'itérations, nous ajoutons une limite au nombre d'itérations avant un branchement. Dans le cas où cette limite est atteinte, n'importe quelle variable peut être sélectionnée par l'heuristique. Afin de limiter le temps de calcul, nous ne regardons pour chaque variable que son meilleur marginal. Nous observons aussi la valeur associée, si elle change au cours des n itérations, alors sa variable ne peut pas être considérée comme bonne à fixer.

La procédure est formalisée dans l'algorithme 1.

Algorithme 1 Belief Propagation avec le Critère basé sur la stabilité du meilleur marginal

$X \leftarrow$ ensemble des variables non fixées
$maxItérations \leftarrow$ nombre maximal d'itérations de BP
$i \leftarrow 1$
$variablesCandidates \leftarrow []$
$compteurs \leftarrow [0$ pour chaque $x \in X]$
$v^* \leftarrow [\emptyset$ pour chaque $x \in X]$ ▷ Contient les valeurs associées au meilleur marginal de chaque variable
Tant que $variablesCandidates$ est vide ET $i < maxItérations$, **faire** :
    $Distributions \leftarrow BeliefPropagationIteration()$
    $\theta^* \leftarrow max(Distributions)$
    Pour $x \in X$, **faire** :
        $\theta \leftarrow max(Distributions[x])$
        $v \leftarrow valeurAvecMeilleurMarginal(x)$
        Si $v^*[x] == v$ ET $\theta \geq \gamma * \theta^*$ **alors**
            $compteurs[i] + +$
        Sinon
            $v^*[x] \leftarrow v$
            $compteurs[i] \leftarrow 1$
        Fin si
        Si $compteurs[i] == n$ **alors**
            $variablesCandidates.ajout(i)$
        Fin si
    Fin pour
    $i + +$
Fin tant que
Si $variablesCandidates$ est vide **alors**
    $variablesCandidates \leftarrow X$
Fin si
renvoyer $Distributions, variablesCandidates$

3.1.2 Critère basé sur les variations de l'entropie

La *Belief Propagation* nous permettant de calculer les distributions marginales de chaque domaine, cela nous permet aussi d'obtenir l'entropie des variables. Nous calculons l'entropie $H(x)$ de chaque variable x avec la formule de l'entropie de Shannon [11] :

$$H(x) = -\sum_{v \in D(x)} \theta_x(v) \log(\theta_x(v)) \tag{3.2}$$

Les variations de l'entropie des variables nous renseignent sur les variations de leurs probabilités marginales. En effet, si l'entropie d'une variable subit d'importantes variations d'une itération de *BP* à une autre, ses probabilités marginales subissent aussi une importante variation. Il n'est alors pas bon d'arrêter la *BP* pour fixer une variable.

L'entropie du modèle peut être calculée comme étant la moyenne des entropies des variables :

$$\overline{H} = \frac{\sum_{x \in X} H(x)}{|X|} \tag{3.3}$$

où $H(x)$ est l'entropie de la variable x.

Nous arrêtons alors la *BP* lorsque $0 \leq \overline{H}' - \overline{H} < \alpha$ où $\overline{H'}$ est l'entropie à l'itération précédente de *BP* et $\alpha > 0$ est un seuil fixé par l'utilisateur. Cette différence doit être positive, afin d'éviter d'arrêter la *BP* lorsque l'entropie commence à remonter.

Cette approche possède un défaut : $\overline{H}$ est très dépendant de la taille des domaines des variables. Ainsi, en fin de recherche, cette valeur risque d'être bien plus faible qu'au début. Les variations seront alors elles aussi plus faibles, et un seuil pertinent en début de recherche risque de perdre en pertinence.

Il faut donc changer le calcul de $\overline{H}$ pour le rendre invariable à la taille des domaines :

$$\overline{H} = \frac{\sum_{x \in X} \frac{H(x)}{\log(|D(x)|)}}{|X|} \tag{3.4}$$

Dans cette formule, l'entropie des variables est divisée par le logarithme de la taille de leur domaine. L'entropie d'une distribution uniforme, et donc l'entropie maximale d'une variable, est $\log(d)$ avec d le nombre de valeurs. Ainsi, cette division permet à la moyenne d'être comprise entre 0 et 1, indépendamment de la taille des domaines.

Algorithme 2 Belief Propagation avec le critère basé sur les variations de l'entropie

$arrêt \leftarrow Faux$
$i \leftarrow 1$
$\overline{H} \leftarrow 0$
$\overline{H'} \leftarrow \infty$
$maxItérations \leftarrow$ nombre maximal d'itérations de BP
Tant que $arrêt == Faux$ ET $i < maxItérations$, **faire** :
    $Distributions \leftarrow BeliefPropagationIteration()$
    $\overline{H} \leftarrow Entropie(Distributions)$
    Si $\overline{H'} - \overline{H} \in [0, \alpha]$ **alors**
        $arrêt \leftarrow Vrai$
    Fin si
    $\overline{H'} \leftarrow \overline{H}$
    $i++$
Fin tant que
renvoyer $Distributions$

3.2 Pondération des messages

Avec la *Belief Propagation*, tous les messages ont le même impact, peu importe la variable ou la contrainte d'où ils viennent. Pourtant toutes les contraintes n'ont pas le même impact au cours de la recherche. Par exemple des contraintes vont impacter plus de variables que d'autres, ou encore vont être à l'origine de plus d'échecs. Pondérer les messages venant des contraintes en fonction de ces paramètres pourrait accélérer la convergence.

3.2.1 Arité

Une contrainte liée avec beaucoup de variables, i.e. une contrainte avec une forte arité, possède une meilleure "connaissance" du problème. En effet, l'information contenue dans ses messages est plus pertinente, car issue d'une inférence sur un plus grand nombre de variables. Accorder plus d'importance aux messages venant de ces contraintes peut donc permettre aux marginales de converger vers des valeurs plus proches des véritables probabilités. On accorde à chaque contrainte $c \in C$ un poids w_c calculé à partir de son arité :

$$w_c = 1 + \frac{|N(c)| - min_{c \in C}|N(c)|}{|X|} \tag{3.5}$$

Dans cette formule l'arité est divisée par le nombre de variables du problème afin de normaliser le poids. Ainsi, $1 < w_c \leq 2, \forall c \in C$. Au dénominateur, au lieu de l'arité de la contrainte, nous avons une soustraction en l'arité de la contrainte et la plus faible arité de toute les contraintes.

De cette manière, pour les problèmes où toutes les contraintes ont la même arité, toutes les contraintes auront un poids de 1. Ce poids est ensuite utilisé pour pondérer les probabilités marginales des variables. Les messages des contraintes deviennent

$$\mu'_{c\to x}(v) = (\mu_{c\to x}(v))^{w_c} \tag{3.6}$$

avec w_c le poids de la contrainte c, $\mu_{c\to x}(v)$ le message de la contrainte c à la variable x, v une valeur du domaine de x. La *BP* met à jour les distributions marginales en faisant le produit des messages des contraintes. Mettre le poids d'une contrainte en exposant de ses messages permet donc d'augmenter l'importance de ces derniers lors du calcul des distributions.

Nous proposons aussi une pondération opposée :

$$w_c = 1 - \frac{|N(c)| - min_{c\in C}|N(c)|}{|X|} \tag{3.7}$$

Ce schéma va donc donner plus de poids aux contraintes ayant une faible arité.

3.2.2 Nombre d'échecs

Certaines contraintes sont à l'origine de plus d'échecs que d'autres. Ces contraintes jouent donc un rôle plus important dans le filtrage des valeurs et l'inférence réalisée par ces contraintes a donc un fort impact sur la résolution. Ainsi, l'inférence réalisée par ces contraintes dans la *BP* est potentiellement plus intéressante : les privilégier pourrait donner des marginales plus proches des distributions réelles.

Chaque contrainte $c \in C$ dispose d'un compteur d'échecs f_c. À chaque échec, la contrainte qui en est à l'origine voit son compteur être incrémenté de 1. Les compteurs sont utilisés pour calculer les poids des contraintes :

$$w_c = 1 + \frac{f_c}{\sum_{c'\in C} f_{c'}} \tag{3.8}$$

avec w_c le poids de la contrainte c. Le dénominateur correspond au nombre total d'échecs ; il permet ainsi de normaliser les poids et de s'assurer que $1 \leq w_c \leq 2, \forall c \in C$.

Là encore, nous proposons aussi une version opposée de la pondération :

$$w_c = 2 - \frac{f_c}{\sum_{c'\in C} f'_c} \tag{3.9}$$

3.3 Heuristiques

3.3.1 Min-entropy

L'entropie introduite à la section 3.1.2 est une donnée très intéressante pour choisir quelle variable devrait être fixée en priorité. En effet, elle nous permet d'estimer l'incertitude que nous avons à propos d'une variable : si une variable possède une valeur avec une marginal très élevé, qui se démarque des autres, alors son entropie sera faible. À l'inverse, une variable n'ayant que des marginales similaires aura une entropie élevée. Ainsi, plus l'entropie d'une variable est faible, plus la connaissance que l'on a à propos de la valeur qu'elle devrait prendre est forte.

Si nous reprenons l'exemple développé dans la section 1.2, nous pouvons calculer l'entropie de chaque variable à partir de leurs distributions. Ces entropies sont présentées dans le tableau 3.1 avec les distributions correspondantes. La distribution de la variable c est celle faisant le plus ressortir une valeur par rapport aux autres. C'est ce que permet de quantifier l'entropie. Pour cette variable, elle est plus faible que pour les autres où les distributions sont moins discriminantes.

Ainsi plus une variable possède une entropie faible, plus il est intéressant de la fixer en priorité. L'heuristique *min-entropy* choisit de fixer la variable ayant la plus faible entropie et choisit la valeur avec la marginale la plus élevée.

Il est intéressant de noter que, dans le cas où les distributions marginales sont uniformes (il n'y a donc pas d'information discriminante entre les valeurs d'un domaine), la variable avec la plus faible entropie est celle disposant du plus petit domaine. Ainsi *min-entropy* peut être vue comme une généralisation de l'heuristique *dom*.

Tableau 3.1 Distributions marginales de l'exemple de la section 1.2 avec les entropies déduites de chaque distribution

	1	2	3	4	Entropie
θ_a	.01	.52	.46	.01	.79
θ_b	.01	.52	.46	.01	.79
θ_c	.98	.02	.00	.00	.09
θ_d	.90	.10	.00	.00	.33

3.3.2 Impact-entropy

Si nous reprenons la définition de l'entropie du modèle $\overline{H}$ de la section 3.1.2, cette entropie est nulle soit lorsqu'une solution est trouvée, soit lorsqu'un échec survient dans la recherche. Ainsi, notre but peut être vu comme étant de chercher à réduire à zéro $\overline{H}$ le plus vite possible. L'idée d'*impact-entropy* est de sélectionner les variables les plus susceptibles de réduire fortement l'entropie. Cette heuristique fonctionne d'une manière similaire à *Impact-Based Search* [15] : en observant les impacts de chaque affectation et en se servant de ces observations pour estimer l'impact d'une variable.

Impact d'une affectation

À chaque affectation $x = a$, l'impact $I(x = a)$ sur $\overline{H}$ est enregistré :

$$I(x = a) = 1 - \frac{\overline{H}_{after}}{\overline{H}_{before}} \tag{3.10}$$

où $\overline{H}_{before}$ et $\overline{H}_{after}$ sont respectivement les entropies du modèle avant et après le branchement.

Selon cette formule, une affectation conduisant à un échec a un impact de 1, tandis qu'une affectation réduisant très peu l'entropie a un impact proche de 0. Les entropies sont mesurées après un nombre d'itérations de *Belief Propagation* en accord avec le critère d'arrêt. De cette manière, nous avons des valeurs d'entropies plus pertinentes, car plus proches des entropies calculées à partir des distributions réelles.

Mais se baser uniquement sur l'impact d'une affectation à un nœud de l'arbre en particulier n'est pas assez précis. Nous estimons l'impact qu'aura l'affectation en calculant la moyenne de ses impacts mesurés :

$$\overline{I}(x = v) = \frac{\sum_{k \in K} I^k(x = v)}{|K|} \tag{3.11}$$

où K est l'ensemble des mesures d'impacts pour l'affectation $x = v$. De cette manière, l'estimation de l'impact d'une affectation devient plus juste chaque fois qu'elle est choisie dans l'arbre de recherche.

Impact d'une variable

Les impacts des affectations sont utiles pour calculer l'impact que l'on peut espérer en fixant une variable. Dans un premier temps nous pouvons considérer que l'impact d'une variable peut s'estimer comme la moyenne des impacts des valeurs de son domaine.

$$I(x) = \frac{\sum_{v \in D'_x} \overline{I}(x = v)}{|D'_x|} \tag{3.12}$$

où $D'_x \subset D_x$ représente les valeurs qui n'ont pas été retirées du domaine au nœud courant. En effet, comme nous cherchons à estimer l'impact au nœud courant, prendre en compte les valeurs déjà filtrées et qui ne pourront donc pas être sélectionnées est incorrect.

Impact-based Search [15] n'utilise pas cette moyenne. C'est plutôt une estimation de la taille de l'espace de recherche en pire cas (c'est-à-dire lorsque le problème n'est pas satisfaisable et que toutes les valeurs de la variable sont essayées) qui est utilisée :

$$I'(x) = \sum_{v \in D'_x} (1 - \overline{I}(x = v)) \tag{3.13}$$

Cette approche fait sens lorsque l'on observe l'impact sur la taille des domaines, car cette valeur est directement liée à la taille de l'espace de recherche. En effet, si l'on se place dans le cas où le sous-arbre n'a pas de solution, $I'(x)$ représente un gain de temps de calcul, car il quantifie les branches que l'algorithme n'aura pas à explorer. Et ce qui rend cette estimation du gain plus intéressante que $I(x)$ est que, lorsque l'on sort du sous-arbre en question, on ne perdra pas ce gain. Car il est gagné sur l'ensemble de l'espace de recherche. Alors que la taille des domaines, elle, remontera avec le *backtracking*.

