Une simple heuristique pour rapprocher DFS et LNS pour les COP

Julien Vion **Sylvain Piechowiak**

Université de Valenciennes et du Hainaut Cambrésis
LAMIH CNRS UMR 8201
{julien.vion, sylvain.piechowiak}@univ-valenciennes.fr

Résumé

Dans cet article, nous montrons comment une combinaison de stratégies de branchement et de redémarrages pour la recherche en profondeur d'abord (DFS) permet de reproduire le fonctionnement de la recherche par grand voisinage (LNS) pour la résolution de problèmes d'optimisation à contraintes, ce qui permet de rapprocher considérablement les deux techniques. En particulier, nous pouvons implémenter une stratégie DFS qui bénéficie des propriétés de passage à l'échelle de LNS tout en étant capable de prouver l'optimalité des solutions.

Abstract

In this paper, we argue that a combination of well-known and new search strategies enables Depth First tree Search (DFS) to mimic the behavior of Large Neighborhood Search (LNS), which reduces considerably the gap between the two techniques. In particular, we are able to implement a DFS strategy that shares the scalability properties of LNS, but the optimality of solutions can eventually be proven.

1 Introduction

Les stratégies de recherche utilisées pour résoudre les problèmes d'optimisation à contraintes (COP pour *Constraint Optimization Problem*) peuvent être complètes ou incomplètes. Les méthodes complètes sont généralement basées sur la recherche arborescente en profondeur d'abord (DFS pour *Depth-First Search*), guidée par une heuristique de branchement comme $^{\mathrm{dom}}/_{\mathrm{wdeg}}$ [2], couplée avec des techniques de filtrage comme la consistance d'arc ou encore la consistance aux bornes.

La recherche par grand voisinage (LNS pour *Large Neighborhood Search*) est une stratégie de recherche hybride [16] qui réalise des « déplacements » itératifs de manière similaire à une recherche locale, mais utilise une DFS et la propagation de contraintes pour améliorer la meilleure solution connue [17]. L'idée de LNS est de *relâcher* la meilleure solution connue en restaurant le domaine d'une partie de ses variables. Le « *fragment* » obtenu est alors *réoptimisé* par une DFS. Comme la plupart des stratégies incomplètes, LNS passe bien mieux à l'échelle qu'une DFS classique, mais elle ne peut pas prouver l'optimalité d'une solution (ou l'inconsistance d'un problème). De plus le choix des fragments à réoptimiser est une tâche difficile et dépendante du problème à traiter. Seuls quelques rares travaux ont été réalisés par le passé pour concevoir des heuristiques générales de sélection de voisinage (e. g., [8, 10, 13]), et peu d'expérimentations ont été menées sur de grandes variétés de problèmes.

Dans cet article, nous présentons une combinaison de stratégies de recherche : certaines, bien connues, sont décrites dans la section 2. D'autres, nouvelles, sont décrites dans la section 4. Ces stratégies permettent à DFS de se comporter de manière très similaire à LNS, comme nous l'expliquons en section 3. L'objectif est à la fois d'améliorer le passage à l'échelle de DFS sur des problèmes d'optimisation de grande taille, et d'éviter les principaux inconvénients de LNS que nous venons de décrire. Des expérimentations sur le problème de *Steel Mill Slab* ainsi que sur les instances du challenge MiniZinc 2016 sont présentées en section 5.

2 Éléments sur DFS

Un modèle d'optimisation discrète à contraintes est défini par un ensemble de n variables, chacune pouvant être instanciée à l'une des d valeurs d'un ensemble

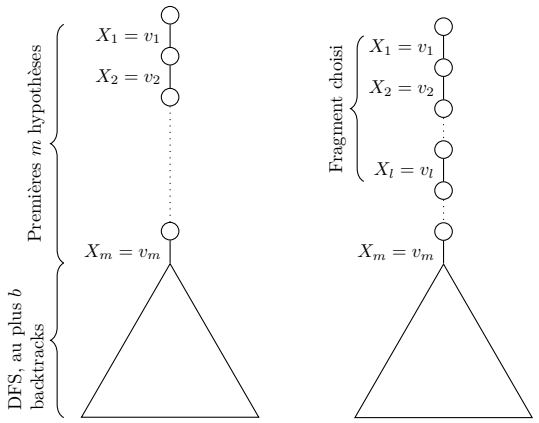
non-vide donné, ainsi qu'un ensemble de contraintes. Chacune d'elles définit les instanciations autorisées d'un sous-ensemble des variables du problème. Le problème d'optimisation discrète à contraintes (COP pour *Constraint Optimization Problem*) consiste à trouver une solution, c'est-à-dire une instantiation de toutes les variables du problème, qui maximise ou minimise la valeur d'une variable particulière, dite « de *coût* ». Ce problème est NP-difficile en général.

Une DFS combinée à la propagation des contraintes et une bonne heuristique d'affectation des variables est la stratégie la plus populaire pour résoudre les problèmes de satisfaction et d'optimisation de contraintes. La propagation est considérée comme le principal atout de la programmation par contraintes. Cette technique est particulièrement efficace en environnement « boite noire », c'est-à-dire n'autorisant pas une adaptation des algorithmes à un problème spécifique.

Pour résoudre une instance de COP, on se ramène généralement à une série de problèmes de décision. On commence par rechercher une solution au modèle, puis on réduit le domaine de la variable de coût pour engager la recherche d'une *meilleure* solution. On s'arrête quand on peut prouver qu'il n'existe plus de meilleure solution. La dernière solution est optimale.

On peut implémenter une DFS basée sur un arbre binaire. Lorsque la phase de propagation est terminée, l'espace de recherche restant est divisé en deux parties en appliquant soit une hypothèse logique, soit sa négation [6]. Cette hypothèse est choisie par une heuristique notée H_{DFS}. Le plus souvent, elle consiste à affecter une valeur à une variable. Si aucune solution n'est trouvée, on supprime la valeur du domaine de la variable. Le choix de la variable à instancier est extrêmement important, et a un impact énorme sur la taille de l'arbre de recherche. Une des heuristiques les plus efficaces est $^{\text{dom}}/_{\text{wdeg}}$ [2], bien que des stratégies inspirées par les solveurs SAT CDCL [11] semblent dominer les challenges MiniZinc depuis quelques années [12, 20]. Le choix de la valeur à affecter, cependant, a toujours été considéré comme de bien moindre impact. Il y a peu de littérature sur le sujet et les quelques tentatives semblent avoir eu des succès mitigés [7].

La principale faiblesse de DFS est son manque de flexibilité. L'arbre de recherche a une taille potentiellement exponentielle, et les hypothèses de branchement réalisées en haut de l'arbre ont peu de chances d'être reconsidérées. Une DFS binaire permet de changer de variable à affecter après un backtrack. Ceci permet restreindre les mauvais choix heuristiques en début de recherche. Cependant, la technique de diversification la plus populaire consiste à *redémarrer* la recherche de manière périodique, à condition que les heuristiques ne sélectionnent pas systématiquement les variables et



(a) DFS avec une limite de backtracks (b) LNS

FIGURE 1 – DFS *vs* LNS

valeurs dans le même ordre [5]. $^{\text{dom}}/_{\text{wdeg}}$ fait partie des heuristiques dites *adaptatives*, qui « apprennent » la structure du problème au cours de la recherche en fonction des conflits rencontrés. Les redémarrages permettent d'exploiter au plus vite l'information apprise. Ajouter une part d'aléatoire dans les heuristiques peut également permettre une diversification.

Les redémarrages peuvent être déclenchés selon des critères très différents. Le plus simple est de fixer une limite du nombre de backtracks à b. On augmente graduellement cette limite après chaque redémarrage, par exemple suivant une progression géométrique [22]. Ainsi, la limite peut devenir supérieure au nombre de backtracks nécessaires pour résoudre le problème, ce qui rend la recherche complète. Cependant, une progression géométrique tend à augmenter b très rapidement, ce qui limite le nombre global de redémarrages à un nombre logarithmique en fonction du nombre total de backtracks. Une stratégie plus aggressive se base sur la série Luby [9] de type 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8… Cette stratégie est également complète, et le nombre de redémarrages est linéaire par rapport au nombre total de backtracks. La communauté SAT a récemment développé une littérature autour de stratégies dynamiques, plus sophistiquées, influencées par les techniques d'apprentissage de clauses qui sont encore mal établies en CP [1].

3 Rapprocher DFS et LNS

Avant de décrire notre contribution, nous faisons une observation décrite par la fig. 1 : la forme d'un arbre de recherche obtenu en utilisant DFS avec redémarrages (i. e., avec une limite du nombre de back-

tracks b) est très similaire à une itération de LNS. Notons X_i (resp. v_i) la $i^{\text{ème}}$ variable (resp. valeur) choisie par l'heuristique de choix de variable (resp. valeur). Nous définissons m comme le dernier niveau de l'arbre de recherche où aucun backtrack n'a eu lieu, i. e., l'hypothèse $X_m = v_m$ a été testée mais sa consistance n'a pu être décidée en raison de la limite du nombre de backtracks. Si et seulement si $m = 0$, l'ensemble de l'espace de recherche a été exploré.