L'entropie n'a pas ce lien avec la taille de l'espace de recherche : comme nous pouvons le constater dans l'exemple de la section 3.3.1, des variables avec des domaines de même taille peuvent avoir des entropies très différentes. Ainsi, une forte réduction de l'entropie ne provoque pas forcément une forte réduction de l'espace de recherche. Calculer $I'(x)$ ne fait donc pas de sens dans notre cas, étant donné qu'il ne représente pas de gain sur l'ensemble de la recherche. Dans les faits, pour *impact-entropy*, $I'(x)$ a montré de plus mauvais résultats que calculer $I(x)$.

Une autre approche possible est de calculer l'espérance de l'impact de la variable dans le cas où le sous-arbre choisi contient bien une solution. L'impact d'une variable devient alors :

$$I''(x) = \sum_{v \in D'_x} \theta_x(v) \times \overline{I}(x = v) \tag{3.14}$$

Cette approche permet de donner plus d'importances aux impacts des affectations qui seront les plus susceptibles d'être sélectionnés en premier par l'heuristique. En effet, si *impact-entropy* choisit la variable qui maximise $I''(x)$, elle choisit pour cette variable la valeur possédant le marginal le plus élevé.

Initialisation des impacts et redémarrages

Les décisions de branchement au début de l'arbre de recherche sont cruciales, car elles ont un très fort impact sur les performances. Or *impact-entropy* exploite des impacts observés au cours de la recherche et qui ne sont donc pas normalement utilisables au départ de cette recherche. Il est donc important d'initialiser les impacts avant le début de la recherche afin que l'heuristique prenne des décisions de branchement pertinentes dès le départ. La solution la plus simple est de tester l'impact de chaque affectation possible avant la recherche. Cette approche assure d'avoir de bonnes estimations des impacts, mais implique un surcoût de calcul conséquent avant de débuter la recherche.

Plus une affectation a été testée, plus l'estimation de son impact est précise. Mais il faut donc remonter dans l'arbre jusqu'à un nœud antérieur à cette affectation pour pouvoir affiner son impact et aussi pour pouvoir bénéficier d'un impact affiné. Ainsi l'affinage des impacts au cours de la recherche n'est que très peu exploité pour les décisions en haut de l'arbre. De même, les variables fixées dans les nœuds les plus hauts ne voient leur impact être ajusté que rarement. Afin de résoudre ce problème, l'usage de redémarrages au cours de la recherche est préconisé. Lorsque la recherche atteint un nombre κ d'échecs, la recherche est redémarrée. À chaque redémarrage, la valeur κ est multipliée par un facteur ψ.

3.3.3 Impact-Min-entropy

Comme expliqué dans la section précédente, une faible entropie du modèle n'induit pas forcément un espace de recherche restreint. Mais cela induit une faible entropie de plusieurs variables. Des variables présentant donc des distributions marginales plus discriminantes. Et donc potentiellement de meilleures performances pour une heuristique telle que *min-entropy*.

L'idée d'*impact-min-entropy* est donc la suivante : *impact-entropy* est utilisée en début de recherche afin de réduire le plus vite possible l'entropie du modèle. L'heuristique sert donc à "nettoyer" le modèle, en permettant de désuniformiser les distributions marginales. L'informa-

tion fournie par la *BP* est alors plus puissante. *Min-entropy*, contrairement à *impact-entropy*, tire directement bénéfice de ces informations. Elle peut donc montrer de meilleures performances. Ainsi, une fois que $\overline{H}$ passe sous un certain seuil ρ, *min-entropy* est utilisée afin de profiter de l'entropie réduite.

CHAPITRE 4 RÉSULTATS THÉORIQUES ET EXPÉRIMENTAUX

Ce chapitre présente nos résultats expérimentaux. Dans un premier temps, nous optimisons les paramètres liés à *impact-entropy* et *impact-min-entropy* afin d'en obtenir les meilleures performances possible. Puis nous comparons toutes nos heuristiques avec *max-marginal*, *max-marginal-strength* et *dom/wdeg* afin de placer leurs performances dans l'état de l'art. Ainsi nous pourrons répondre à notre première question de recherche :

Q1 : Est-ce qu'exploiter l'entropie est une approche intéressante pour exploiter les distributions marginales ?

Enfin, nous essayons d'améliorer les performances obtenues par nos meilleures heuristiques en pondérant les messages lors de la *BP* et en appliquant nos critères d'arrêt dynamiques. Cela nous permettra de répondre à nos deux autres questions de recherche :

Q2 : Est-il possible de décider dynamiquement quand stopper les itérations de *BP* afin de gagner du temps de calcul ?

Q3 : Pondérer les messages des contraintes peut-il permettre d'obtenir des distributions plus proches des distributions réelles ?

Les tests ont été exécutés sur le solveur *MiniCPBP* [1], un solveur basé sur *MiniCP* et possédant les algorithmes nécessaires à la *BP*.

4.1 Explications préliminaires

4.1.1 Banc d'essai

Afin d'évaluer la robustesse de nos contributions, tous nos tests sont faits sur un ensemble de 1474 exemplaires venant de XCSP [2]. Afin de pouvoir observer des cas de figure différents, ces exemplaires appartiennent à onze problèmes différents : *Kakuro, LatinSquare, MagicSequence, MagicSquare, MarketSplit, MultiKnapsack, Nonogram, Ortholatin, Primes, PseudoBoolean et Sudoku*. En ce qui concerne *Ortholatin*, notre solveur n'est capable que de résoudre 3 à 4 instances et cela rend l'analyse des performances compliquées. C'est pourquoi les performances obtenues sur ce problème ne sont présentées que dans la section 4.3, où nous comparons les performances de différentes heuristiques, afin d'avoir une vision plus complète des performances de notre approche. Pour chaque exemplaire, le temps limite pour la résolution a été

1. Disponible ici : https ://github.com/PesantGilles/MiniCPBP
2. Disponibles ici : http ://www.xcsp.org/instances/

fixé à 20 minutes, et jusqu'à 12Go d'espace mémoire étaient disponibles.

4.1.2 Damping

Dans le cas des réseaux de contraintes contenant des cycles, comme dans notre cas, il est possible que les marginaux ne convergent pas et oscillent. Il est possible de réduire cette oscillation en appliquant du *damping* aux messages [26]. Le principe est de calculer une moyenne pondérée entre les anciens et nouveaux messages afin d'aplanir la courbe de leur évolution. Nous n'appliquons le *damping* qu'aux messages issus des variables. Les messages envoyés deviennent donc :

$$\mu^{(t)}_{x\to c}(v) = \lambda\mu_{x\to c}(v) + (1-\lambda)\mu^{(t-1)}_{x\to c}(v) \tag{4.1}$$

où $\mu^{(t)}_{x\to c}(v)$ est le message envoyé de la variable x à la contrainte c à l'itération t de *BP* et λ est une constante permettant de pondérer l'importance du message précédent dans le nouveau message. Lorsque $\lambda = 1$, cela correspond à une absence de *damping*. À l'inverse lorsque $\lambda = 0$, tous les messages envoyés de x à c sont identiques, peu importe l'itération t.

Appliquer du *damping* avec $\lambda = 0.5$ a montré de bons résultats avec *max-marginal* [2]. Ainsi toutes les expériences suivantes se feront en appliquant du damping avec $\lambda = 0.5$.

4.1.3 Présentation des résultats

Les résultats sont présentés sous la forme de profils de performance. L'abscisse correspond à une mesure, par exemple le temps de calcul, et l'ordonnée à une proportion d'exemplaires résolus. Si l'on reste dans l'exemple où l'abscisse correspond à un temps de calcul, un point d'une courbe de coordonnées (x, y) signifie que la configuration correspondant à la courbe est capable de résoudre y% d'exemplaires en un temps inférieur où égal à x. Plus la courbe croit rapidement et tôt, plus les performances sont bonnes.

Nous mesurons trois types de performances différentes. Premièrement, les performances en termes de nombre d'échecs au cours de la recherche. Cela nous permet d'évaluer la qualité des décisions de branchement. Nous observons la pertinence de notre heuristique. Ensuite, les performances en termes de temps de calcul. Ces mesures sont souvent liées au nombre d'échecs, mais cela permet aussi de prendre en compte les coûts de calculs impliqués par l'heuristique. Enfin, le troisième type est lié aux critères d'arrêt dynamiques pour la *BP*. Lors de la résolution d'un exemplaire, le nombre d'itérations avant chaque branchement est mesuré. Cela permet de calculer, pour chaque exemplaire, le nombre moyen d'itérations avant chaque

branchement. Et c'est cette mesure qui est utilisée pour le troisième type de performance. Cela nous permet d'observer l'impact des critères d'arrêt sur le nombre d'itérations, de savoir s'ils permettent une économie sur le nombre d'itérations.

4.2 Optimisation d'impact-entropy

4.2.1 Choix de la méthode de calcul pour l'impact d'une variable

Dans le cas de *Impact-entropy*, nous disposons de deux manières de calculer l'impact d'une variable. Soit en calculant la moyenne des impacts des valeurs, soit en calculant l'espérance de l'impact. Le but de ce test est de déterminer quelle méthode donne les meilleures performances. Pour les deux configurations, nous utilisons une *DFS* avec des redémarrages. Le premier redémarrage s'effectue après 100 échecs et ce nombre est multiplié par 1,5 à chaque redémarrage. Les tests ont été exécutés sur un serveur avec deux Intel E5-2683 v4 Broadwell @ 2.1Ghz.

Si l'on observe le nombre d'échecs (figure 4.1), pour la plupart des problèmes les deux courbes sont confondues. Les performances sont donc similaires pour ces problèmes. Mais dans le cas de *Primes*, *MultiKnapsack* et *MagicSquare*, la configuration avec l'impact moyen (en orange) montre de meilleures performances que la configuration avec l'espérance de l'impact. Dans le cas de *Nonogram* c'est l'inverse, mais cette amélioration n'est pas suffisante pour nous permettre de conclure que l'espérance de l'impact est une meilleure option que l'impact moyen. Ainsi, calculer la moyenne des impacts semble être une méthode plus robuste que l'espérance de l'impact. Pour la suite des expériences, nous avons choisi d'utiliser la moyenne comme méthode de calcul de l'impact d'une variable.

4.2.2 Optimisation des paramètres de redémarrage

Impact-entropy apprenant les impacts des affectations au cours de la recherche, recommencer la recherche en cours de route peut être intéressant. En effet, cela permet notamment d'exploiter les estimations des impacts plus précises pour prendre de nouvelles décisions à la racine de l'arbre, ces décisions étant déterminantes pour la suite de la recherche. Après un certain nombre f d'échecs, celle-ci est redémarrée. Mais ce nombre ne peut pas rester le même à chaque fois. Au fur et à mesure des redémarrages, l'heuristique dispose d'une connaissance de plus en plus précise des impacts et redémarrer devient alors moins pertinent. Et redémarrer trop tôt empêcherait l'algorithme d'explorer certaines parties de l'arbre. Il faut plutôt essayer d'exploiter plus les impacts, d'aller plus loin dans la recherche. C'est pourquoi la valeur f est augmentée après chaque redémarrage en la multipliant par un facteur ϕ. Ce