Pour DFS (fig. 1a), les m premières instanciations sont choisies suivant l'heuristique H_{DFS}, puis un arbre de recherche d'au plus b backtracks est exploré. La recherche est interrompue si une solution est trouvée ou si la limite de backtracks est atteinte. Il y a un lien très clair entre m et b, de l'ordre de $b \in O(d^{n-m})$: plus b est grand, plus m sera faible et *vice-versa*.

Pour LNS (fig. 1b), les variables X_1 à X_l constituent le « fragment » à explorer et les valeurs correspondant à la meilleure solution connue leurs sont affectées. La taille et les variables du fragment sont choisies par une heuristique H_{LNS}. Puis une DFS est exécutée sur le reste de l'espace de recherche, en utilisant H_{DFS}. On fixe également une limite b pour éviter de bloquer sur un sous-problème trop difficile et pour permettre de diversifier la recherche même si l est trop faible. La recherche est interrompue si une solution est trouvée, si le nombre maximal de backtracks fixé est atteint ou si le sous-problème est prouvé inconsistant. Par définition de l'algorithme LNS, on a $m \geq l$. On aura $m = l$ si et seulement si le sous-problème est prouvé inconsistant, ce qui nécessite $b \in O(d^{n-l})$. On peut dégénérer LNS en un DFS si $l = 0$; dans le cas contraire on aura $m > 0$: LNS est un algorithme incomplet.

La principale différence entre les deux stratégies est qu'avec LNS, les l premières hypothèses consistent à instancier le fragment choisi par H_{LNS} avec les valeurs de la meilleure solution connue, alors que pour DFS la stratégie de branchement est *a priori* homogène [1]. Notons également qu'il est possible que le fragment choisi par LNS soit immédiatement détecté comme inconsistant par propagation. Une variante nommée PGLNS pour *Propagation-Guided* LNS instancie et propage chaque variable du fragment de manière itérative pour restreindre ce cas de figure, et exploiter l'impact de la propagation pour construire les fragments [13]. Cette dernière stratégie rapproche plus encore LNS de DFS. Concernant les stratégies de redémarrage pour LNS, il est habituel de la faire évaluer en fonction de la facilité à trouver une meilleure solution (i. e., b est réduit quand une solution est trouvée, augmenté sinon). Le résultat est alors proche de ce qui est fait en DFS (cf section 2).

Dans la section suivante, nous proposons de construire une stratégie de DFS en exploitant la principale caractéristique de LNS : affecter les valeurs de la meilleure solution connue aux variables. On peut fusionner les heuristiques H_{DFS} et H_{LNS}, et en abandonner la notion de fragment de taille l : on obtient alors une stratégie très proche de LNS à travers une simple heuristique de choix de valeur.

4 L'heuristique de choix de valeur *Best-Solution*

Nous proposons une heuristique de choix de valeur, que nous nommons *Best-Solution* (BS). Combinée à une heuristique de choix de variable quelconque notée H_{var}, elle définit une heuristique de branchement pour DFS. BS requiert également une heuristique de choix de valeurs « de repli » quelconque notée H_{val}.

Soit S la meilleure solution connue du COP, s'il y en a une. Si S est disponible, on note S_i la valeur correspondant à la variable X_i. Si X_i est la variable choisie par H_{var}, alors BS s'opère en deux étapes :

1. Si S est disponible et S_i appartient au domaine courant de X_i, alors choisir S_i.
2. Sinon, choisir une valeur du domaine de X_i selon l'heuristique H_{val}.

Cette stratégie est très simple et très adaptable : on peut utiliser une heuristique connue pour H_{var}, H_{val}, ou concevoir de nouvelles heuristiques adaptées au problème à résoudre. On peut également manipuler H_{var} pour exploiter les heuristiques de choix de fragment couramment utilisées par LNS. La stratégie utilisée pour les restarts est tout aussi libre. Cependant, pour obtenir un comportement plus proche d'une recherche locale, notamment au niveau des caractéristiques de passage à l'échelle, nous pensons qu'il est préférable d'utiliser une politique de redémarrage aggressive, et de biaiser les heuristiques H_{var} et H_{val} pour augmenter la diversification de la recherche.

Pour H_{var}, nous suggérons d'utiliser une heuristique connue et efficace, si possible adaptative, permettant la diversification. Nous avons opté pour $^{\text{dom}}/_{\text{wdeg}}$ dans nos expérimentations, mais il est envisageable de concevoir des heuristiques inspirées par LNS comme *Propagation-Guided* [13] ou *Cost-Impact* [10]. Pour améliorer la diversification, nous suggérons d'utiliser au minimum un départage aléatoire en cas d'égalité. On peut augmenter encore la diversification : nous fixons une probabilité p avec laquelle la variable à affecter sera choisie aléatoirement, au lieu de faire confiance

1. Il reste possible de construire H_{DFS} de sorte que les variables soient choisies selon une politique différente pour les premières variables à affecter, qui pourraient correspondre aux l variables du fragment LNS.