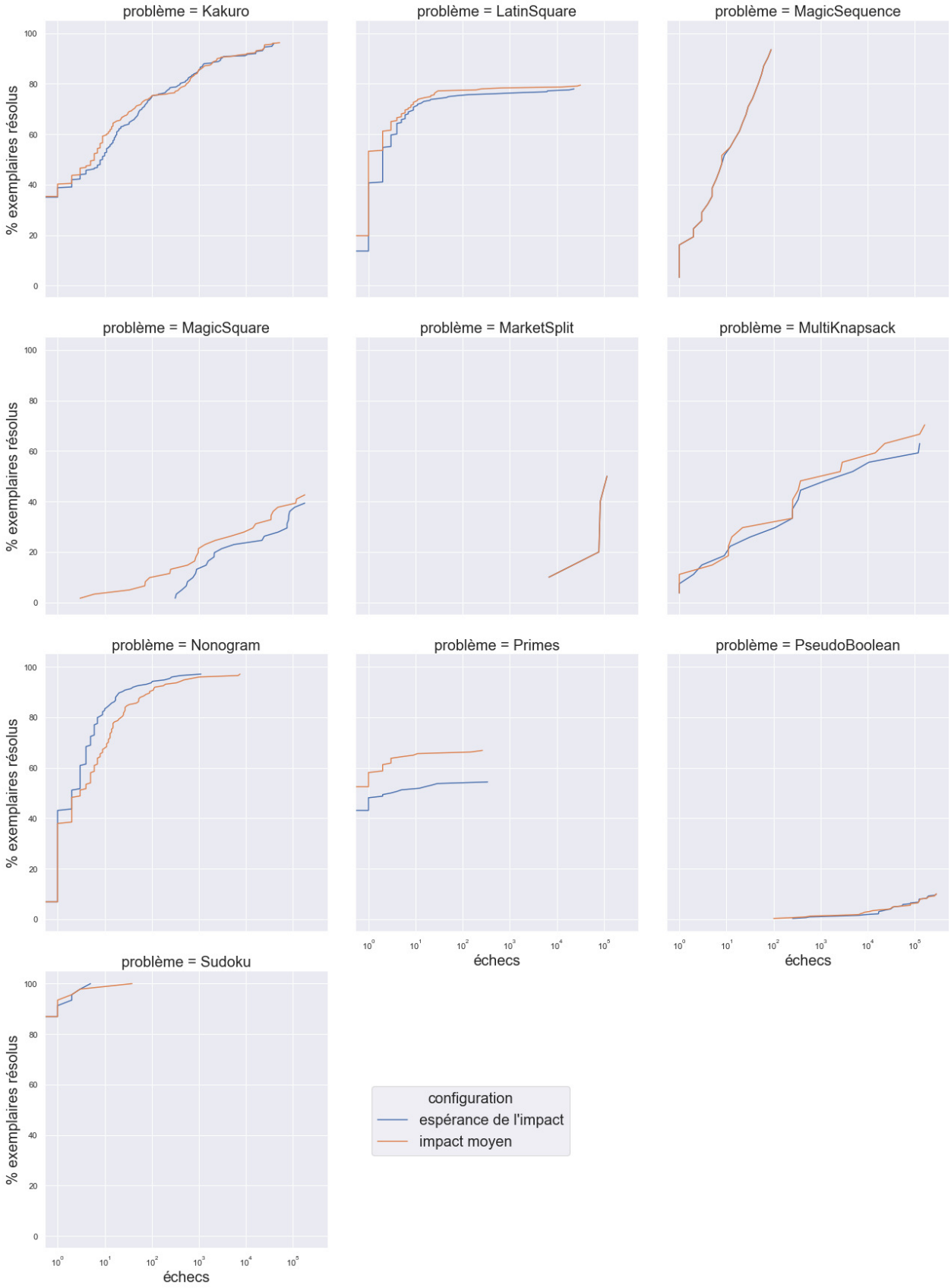

FIGURE 4.1 Pourcentages d'exemplaires résolus en fonction du nombre d'échecs pour *impact-entropy* avec les deux méthodes de calcul d'impact de variable

facteur est fixe tout au long de la recherche.

Les performances étant impactées par ϕ et f, le but de ce test est d'ajuster leur valeur afin d'obtenir des résultats optimaux. La seule heuristique utilisée est *impact-entropy*. Nous testons toutes les configurations possibles avec :

— $f \in \{50, 100, 150, 200\}$
— $\phi \in \{1.25, 1.5, 2\}$

Afin de vérifier l'importance des redémarrages, nous ajoutons une configuration d'*impact-entropy* sans redémarrage. Les impacts sont initialisés avant la recherche. Nous utilisons une *DFS*. Les tests ont été exécutés sur un serveur avec deux Intel E5-2683 v4 Broadwell @ 2.1Ghz.

La figure 4.2 montre les profils des performances en termes de nombre d'échecs des différentes configurations. Dans un premier temps, concentrons-nous sur les configurations avec redémarrages (en couleur et lignes pointillées). La première chose que nous pouvons remarquer est que beaucoup de problèmes ne sont pas impactés par la valeur de f et ϕ. En effet, pour *Primes, Sudoku, MagicSequence, Kakuro, LatinSquare, Nonogram* et *PseudoBoolean* toutes les courbes avec redémarrages sont très rapprochées, voir superposées. Avec les performances des trois problèmes restants, il est compliqué d'en tirer une configuration qui soit optimale pour tous les problèmes. Pour *MultiKnapsack* ce sont les configurations avec $f = 50$ (en jaune) qui donnent les meilleures performances. Pour *MagicSquare* c'est la configuration avec $f = 50$ et $\phi = 1.5$ (en pointillés jaunes) qui semble être au-dessus des autres. Enfin du côté de *MarketSplit*, la configuration avec $f = 100$ et $\phi = 2$ (en rouge et tirets/pointillés) a la courbe qui croît la plus vite, mais la configuration résolvant le plus d'exemplaires est $f = 50$ et $\phi = 2$ (en jaune et tirets/pointillés). Ainsi, il est difficile de choisir une configuration comme étant la meilleure étant donné soit la similarité des performances pour un même problème, soit les différences entre problèmes pour la configuration fonctionnant le mieux. Pour la suite nous utiliserons $f = 200$ et $\phi = 1.5$ lorsqu'il faudra utiliser des redémarrages avec *impact-entropy*. Même si cette configuration n'est pas la meilleure pour les exemplaires de *MultiKnapsack*, elle montre une bonne performance sur tous les autres problèmes.

Enfin, comparons les configurations utilisant des redémarrages avec la configuration référence sans redémarrage (en noir). En observant la figure 4.2, nous voyons que l'heuristique bénéficie moins des relances que l'on pourrait s'y attendre. Effectivement pour *LatinSquare* l'usage de redémarrages permet une amélioration des performances, mais elle reste très légère. La courbe sans redémarrage (en noir) n'est que légèrement en dessous des autres. En ce qui concerne *MultiKnapsack*, on observe une amélioration significative des performances sur certains exemplaires mais c'est la configuration sans relance qui résout le plus d'exemplaires. Pour *Primes*,

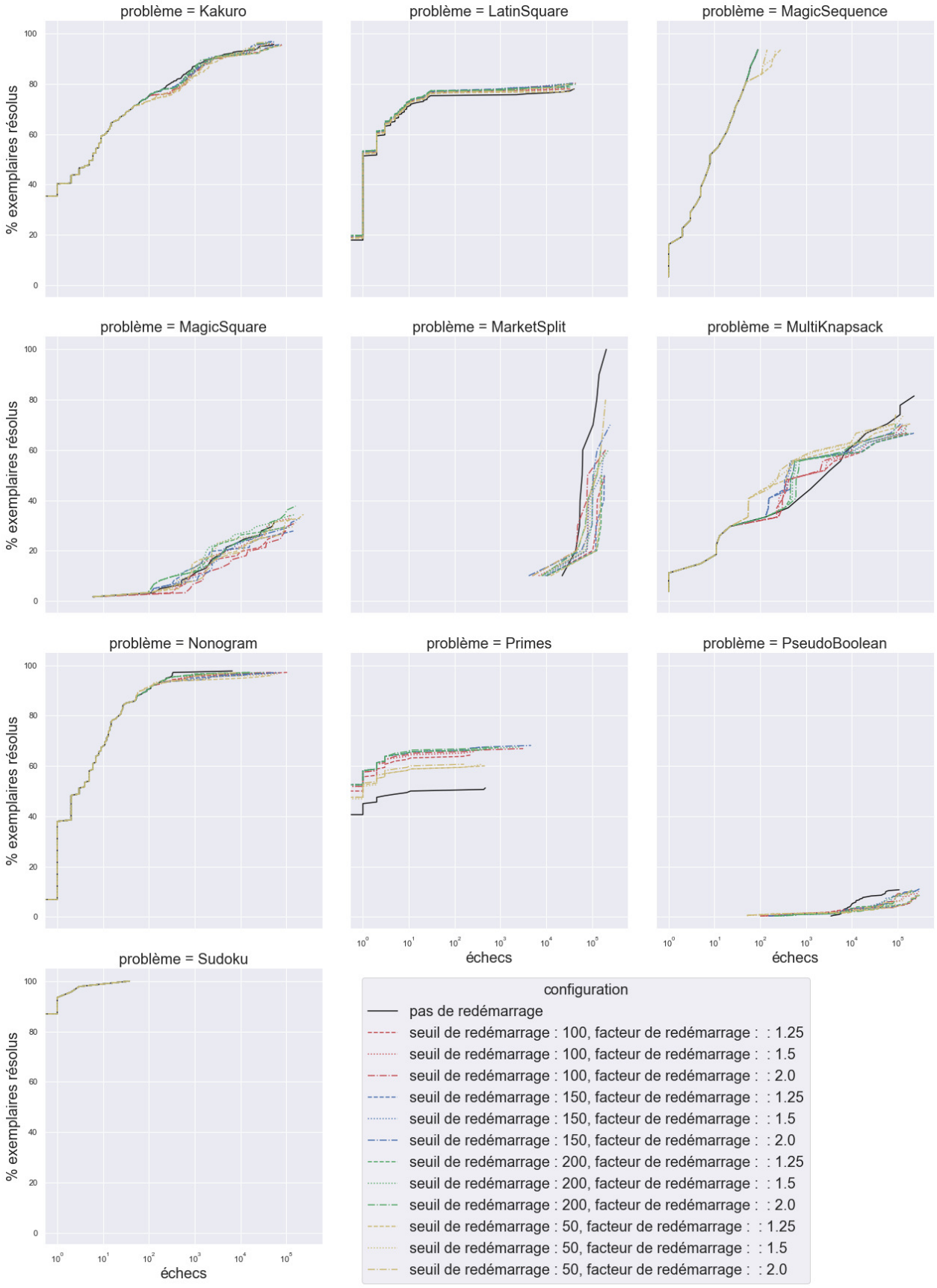

FIGURE 4.2 Pourcentages d'exemplaires résolus en fonction du nombre d'échecs pour *impact-entropy* selon les paramètres de relance

il faut tout de même noter que l'amélioration apportée par les redémarrages est notable. Mais pour *MarketSplit*, les redémarrages apportent une pénalité conséquente. Ce problème contient un nombre de variables relativement faible, chacune ayant un domaine binaire, et des contraintes *sum* très restrictives. Ainsi, les arbres de recherche ne sont pas très profonds. Utiliser des redémarrages pour s'extraire de sous-arbres sans solution est donc moins pertinent avec ce problème. Et le fait de recommencer la recherche fait que des sous-arbres sans solutions sont sûrement explorés plusieurs fois, d'où la détérioration des performances. Pour les autres problèmes, il n'y a aucun impact.

4.2.3 Optimisation d'impact-min-entropy

Avec *impact-min-entropy* lorsque l'entropie du modèle est supérieure à un certain seuil ρ, c'est *impact-entropy* qui est utilisée. En donnant plus ou moins d'importance à *min-entropy* au cours de la recherche, ce seuil a un impact sur les performances. Ce test a pour but de tester différentes valeurs de ρ afin d'observer son impact sur les performances et de choisir la meilleure valeur. Les performances d'*impact-min-entropy* ont été évaluées avec $\rho \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, avec et sans redémarrage au cours de la recherche. Dans les configurations avec redémarrage, le premier redémarrage s'effectue après 200 échecs et ce nombre est multiplié par 1.5 à chaque redémarrage. Les tests ont été exécutés sur un serveur avec deux Intel E5-2683 v4 Broadwell @ 2.1Ghz.

La figure 4.3 présente les performances en termes de nombre d'échecs. Sur ces courbes, la couleur est liée à la valeur de ρ et le style de ligne indique si la configuration en question utilise des redémarrages. Tout d'abord, nous pouvons noter qu'en ce qui concerne *Sudoku, Magi-Sequence* et *Kakuro* les performances sont similaires pour toutes les configurations. En effet, toutes les courbes sont très rapprochées pour ces problèmes-là. Pour les autres problèmes les conclusions varient : Pour *MultiKnapsack* sur la plupart des exemplaires les différentes configurations donnent des résultats semblables, mais pour les exemplaires les plus compliqués la configuration avec redémarrages et avec un $\rho = 0.9$ (en ligne pleine et jaune) se démarque. Pour *MarketSplit* les configurations sans redémarrages (en lignes pleines) sont celles qui fonctionnent le mieux. La valeur de ρ n'a pas d'impact significatif. En ce qui concerne *MagicSquare* selon la valeur de ρ les performances observées sont très différentes. Privilégier *min-entropy* est la meilleure option : plus ρ est grand, plus la courbe domine les autres. C'est aussi la meilleure stratégie pour les problèmes *Primes* et *PseudoBoolean*, où les courbes correspondant à $\rho = 0.9$ (en jaunes) dominent légèrement les autres. Mais pour ces problèmes, les différences de performances sont bien moins prononcées : les courbes sont plus resserrées. De plus les configurations avec et sans redémarrage montrent des résultats identiques.

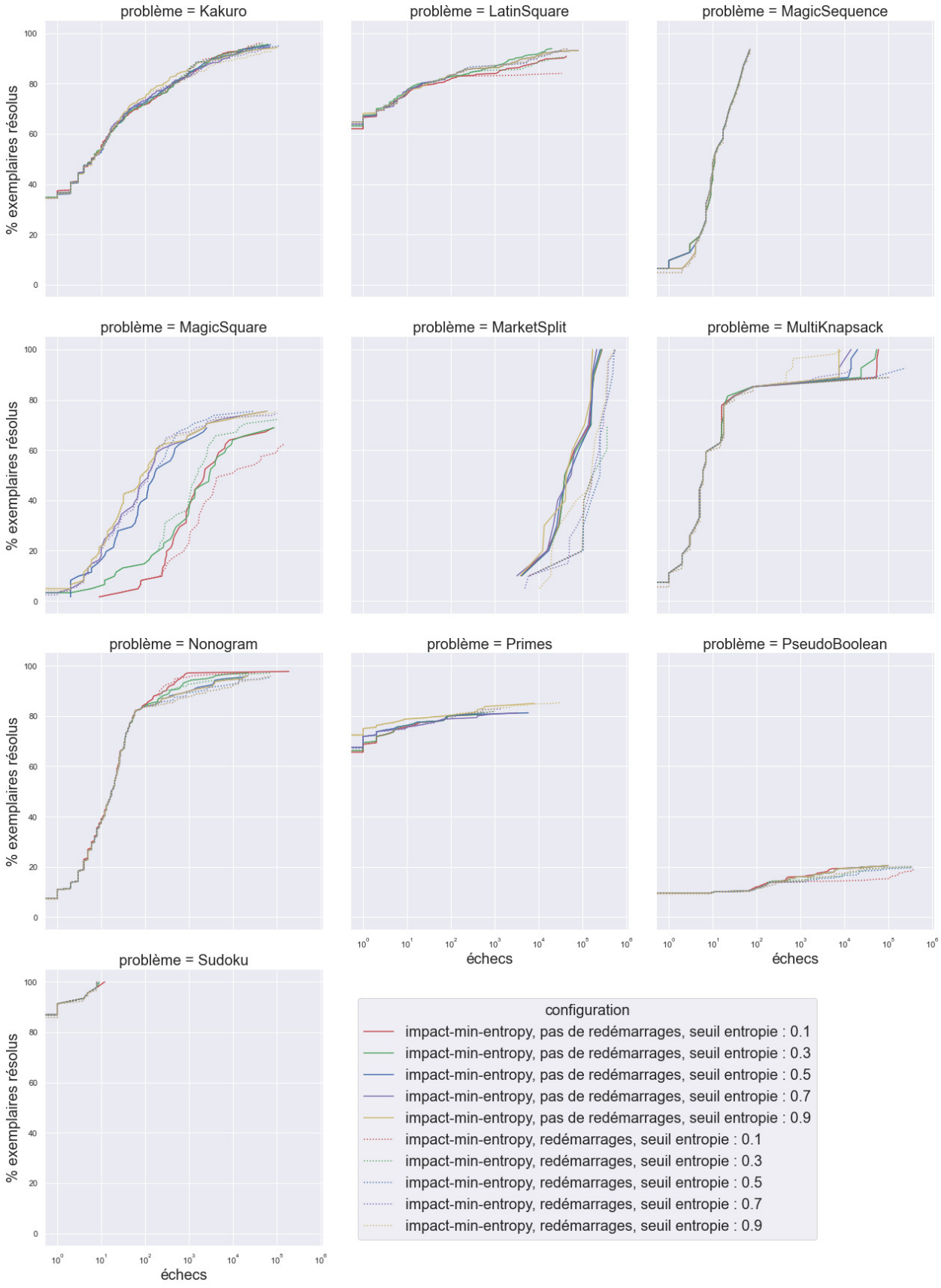

FIGURE 4.3 Pourcentages d'exemplaires résolus en fonction du nombre d'échecs pour *impact-min-entropy* avec et sans redémarrage et pour différents seuils d'entropie

L'utilisation de redémarrages semble n'avoir d'impact significatif sur les performances que dans le cas de *MarketSplit*, *MagicSquare* et *LatinSquare*. Pour les autres problèmes, les courbes pleines et pointillées d'une même couleur sont très similaires. Mais il faut noter que pour ces problèmes, plus l'heuristique donne de l'importance à *impact-entropy*, plus l'heuristique est pénalisée par les redémarrages. C'est particulièrement visible pour *LatinSquare* et *MagicSquare* où les courbes correspondant à $\rho = 0.1$ (en rouge) sont plus écartées que les courbes où $\rho = 0.9$ (en jaune).

4.3 Comparaison des heuristiques de branchement

Le but de cette expérience est d'évaluer les performances de nos heuristiques *min-entropy*, *impact-entropy* et *impact-min-entropy*. Nous les comparons avec les heuristiques *max-marginal* et *max-marginal-strength*, afin de voir si nos nouvelles heuristiques sont un meilleur usage de la *BP*. Elles sont aussi comparées avec les performances de l'heuristique *dom/wdeg* afin de les situer dans l'état de l'art et d'observer si l'usage de la *BP* est rentable. Pour chaque heuristique à l'exception d'*impact-entropy* et *impact-min-entropy*, nous testons une configuration avec une *LDS* et une configuration avec une *DFS*. Pour *impact-entropy* et *impact-min-entropy*, il y a une configuration avec redémarrages et une configuration sans. Pour ces deux heuristiques, le premier redémarrage s'effectue après 200 échecs et ce nombre est multiplié par 1.5 à chaque redémarrage, ce réglage étant celui qui donne les meilleures performances pour *impact-entropy*. Pour *impact-min-entropy* le seuil ρ a été fixé à 0.9, car c'est pour cette valeur que nous observions les meilleurs résultats pour le plus de problèmes.

Dans les profils de performance, les couleurs utilisées pour chaque courbe correspondent aux heuristiques. Le style de la ligne (ligne pleine ou pointillée) indique soit l'utilisation d'une *LDS* ou d'une *DFS*, soit l'utilisation ou non de redémarrages. Si nous commençons par observer la figure 4.5, présentant les performances en termes de nombre d'échecs, une première observation possible est que *min-entropy* performe globalement mieux avec une *LDS* qu'avec une *DFS*. En effet, les courbes correspondantes (en jaune) sont soient confondues, soit la courbe correspondant à la *LDS* (en pointillée) est au-dessus de la courbe de la *DFS* (en ligne pleine). L'exception étant *Kakuro*, où l'on observe la situation inverse. Cette heuristique montre aussi de très bonnes performances. Pour *Primes, PseudoBoolean, MultiKnapsack* et *LatinSquare* elle est à égalité avec *max-marginal* (en rouge) et/ou *max-marginal-strength* (en vert). Pour *MarketSplit* et *MagicSquare*, c'est l'heuristique montrant les meilleures performances. Enfin, en dehors des problèmes *Kakuro* et *Magic-Sequence* elle est toujours au moins compétitive avec *dom/wdeg* (en noir), et la surpasse même pour plusieurs problèmes. *Impact-entropy* (en bleu) donne des résultats moins intéressants. Nous pouvons observer que dans le cas de *Ma-*

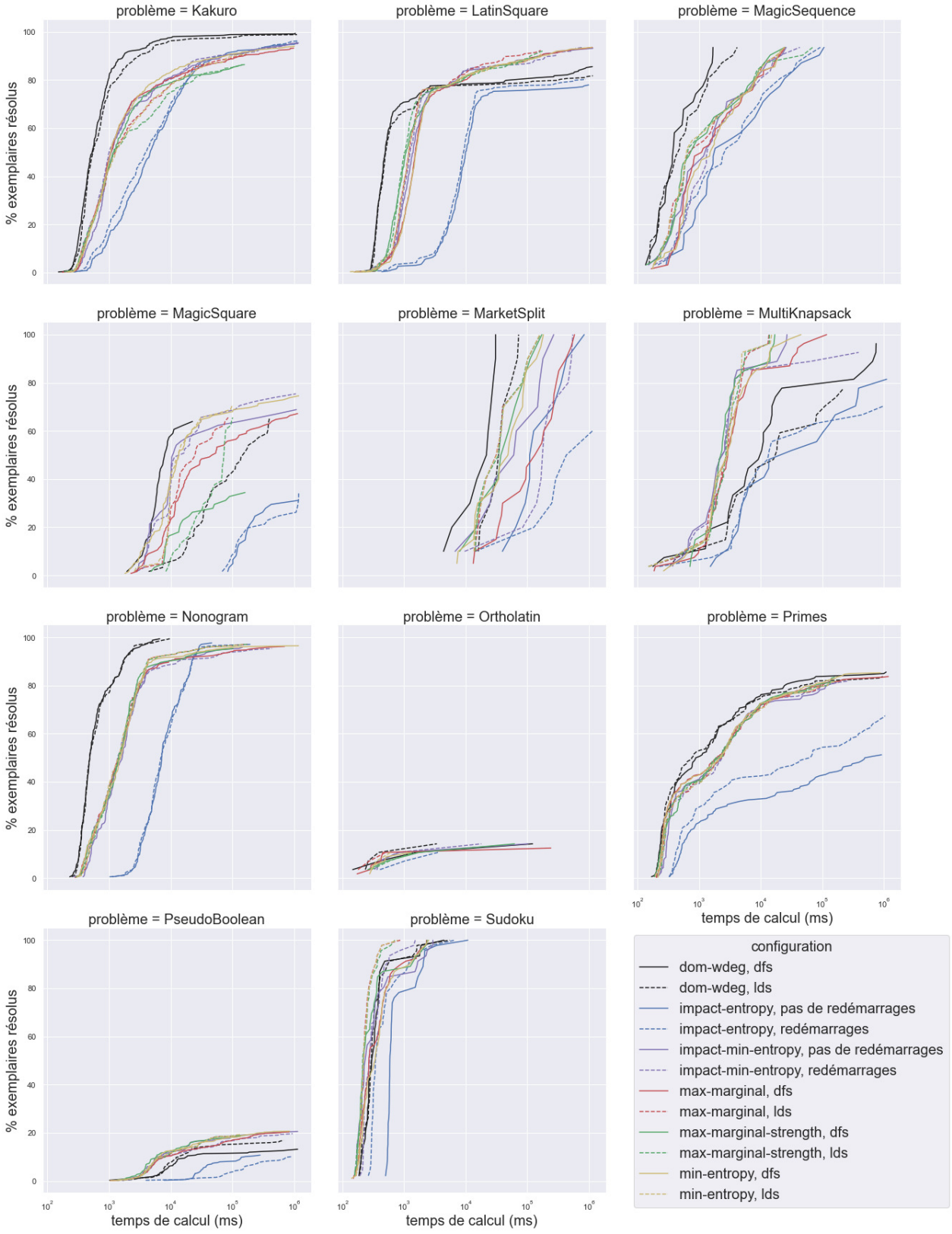

FIGURE 4.4 Pourcentages d'exemplaires résolus en fonction du temps de calcul (ms) selon l'heuristique et le type de recherche utilisés

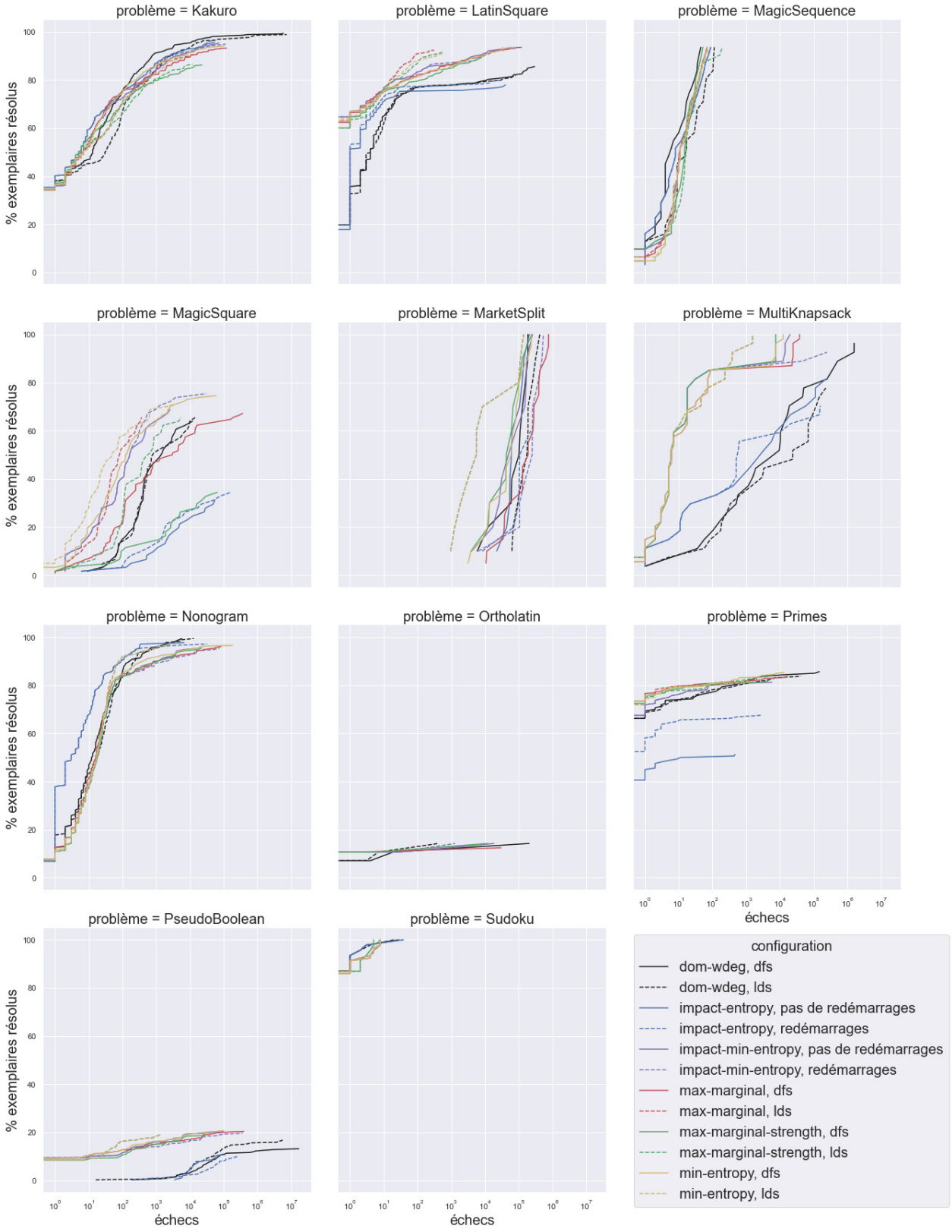

FIGURE 4.5 Pourcentages d'exemplaires résolus en fonction du nombre d'échecs selon l'heuristique et le type de recherche utilisés

gicSquare, *Primes*, *LatinSquare* et *PseudoBoolean* elle est parmi les heuristiques les moins performantes. Voir la moins performante tout court, avec un écart conséquent pour *Primes* et *MagicSquare*. Enfin, *Impact-min-entropy* (en violet), donne des performances très proches de *min-entropy*, pour tout les problèmes à l'exception de *MarketSplit*, mais ne la surpasse jamais. Nous pouvons donc conclure que l'approche n'est finalement pas très pertinente, car elle ne réussit pas à améliorer les performances de *min-entropy*, ce qui était le but.

Intéressons-nous maintenant au temps de calcul en observant la figure 4.4. *Dom/wdeg* ne requiert pas d'itérations de *BP*, contrairement aux autres heuristiques. Ces dernières sont donc pénalisées par un surcoût lié à cette propagation. C'est pour cela que *dom/wdeg* montre les meilleures performances pour *Nonogram*, *Primes*, *MarketSplit* malgré un plus grand nombre d'échecs que certaines heuristiques utilisant la *BP*. Mais ce constat est déjà plus mitigé pour *MagicSquare* et *LatinSquare* où, bien que *dom/wdeg* soit la plus rapide pour les exemplaires qu'elle arrive à résoudre, c'est *min-entropy* et *impact-min-entropy* (à égalité avec *max-marginal* et *max-marginal-strength* pour *LatinSqaure*) qui résolvent le plus d'exemplaires. En ce qui concerne *MultiKnapsack*, toute les heuristiques basées sur la *BP*, à l'exception de *impact-entropy*, dominent fortement *dom/wdeg*. Par contre, *impact-entropy* est pénalisé par l'initialisation des impacts en plus de ces faibles performances en termes de nombre d'échecs, et est donc la moins bonne heuristique.

4.4 Critère d'arrêt

4.4.1 Optimisation des paramètres pour le critère basé sur la stabilité

Le critère d'arrêt basé sur la stabilité dispose de deux paramètres sur lesquels nous pouvons jouer :

- le nombre d'itérations pendant lesquelles le marginal doit être l'un des plus élevés, n
- la proportion du meilleur marginal pour qu'un marginal soit considéré comme élevé, γ

Nous testons toutes les configurations possibles avec :

- $n \in \{2, 3\}$
- $\gamma \in \{0.8, 0.9, 1.0\}$

La logique de ce critère est basée sur le choix que l'heuristique *max-marginal* va faire. Il est donc moins pertinent de l'utiliser avec une autre heuristique. Nous avons donc fait les tests avec l'heuristique *max-marginal*. Le nombre maximal d'itérations a été fixé à 15 et nous utilisons une *LDS*. Afin d'observer si le critère permet une amélioration des performances, idéalement un gain sur le temps de calcul dû à l'économie d'itérations de *BP*, nous compa-

rons avec les performances obtenues avec l'heuristique *max-marginal*, un nombre fixe de 5 itérations et une *LDS*. Les tests ont été exécutés avec sur un serveur avec deux AMD Rome 7532 @ 2.40 GHz 256M cache L3.

En observant la figure 4.7, qui présente les performances en termes du nombre d'échecs, on remarque qu'avec les mauvais paramètres le critère est capable de détériorer la qualité des décisions de branchement. En effet pour *MagicSquare*, *MarketSplit*, *LatinSquare* et *PseudoBoolean*, la plupart des configurations avec nombre d'itérations dynamiques donnent de moins bonnes performances que celle avec un nombre fixe d'itérations (en noir), surtout pour *MagicSquare* et *MarketSplit*. Néanmoins pour chacun de ces problèmes, à l'exception de *MarketSplit*, la configuration $\{n = 3, \gamma = 1.0\}$ est compétitive avec la référence. Pour *MultiKnapack* par contre, le critère dynamique permet de réduire le nombre d'échecs. La courbe de référence (en noir) y est en effet la plus basse. Pour les autres problèmes, les performances sont très similaires, peu importe la configuration.

Regardons maintenant la figure 4.8 qui présente les performances en termes de nombre d'itérations moyen avant branchement. Afin d'observer plus facilement si le critère permet une économie d'itérations de *BP*, une ligne noire verticale a été ajoutée. Son abscisse est de 5. Ainsi, lorsqu'une courbe est à gauche de cette ligne, c'est que des économies d'itérations sont faites. À l'inverse, lorsqu'elle est à droite c'est que l'on fait en moyenne plus d'itérations que la référence. Comme nous pouvions nous y attendre, plus le critère est rigide plus le nombre moyen d'itérations augmente. La configuration la plus rigide $\{n = 3, \gamma = 1.0\}$ (en violet) est systématiquement la courbe la plus à droite, tandis que la configuration la plus souple $\{n = 2, \gamma = 0.8\}$ (en rouge) est la plus à gauche.

En observant maintenant les performances en termes de temps de calcul, nous pouvons observer si le critère d'arrêt permet effectivement de gagner du temps. Avec les configurations ayant le plus faible nombre moyen d'itérations, nous pouvons espérer au maximum un temps de calcul divisé par 2. Cette réduction est difficile à voir avec l'échelle logarithmique utilisée sur nos autres graphes. C'est pourquoi la figure 4.6 utilise une échelle linéaire. Mais cette échelle induit un problème de lisibilité. Le temps maximal pour la résolution est de 1200 secondes. Or la plupart des exemplaires sont résolus dans un temps inférieur à 10 secondes. La proportion de mesures intéressantes est donc répartie sur faible proportion de l'échelle, les rendant difficiles à distinguer sur le graphe. Étant donné que les résolutions avec un temps de calcul supérieur à 10 secondes sont une minorité, nous les retirons des graphes afin d'améliorer la lisibilité pour la majorité des résultats. De plus, les mesures de temps de calcul trop faibles sont des valeurs peu fiables, car fortement soumises au bruit. Nous retirons donc les mesures inférieures à 1 seconde. Cela implique aussi que les mesures pour *Sudoku* sont inutilisables.

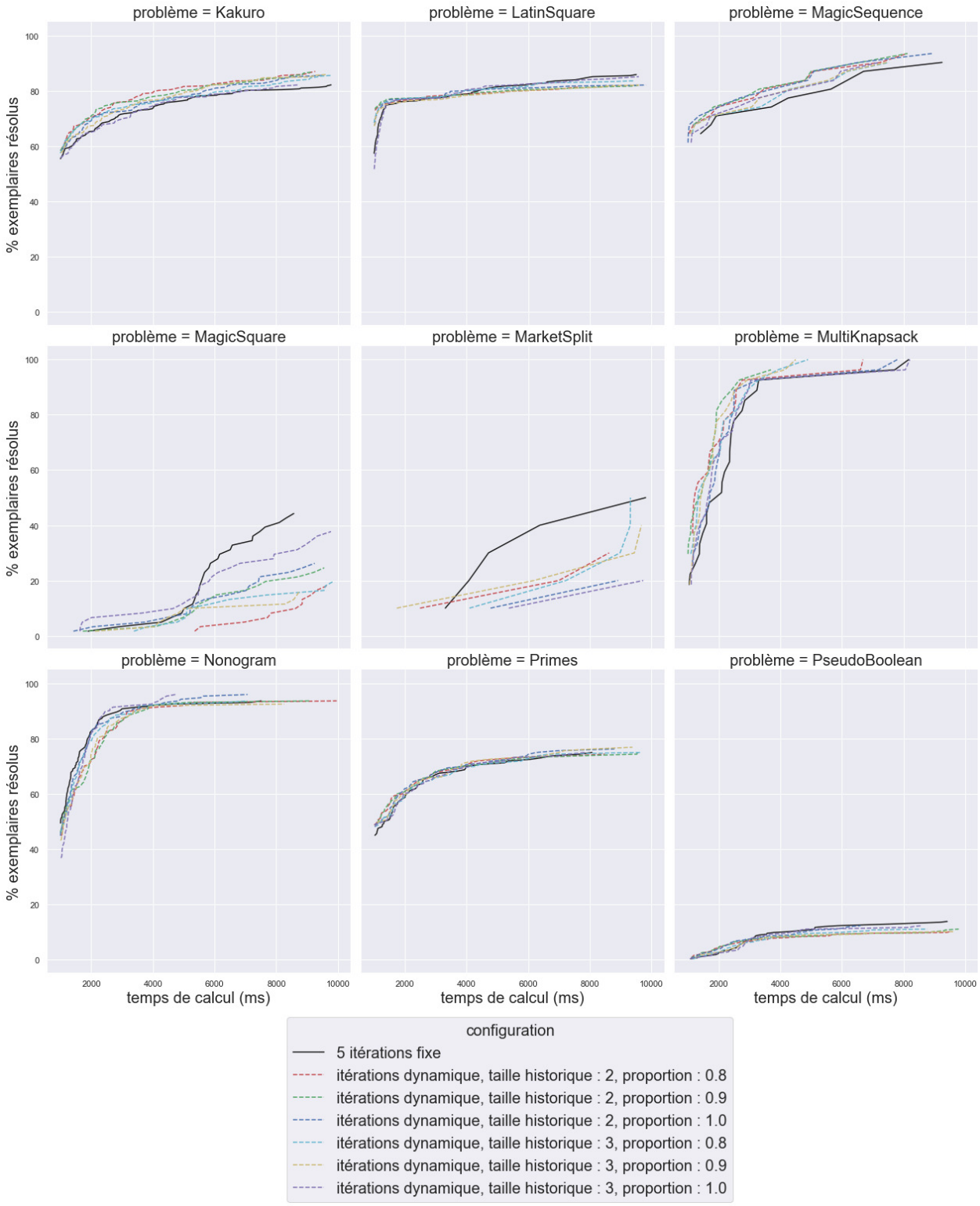

FIGURE 4.6 Pourcentages d'exemplaires résolus en fonction du temps de calcul (ms) pour *max-marginal* avec et sans critère d'arrêt basé sur la stabilité du meilleur marginal

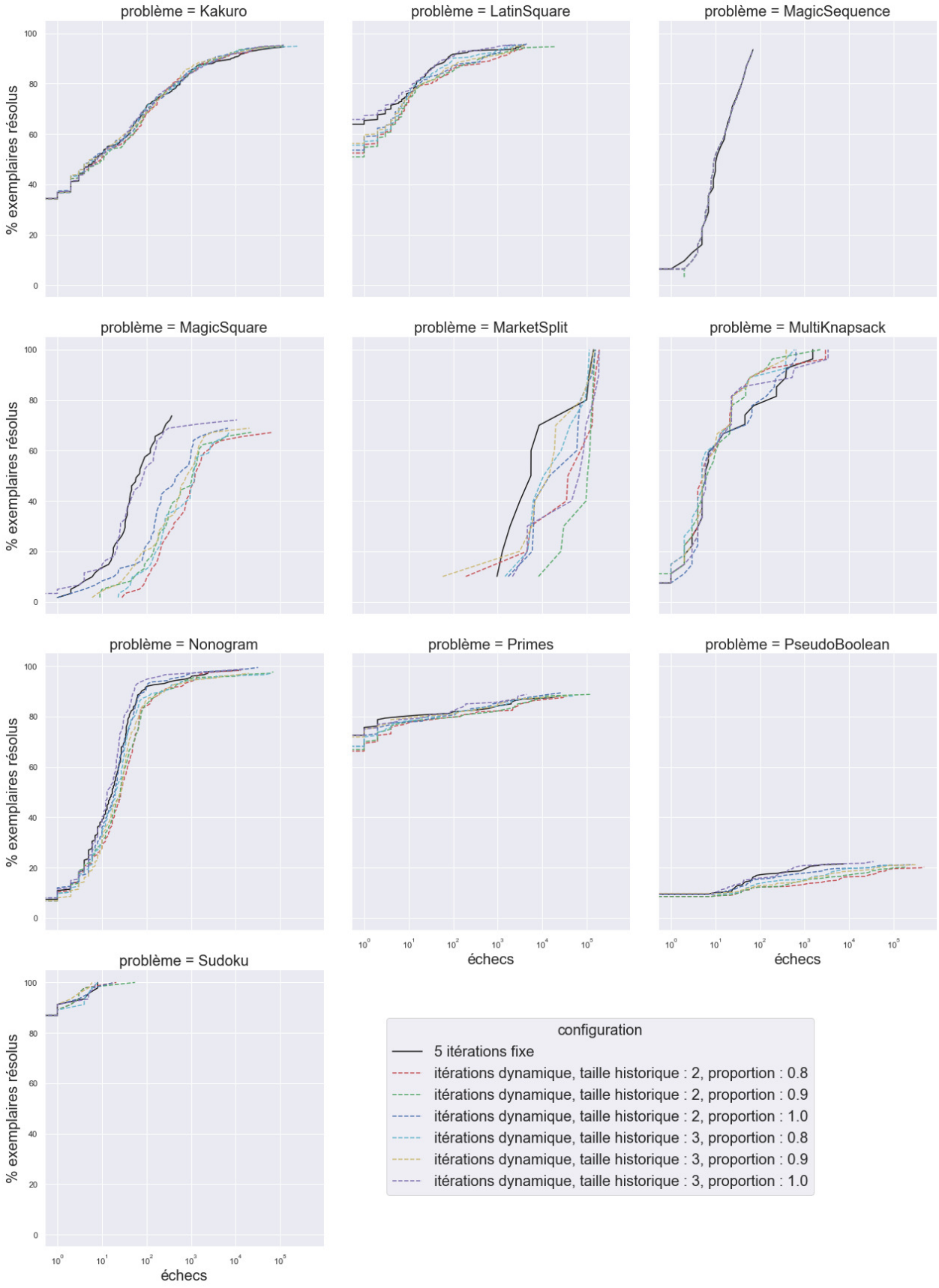

FIGURE 4.7 Pourcentages d'exemplaires résolus en fonction du nombre d'échecs pour *max-marginal* avec et sans critère d'arrêt basé sur la stabilité du meilleur marginal

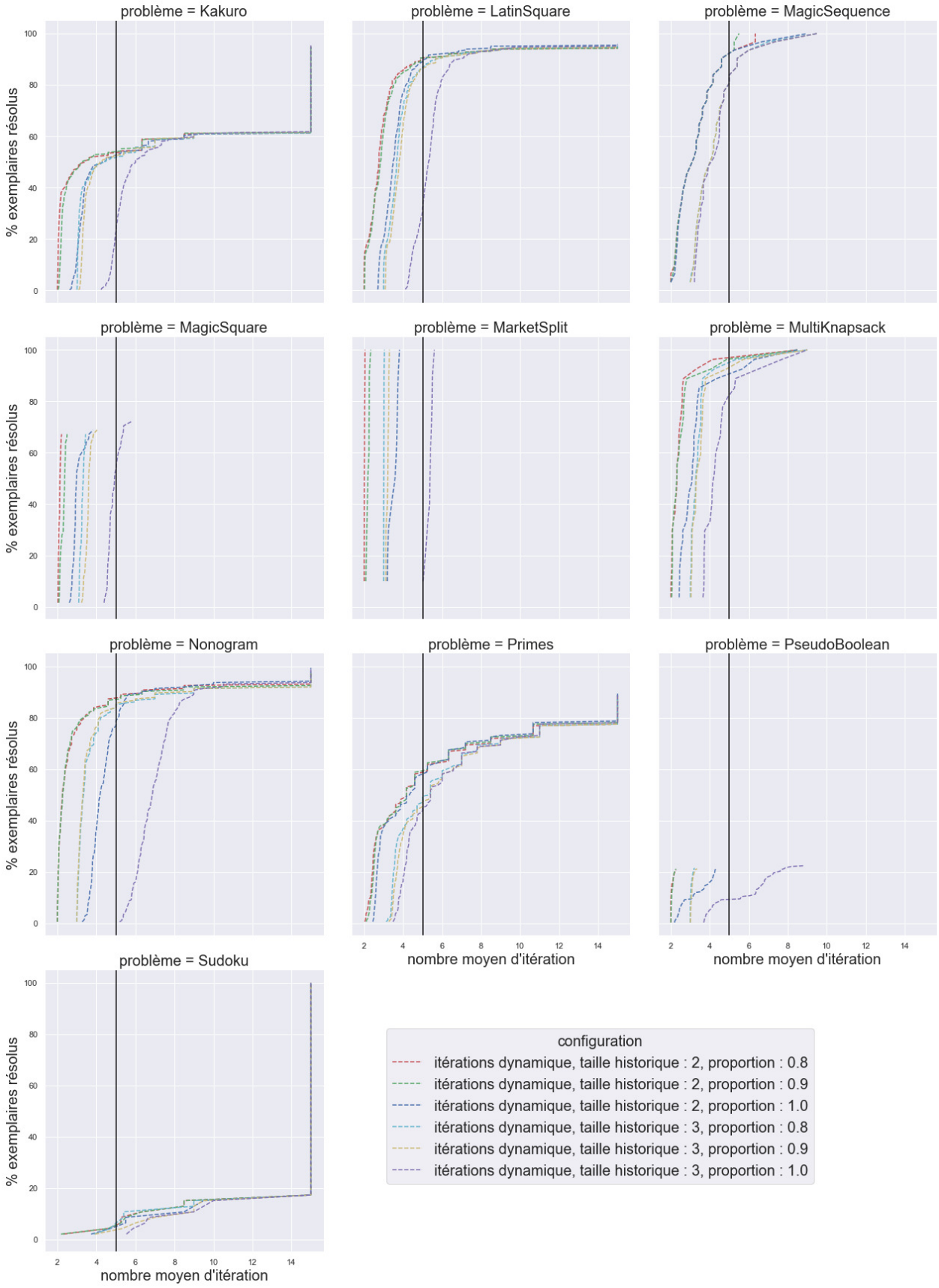

FIGURE 4.8 Pourcentages d'exemplaires résolus en fonction du nombre moyen d'itérations avant branchement pour *max-marginal* avec et sans critère d'arrêt basé sur la stabilité du meilleur marginal

Nous pouvons observer plusieurs cas différents : Pour *MagicSequence* et *Multiknapsack* nous pouvons en effet constater un gain de temps de calcul. La courbe correspondant au nombre fixe d'itérations (en noir) est en dessous des autres. Cette baisse du temps de calcul est en accord avec les observations précédentes pour ces problèmes : un nombre d'échecs similaire ou plus faible pour les configurations avec le critère d'arrêt dynamique et une économie d'itérations de *BP* sur la majorité des exemplaires. De plus, nous pouvons voir que les configurations entraînant la plus forte économie de temps de calcul sont celles qui entraînent la plus forte baisse du nombre moyen d'itérations Mais pour *Nonogram* aussi nous avons observé des nombres d'échecs similaires à la référence et une baisse sur le nombre moyen d'itérations avant branchement. Pourtant on ne peut voir d'économie de temps de calcul. C'est parce que dans ce problème, toutes les variables sont binaires. Ainsi, l'algorithme du point fixe ne réduit pas seulement les domaines des variables, il fixe ces dernières. Cela induit une grande variance sur la taille de l'arbre de recherche. Avec un nombre d'échecs similaires au cours de la recherche, le nombre de nœuds explorés peut être très différent d'une heuristique à une autre. Ceci explique donc l'absence de réduction du temps de calcul pour ce problème. Pour *MarketSplit* et *MagicSquare*, on observe une augmentation du temps de calcul pour les configurations avec critère d'arrêt dynamique. Celle-ci est facilement justifiable avec l'augmentation du nombre d'échecs observée. Pour *PseudoBoolean*, nous observons le même phénomène, mais en plus atténué. Pour *Primes* et *Kakuro*, les figures 4.7 et 4.8 montraient des situations similaires : des nombres d'échecs non impactés par le critère dynamique et une économie d'itérations sur seulement un peu plus de la moitié des exemplaires. Pour le reste des exemplaires on observe une moyenne qui atteint le maximum (15 itérations). Cette situation est en accord avec le profils des temps de calcul de *Primes* ou la courbe de la configuration avec un nombre fixe d'itérations (en noir) est assez similaire à celles des configurations dynamiques. Mais pour Kakuro, nous observons tout de même une amélioration significative des performances. Des observations plus précises ont montré que les exemplaires où le nombre moyen d'itérations avant branchement étaient le plus élevé étaient des exemplaires où le nombre total d'itérations au cours de la recherche était bas. Ainsi, les moyennes élevées ne concernent que les exemplaires les plus simples où l'impact de cette moyenne est plus faible. L'économie de temps de calcul observée est donc cohérente.

4.4.2 Optimisation du seuil de variation pour le critère basé sur l'entropie

Nous devons trouver une valeur pour le seuil de variation de l'entropie α donnant les meilleurs résultats possible. Nous comparons les performances de *min-entropy* avec $\alpha \in \{0.01, 0.05, 0.1, 0.2\}$. Nous exécutons une *LDS* et le nombre maximal d'itérations a été fixé à 15. Les tests ont été exécutés avec sur un serveur avec deux AMD Rome 7532 @ 2.40 GHz 256M cache L3.

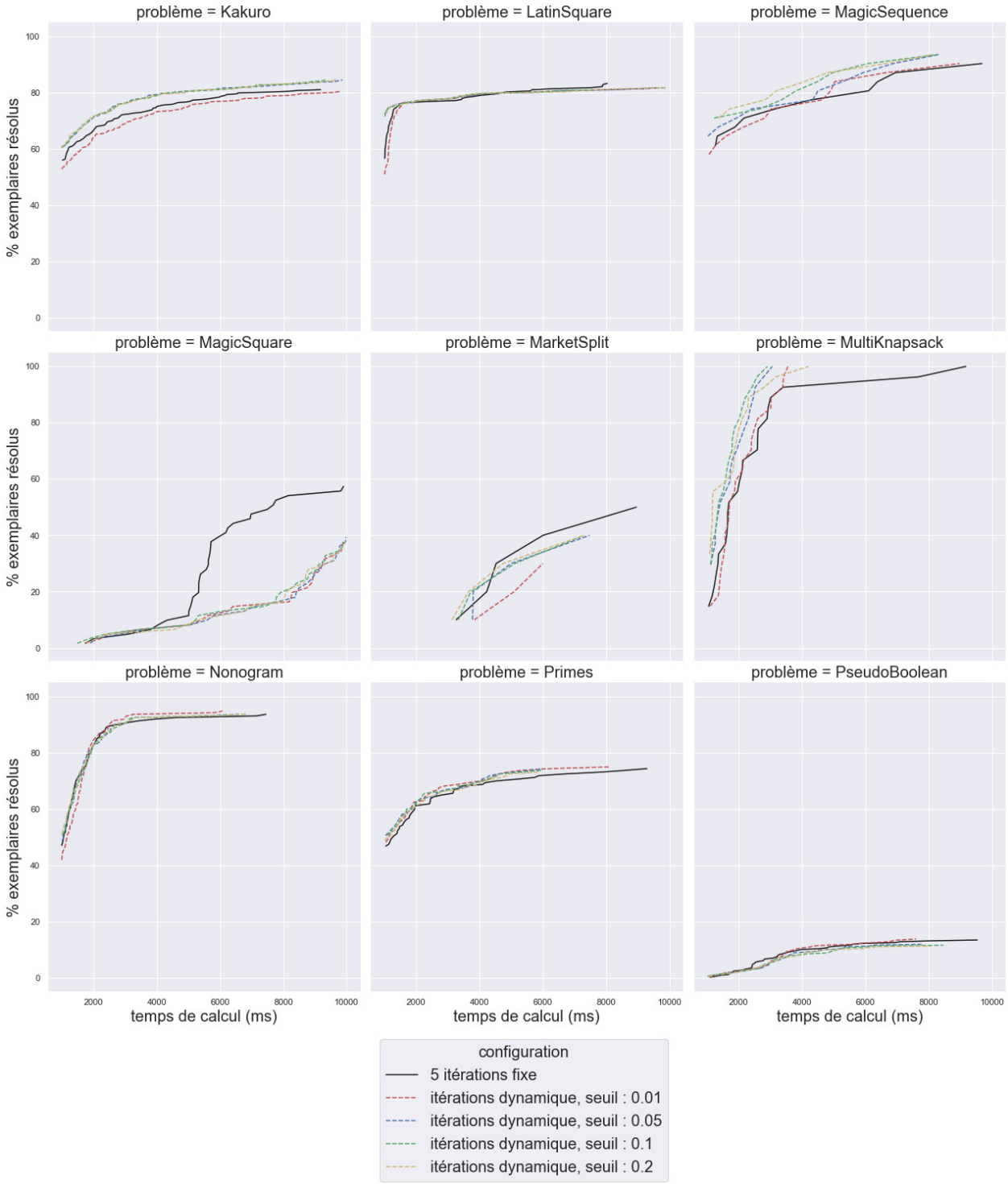

FIGURE 4.9 Pourcentages d'exemplaires résolus en fonction du temps de calcul (ms) pour *min-entropy* avec et sans critère d'arrêt basé sur l'entropie

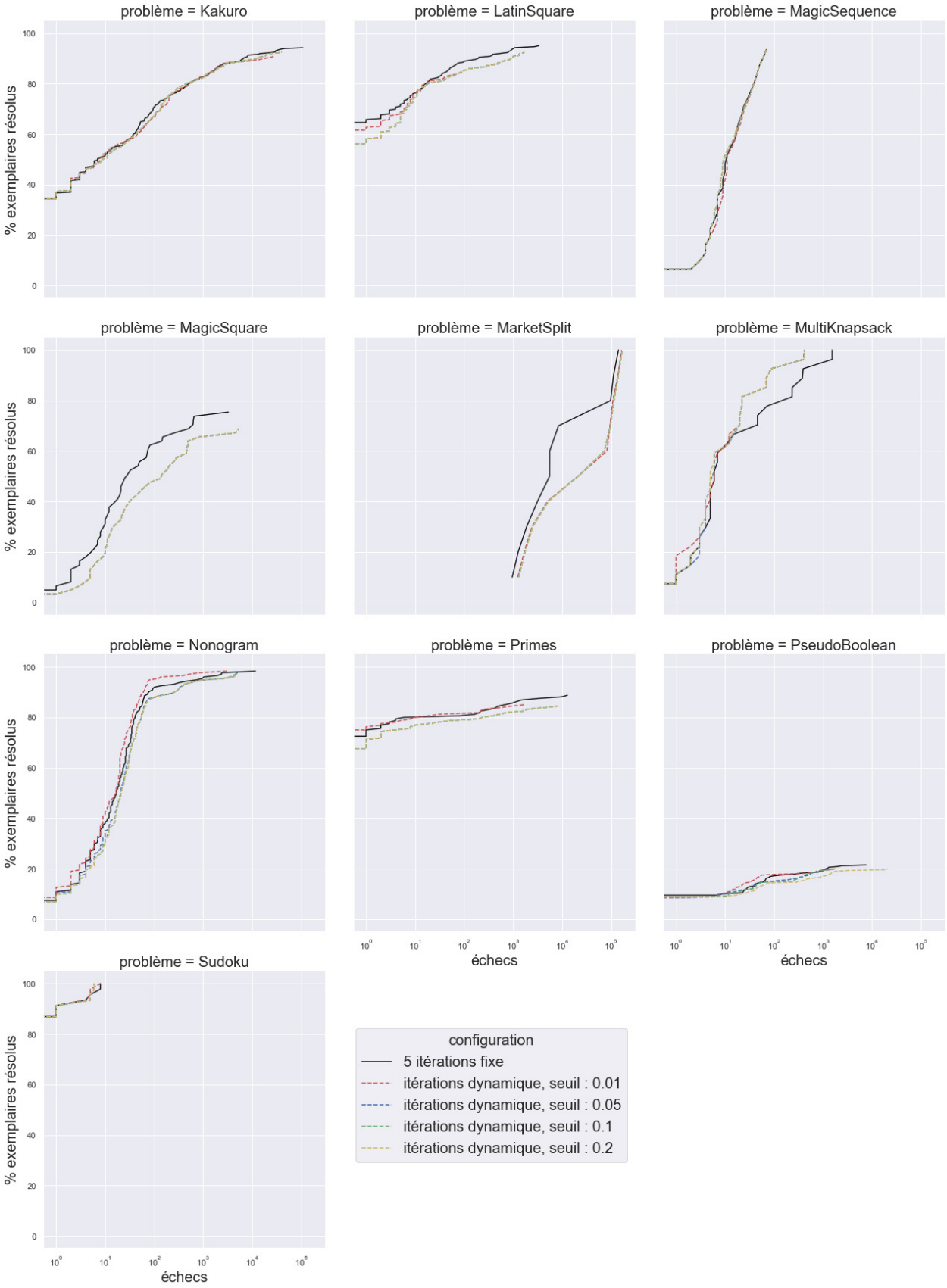

FIGURE 4.10 Pourcentages d'exemplaires résolus en fonction du nombre d'échecs pour *min-entropy* avec et sans critère d'arrêt basé sur l'entropie

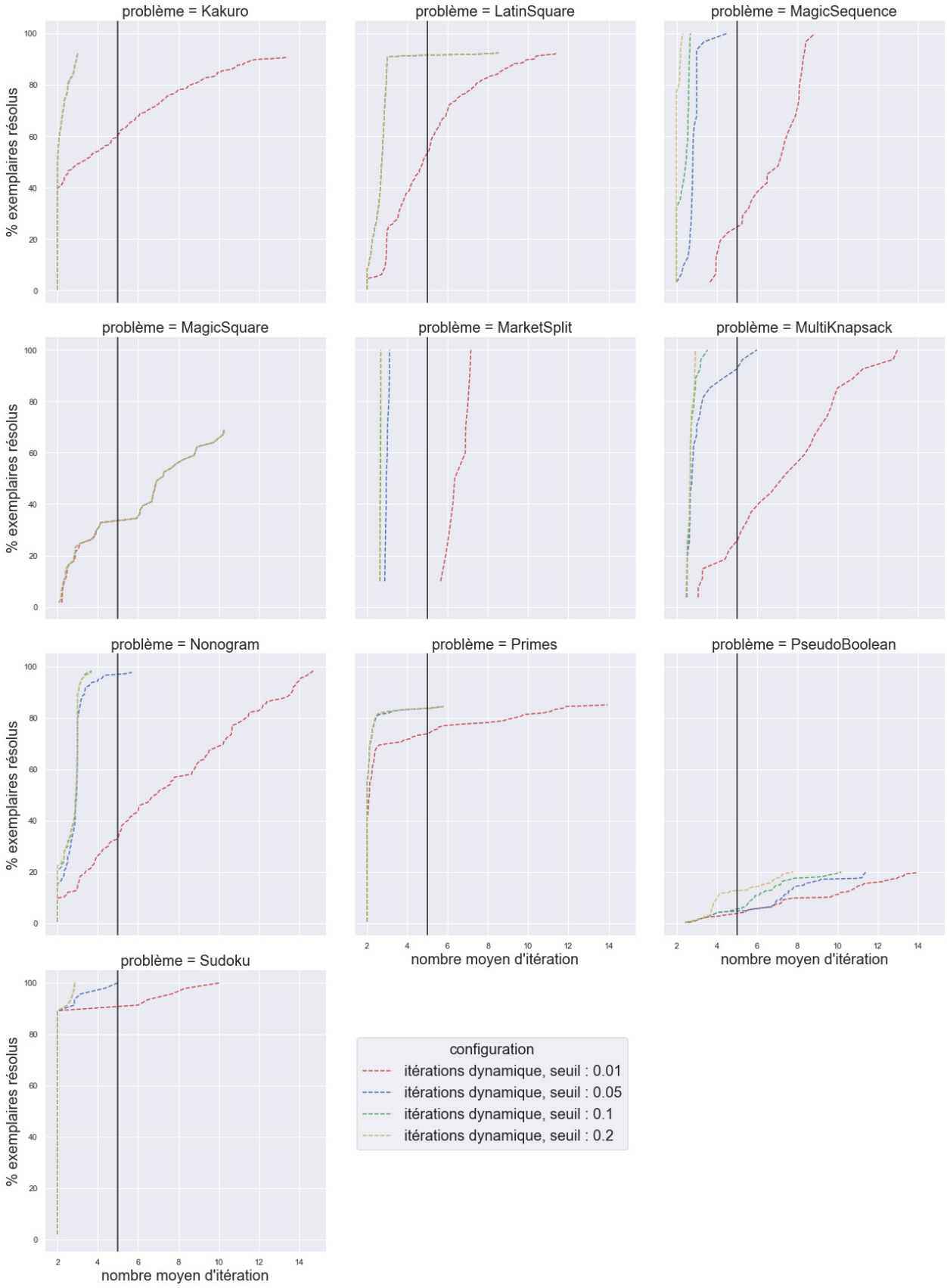

FIGURE 4.11 Pourcentages d'exemplaires résolus en fonction du nombre moyen d'itérations avant branchement pour *min-entropy* avec et sans critère d'arrêt basé sur l'entropie

Les figures 4.9, 4.10 et 4.11 montrent les performances obtenues. Sur ces graphes, la courbe en noir correspond aux performances obtenues avec un nombre fixe de 5 itérations avant chaque branchement. C'est donc la référence, la configuration qu'il faudrait surpasser. Les autres courbes, colorées et en pointillés, correspondent aux performances obtenues avec le critère d'arrêt basé sur l'entropie. La seule différence entre ces configurations est la valeur du seuil α. Commençons par observer la figure 4.10. Ces graphes nous montrent pour chaque configuration et chaque problème, la proportion d'exemplaires résolus avec un nombre d'échecs inférieur ou égal à la valeur en abscisse. Ainsi, ils nous permettent de comparer les qualités des décisions de branchement prises par chaque heuristique. Pour les problèmes *Kakuro, MagicSequence, PseudoBoolean et Sudoku* nous pouvons observer que toutes les courbes sont très rapprochées, voire superposées. Ainsi pour ces problèmes-là, le critère d'arrêt n'impacte que très peu la qualité des branchements. Pour *MagicSquare, MarketSplit, LatinSquare et Primes*, le critère d'arrêt détériore légèrement les performances. En effet nous pouvons y observer que les courbes utilisant le critère se trouvent sous la référence. Surtout pour *MagicSquare* et *MarketSplit*. Il faut aussi noter que pour ces deux derniers, toutes les courbes correspondant aux configurations avec un nombre d'itérations dynamique sont superposées. Peu importe la valeur du seuil, les mêmes performances sont observées. Enfin, si nous nous intéressons à *MultiKnapsack*, nous observons que l'usage du seuil donne de meilleures performances qu'un nombre d'itérations fixe. Là encore, peu importe la valeur du seuil, les performances sont les mêmes.

Intéressons-nous maintenant à la figure 4.11. Elle présente des graphes similaires, mais au lieu d'observer le nombre d'échecs au cours de la recherche, on observe le nombre moyen d'itérations avant branchement. Sur chaque graphe, la ligne noire verticale correspond à la référence, c'est-à-dire à 5 itérations en moyenne. Rapidement, nous pouvons comprendre pourquoi pour *MagicSquare* les profils de performance en termes d'échecs étaient similaires, peu importe la valeur du seuil. En effet, là encore les courbes sont superposées. Mais pour tous les problèmes, à l'exception de *PseudoBoolean* et *MagicSquare*, seule la configuration avec $\alpha = 0.01$ (en rouge) dépasse de manière significative le seuil des 5 itérations moyennes. Ainsi, pour toutes les autres valeurs de seuil, le critère permet d'économiser des itérations de *BP*. Pour *Sudoku*, nous pouvons noter que toutes les courbes commencent avec une croissance verticale jusqu'à environ 90% d'exemplaires. Le minimum d'itération étant de 2 (afin de pouvoir observer une variation de l'entropie), cela signifie que les entropies ne varient que très peu au cours des itérations de *BP*.

Enfin, la figure 4.9 présente les profils de performance, mais en termes de temps de calcul. Nous avons réalisé le même traitement des données que dans la section précédente. Comme dans la section précédente, nous observons une réduction du temps de calcul provoquée par

le critère d'arrêt pour *Kakuro, MagicSequence* et *Multiknapsack*, en accord avec les nombres d'échecs et les nombres moyens d'itérations. Pour *MagicSquare* et *MarketSplit*, l'augmentation du nombre d'échecs provoquée par l'utilisation du critère d'arrêt induit une augmentation du temps de calcul. Pour *LatinSquare* la légère augmentation du nombre d'échecs compense les économies d'itérations de *BP*, nous avons donc des temps de calcul similaires pour le nombre fixe d'itérations et les configurations dynamiques. En ce qui concerne *Primes*, on observe une légère amélioration des temps de calcul, les courbes correspondant à l'utilisation du critère dynamique sont au-dessus de la référence (en noir) peu importe la valeur de α. En ce qui concerne *PseudoBoolean*, toutes les configurations donnent des performances similaires, malgré l'augmentation du nombre moyen d'itérations avant branchement. Comme *Nonogram*, ce problème est composé de variables booléennes, le nombre de nœuds explorés est donc moins lié aux nombres d'échecs que pour les autres problèmes. Cette particularité fait qu'une économie du nombre de nœuds explorés peut compenser l'augmentation du nombre moyen d'itérations.

4.5 Comparaison des schémas de pondération

Le but de cette expérience est d'observer si les schémas de pondération permettent d'améliorer les performances de nos heuristiques. Nous disposons de 5 types de pondérations à comparer :

- Poids identiques (la référence)
- Donner plus de poids aux contraintes avec une grande arité (Arité)
- Donner moins de poids aux contraintes avec une grande arité (Anti-Arité)
- Donner plus de poids aux contraintes à l'origine du plus de conflits (Conflits)
- Donner moins de poids aux contraintes à l'origine du plus de conflits (Anti-Conflits)

Nous comparons les performances de ces pondérations avec nos deux meilleures heuristiques : *max-marginal* et *min-entropy*. Nous appliquons une *LDS*. Le nombre d'itérations de *BP* a été fixé à 5. Le solveur a 20 minutes pour résoudre chaque exemplaire et jusqu'à 12 Go de mémoire étaient disponibles. Les tests ont été exécutés avec sur un serveur avec deux AMD Rome 7532 @ 2.40 GHz 256M cache L3.

La figure 4.12 présente les performances en termes de nombre d'échecs rencontrés pendant la recherche. Ainsi, nous pouvons voir si les schémas de pondération améliorent les décisions prises par les heuristiques, ou si au contraire ils les détériorent. Les résultats sont assez différents selon les problèmes. Pour *Primes, PseudoBoolean* et *Sudoku* toutes les courbes sont très rapprochées, les schémas de pondération n'ont que peu d'impact sur les performances. Pour ces problèmes les contraintes ont des arités très similaires voir identiques pour *Sudoku*, il est donc normal que pondérer en fonction de l'arité n'ait pas beaucoup d'impact. Ainsi,

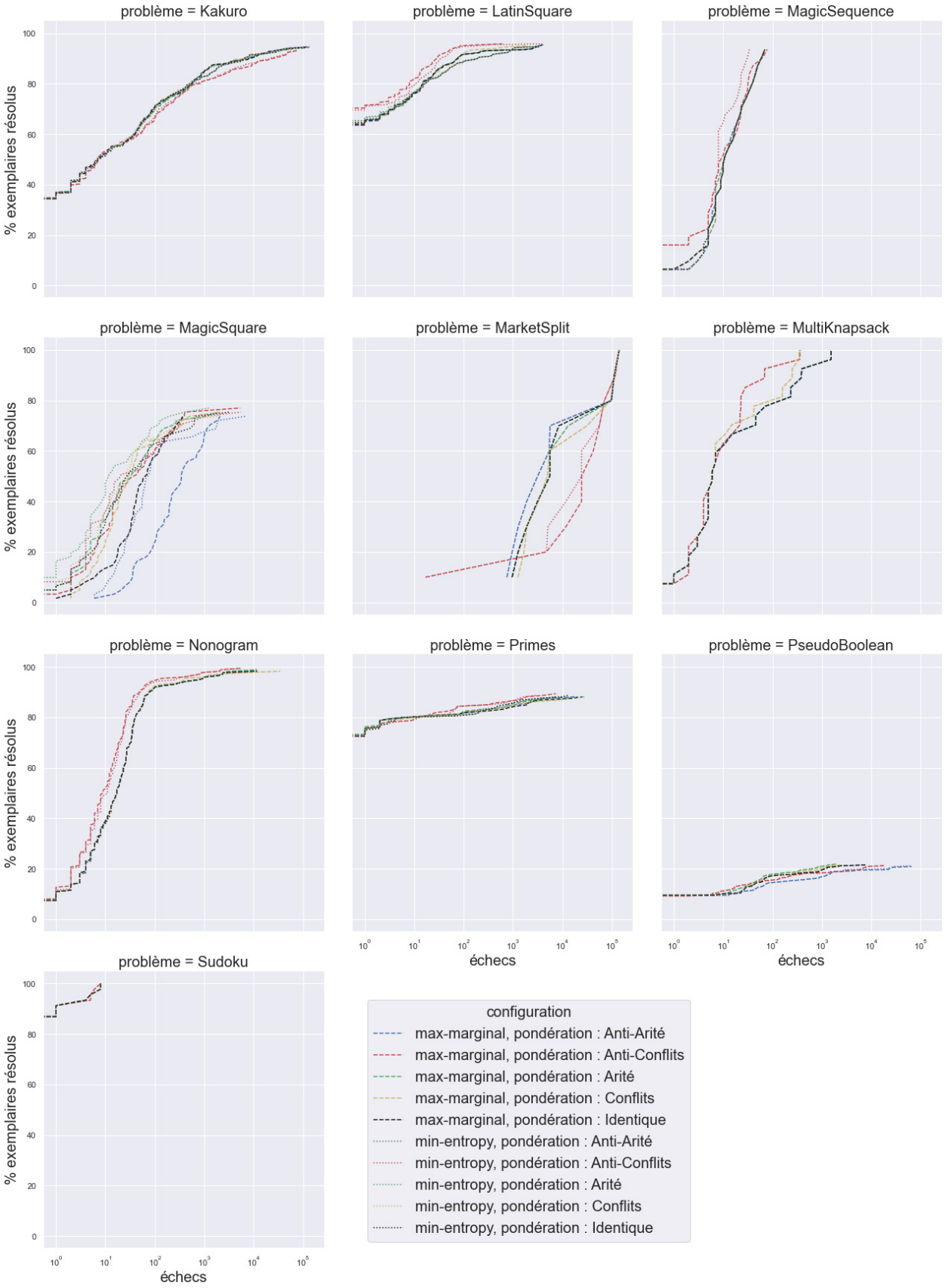

FIGURE 4.12 Pourcentages d'exemplaires résolus en fonction du nombre d'échecs pour *min-entropy* et *max-marginal* selon le schéma de pondération

les autres schémas semblent détériorer légèrement les performances. Nous pouvons faire un constat similaire pour *Kakuro*, mais il faut noter que les schémas *Arité* et *Conflits* (respectivement en vert et en jaune) donnent des performances similaires à la pondération identique. Pour *LatinSquare, Nonogram* et *MagicSequence* les résultats sont plus positifs : les pondérations favorisant l'arité et le nombre de conflits (en vert et en jaune) donnent des performances identiques à l'absence de pondération (en noir), mais les pondérations défavorisant ces attributs (en rouge et en bleu) améliorent légèrement les performances. Pour *MultiKnapsack* les pondérations au-dessus des autres sont celles basées sur le nombre de conflit, surtout celle qui défavorise les contraintes à l'origine de nombreux échecs (en rouge). Les pondérations basées sur l'arité donnent des résultats identiques à l'absence de pondération. Cela s'explique simplement par le fait que toutes les contraintes de ce problème soient appliquées à toutes les variables du problème et aient donc la même arité. Enfin pour les problèmes *MagicSquare* et *MarketSplit* nous pouvons observer de grandes différences de performance. Pour le premier, favoriser l'arité permet d'améliorer considérablement les performances. Cela se comprend très facilement en observant sa structure : des contraintes d'arités similaires sur chaque ligne et chaque colonne de la grille, et une contrainte $AllDifferent$ d'une arité bien plus grande, car appliquée à toutes les variables du problème. Surtout pour min-entropy, la courbe correspondante (en pointillé vert) étant au-dessus de toutes les autres. À l'inverse, défavoriser l'arité (les courbes en bleu) donne de mauvaises performances. Enfin pour *MarketSplit*, la pondération défavorisant l'arité (en bleu) donne pour la moitié des exemplaires des résultats légèrement meilleurs que l'absence de pondération (en noir). Pour l'autre moitié des exemplaires, c'est cette dernière qui donne les meilleurs résultats. Dans ce problème, les contraintes ont des arités très similaires, ce qui explique l'absence d'amélioration apportée par la pondération basée sur l'arité.

Les résultats étant très différents d'un problème à un autre, nous pouvons conclure que si la pondération est bien capable d'améliorer les performances, il est très compliqué de proposer un schéma de pondération comme étant à coup sûr optimal.

CHAPITRE 5 CONCLUSION

5.1 Résumé de la contribution

Dans ce mémoire nous avons voulu répondre aux questions suivantes :

Q1 : Est-ce qu'exploiter l'entropie est une approche intéressante pour exploiter les distributions marginales ?

Q2 : Est-il possible de décider dynamiquement quand stopper les itérations de *BP* afin de gagner du temps de calcul ?

Q3 : Pondérer les messages des contraintes peut-il permettre d'obtenir des distributions plus proches des distributions réelles ?

Nous avons présenté tout d'abord deux critères pour décider dynamiquement quand stopper les itérations de *BP*. Le premier observe les variables et valeurs associées aux marginaux les plus élevés et arrête la *BP* lorsqu'elles n'ont pas changé pendant plusieurs itérations consécutives. Le deuxième observe l'entropie du modèle et stoppe les itérations lorsque celle-ci commence à stagner. Avec les bons réglages, ces critères permettent effectivement de dynamiser le nombre d'itérations de *BP* sans détériorer les performances. Cette réduction a permis d'induire une réduction du temps de calcul pour plusieurs problèmes. Nous pouvons donc répondre à la Q2 en disant qu'il est effectivement possible de réduire le temps de calcul en déterminant dynamiquement lorsqu'il est adéquat de stopper la *BP*.

Nous avons aussi proposé deux manières de pondérer les messages pendant la *BP*. La première pondère les messages des contraintes en fonction de l'arité de ces dernières. La seconde pondère ces mêmes messages en fonction du nombre de conflits provoqués par ces contraintes. Ces pondérations ont démontré une capacité à améliorer significativement les performances sur plusieurs problèmes différents. Mais la pondération à privilégier change d'un problème à l'autre, et peut parfois détériorer les performances. Notre réponse à la Q3 est qu'utiliser de telles pondérations permet d'améliorer les performances en réduisant le nombre d'échecs au cours de la recherche. Cette réduction sous-entend une meilleure estimation des distributions marginales réelles.

Pour finir, nous avons aussi décrit trois nouvelles heuristiques de branchement pour exploiter différemment les distributions marginales. La première, *min-entropy*, choisit la variable avec la plus faible entropie. Elle a montré d'excellents résultats en surpassant *dom/wdeg* sur plusieurs problèmes. La seconde, *impact-entropy*, observe les impacts sur l'entropie du modèle des

affectations tout au long de la recherche. Elle choisit la variable étant censée réduire le plus fortement l'entropie du modèle. Elle a montré des résultats beaucoup plus mitigés, sauf pour le problème *Nonogram* où elle a montré les meilleures performances. La troisième est une combinaison des deux précédentes. *Impact-min-entropy* commence par choisir les variables en fonction de leur impact sur l'entropie du modèle, puis choisit les variables avec la plus faible entropie lorsque celle du modèle passe sous un certain seuil. Elle a montré de bons résultats, mais ne surpasse jamais *min-entropy*. Ainsi, en réponse à la Q1, nous pouvons dire que l'entropie est une approche très prometteuse pour choisir quelles variables doivent être fixées en priorité.

5.2 Limitations

Notre travail possède plusieurs limitations. Tout d'abord, bien que les schémas de pondération et les critères d'arrêt aient prouvé qu'ils étaient capables d'améliorer les performances, l'amélioration qu'ils induisent est variable selon les problèmes. On peut même observer une détérioration des performances dans certains cas. Notre but étant de trouver des approches fonctionnant sur un large panel de problème, cette variabilité est une limitation. Ensuite, notre banc de test ne contient que des problèmes très théoriques. La présence d'exemplaires tirés de problèmes réels donnerait une information plus complète sur la potentielle applicabilité de notre approche.

5.3 Recherches futures

Plusieurs améliorations pourraient être apportées à notre approche. Premièrement, *Impact-entropy* n'a pas montré de bonnes performances sauf pour *Nonogram*, où elle a surpassé les autres heuristiques. Cela indique que l'approche n'est pas dénuée d'intérêt. Deuxièmement, certaines pistes pour l'améliorer pourraient être explorées comme essayer d'autres manières d'estimer les impacts des variables. Ensuite, les schémas de pondération et les critères d'arrêt ont montré un vrai potentiel pour l'amélioration des performances. Il serait donc bon de continuer les recherches dans ce sens afin de trouver un critère ou un schéma donnant de bonnes performances sur une plus large variété de problèmes. Enfin, tester les contributions de ce mémoire sur des applications concrètes permettrait d'avoir une meilleure connaissance du potentiel de notre approche.

RÉFÉRENCES

[1] G. Pesant, "From support propagation to belief propagation in constraint programming," vol. 66, 2019, p. 123–150.

[2] B. Babaki, B. Omrani et G. Pesant, "Combinatorial search in cp-based iterated belief propagation," dans *Principles and Practice of Constraint Programming*, H. Simonis, édit. Cham : Springer International Publishing, 2020, p. 21–36.

[3] F. Rossi, P. van Beek et T. Walsh, édit., *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence. Elsevier, 2006, vol. 2. [En ligne]. Disponible : https://www.sciencedirect.com/science/bookseries/15746526/2

[4] W. D. Harvey et M. L. Ginsberg, "Limited discrepancy search," dans *IJCAI (1)*, 1995, p. 607–615.

[5] C. P. Gomes, B. Selman, H. Kautz *et al.*, "Boosting combinatorial search through randomization," *AAAI/IAAI*, vol. 98, p. 431–437, 1998.

[6] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang et S. Malik, "Chaff : Engineering an efficient sat solver," dans *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA : Association for Computing Machinery, 2001, p. 530–535. [En ligne]. Disponible : https://doi.org/10.1145/378239.379017

[7] A. Biere, "Picosat essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, n°. 2-4, p. 75–97, 2008.

[8] R. M. Haralick et G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence*, vol. 14, n°. 3, p. 263–313, 1980. [En ligne]. Disponible : https://www.sciencedirect.com/science/article/pii/000437028090051X

[9] F. Boussemart, F. Hemery, C. Lecoutre et L. Sais, "Boosting systematic search by weighting constraints." 01 2004, p. 146–150.

[10] J. Pearl, *Reverend Bayes on inference engines : A distributed hierarchical approach*. Cognitive Systems Laboratory, School of Engineering and Applied Science . . ., 1982.

[11] C. E. Shannon, "A mathematical theory of communication," *The Bell System Technical Journal*, vol. 27, n°. 3, p. 379–423, July 1948.

[12] E. C. Freuder, "A sufficient condition for backtrack-free search," *Journal of the ACM (JACM)*, vol. 29, n°. 1, p. 24–32, 1982.

[13] R. Dechter et I. Meiri, "Experimental evaluation of preprocessing algorithms for constraint satisfaction problems," *Artificial Intelligence*, vol. 68, n°. 2, p. 211–241, 1994.

[14] C. Bessiere et J.-C. Régin, “Mac and combined heuristics : Two reasons to forsake fc (and cbj ?) on hard problems,” 08 1996, p. 61–75.

[15] P. Refalo, “Impact-based search strategies for constraint programming,” dans *Principles and Practice of Constraint Programming – CP 2004*, M. Wallace, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2004, p. 557–571.

[16] G. Pesant, C.-G. Quimper et A. Zanarini, “Counting-based search : Branching heuristics for constraint satisfaction problems,” *Journal of Artificial Intelligence Research*, vol. 43, p. 173–210, 2012.

[17] G. Pesant, “Counting solutions of csps : A structural approach,” dans *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, ser. IJCAI’05. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005, p. 260–265.

[18] S. Brockbank, G. Pesant et L.-M. Rousseau, “Counting spanning trees to guide search in constrained spanning tree problems,” dans *Principles and Practice of Constraint Programming*, C. Schulte, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, p. 175–183.

[19] G. Pesant et C.-G. Quimper, “Counting solutions of knapsack constraints,” dans *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, L. Perron et M. A. Trick, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 203–217.

[20] A. Zanarini et G. Pesant, “Solution counting algorithms for constraint-centered search heuristics,” dans *Principles and Practice of Constraint Programming – CP 2007*, C. Bessière, édit. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 743–757.

[21] G. Pesant, “Counting-based search for constraint optimization problems,” dans *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[22] L. Michel et P. Van Hentenryck, “Activity-based search for black-box contraint-programming solvers,” vol. abs/1105.6314, 05 2011.

[23] D. Habet et C. Terrioux, “Conflict history based search for constraint satisfaction problem,” dans *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC ’19. New York, NY, USA : Association for Computing Machinery, 2019, p. 1117–1122. [En ligne]. Disponible : https://doi.org/10.1145/3297280.3297389

[24] P. Vilím, P. Laborie et P. Shaw, “Failure-directed search for constraint-based scheduling,” dans *Integration of AI and OR Techniques in Constraint Programming*, L. Michel, édit. Cham : Springer International Publishing, 2015, p. 437–453.

[25] P. Laborie et D. Godard, “Self-adapting large neighborhood search : Application to single-mode scheduling problems,” *Proceedings MISTA-07, Paris*, vol. 8, 2007.

[26] K. P. Murphy, *Machine learning : a probabilistic perspective*. MIT press, 2012.