à l'heuristique. Nous appelons cette stratégie « diversification aléatoire ». Si $p = 0$, l'heuristique obtenue est strictement équivalente à H_{var} ; si $p = 1$, le comportement est complètement aléatoire.

En ce qui concerne H_{val}, il y a peu de littérature sur le sujet, et, comme dans la plupart des cas, il est relativement efficace de choisir la borne inférieure du domaine de la variable, c'est-à-dire affecter les valeurs par ordre lexicographique. Cette heuristique est souvent plus efficace qu'une stratégie purement aléatoire car elle correspond à des propriétés des problèmes et des algorithmes de propagation. Une technique populaire issue de SAT, qui pourrait d'ailleurs sembler similaire à notre proposition, est le *Phase Saving* [14]. Elle consiste à choisir en priorité la même valeur que la dernière affectée pour les variables. Cette stratégie ne peut avoir d'impact qu'après un redémarrage, un backjump, ou encore un changement de variable à affecter dans un branchement binaire : après un retour-arrière simple, la dernière valeur affectée n'est généralement plus disponible. L'objectif est de réutiliser ainsi les solutions des sous-problèmes difficiles situés en haut de l'arbre de recherche. Ce n'est pas nécessaire pour SAT, mais une implémentation pour des variables non booléennes nécessite également une heuristique de repli au cas où la dernière valeur affectée ne serait plus disponible. Nos expérimentations en section 5 montrent que BS est plus robuste que le Phase Saving sur les benchmarks évalués.

5 Expérimentations

Nous avons utilisé notre solveur *Concrete 3.3* [21] sur un ensemble de machines équipées de CPU Intel Core i5-6500 @ 3,2 GHz. Notre solveur est implémenté à l'aide du langage et de l'API Scala 2.12.1 et fonctionne sur une JRE Java 8 update 121 Server avec 4 Gio de mémoire de tas autorisée.

5.1 Environnement « boite noire » : le challenge MiniZinc

Évaluer les performances d'algorithmes d'optimisation n'est pas trivial. Trouver la solution optimale est dans la plupart des cas hors de la portée des solveurs : l'objectif est plutôt de trouver la meilleure solution possible dans le temps imparti. Cependant, la qualité des solutions n'est pas comparable d'une instance à l'autre. Nous avons choisi d'utiliser ici le même système d'évaluation que celui conçu pour le challenge MiniZinc, basé sur le système de vote de Borda [18] : « Chaque instance est considérée comme un votant qui trie les solveurs en fonction de la qualité du résultat obtenu. Pour chaque instance, chaque solveur s gagne des points en comparant son résultat par rapport à chaque autre solveur s' sur cette même instance :

— Si s obtient une meilleure solution que s', i. e., la variable de coût est plus proche de l'optimal, ou s prouve l'optimalité alors que s' échoue à le faire dans une limite de 1 000 secondes, il obtient 1 point.

— Si s et s' (avec des temps d'exécution respectifs t et t') donnent un résultat identique, le score obtenu se base sur une comparaison des temps d'exécution $\frac{t'}{t+t'}$, à une exception près :

— Si s donne une solution moins bonne que s', ou encore ne trouve aucune solution sans prouver l'inconsistance de l'instance, il obtient 0 point, même si s' échoue également. »

Le solveur ayant le meilleur score cumulé remporte le challenge. Nous comparons l'efficacité de notre solveur paramétré avec chaque combinaison des stratégies suivantes :

Choix de variable : $^{\text{dom}}/_{\text{wdeg}}$ avec diversification aléatoire, $p = 0$ et 20 %.

Choix de valeur : Lexicographique (Lex), Phase Saving (PS) ou Best-Solution (BS). PS et BS utilisent Lex comme heuristique de repli si la valeur sélectionnée n'est plus disponible.

Strategy de redémarrage : Geometrique (base 100, croissance 20 %) ou Luby (base 100).

Bien sûr, nous voudrions expérimenter plus de stratégies et de combinaisons, certains choix pouvant paraitre arbitraires. Cependant, le nombre de combinaisons croît exponentiellement, les interactions entre les paramètres sont complexes et il est difficile de les analyser séparément. Notons que le système de Borda n'a pas la propriété d'*indépendance des alternatives non-pertinentes*, c'est-à-dire qu'ajouter ou supprimer un paramétrage peut changer le classement des autres solveurs. Nous avons été contraints de réduire le nombre de paramètres pour conserver une certaine lisibilité. Nous pensons que la liste ci-dessus est représentative. De manière un peu surprenante, des résultats préliminaires ont montré que la valeur du paramètre p n'avait que peu d'influence sur les performances, tant qu'il reste compris entre 10 et 50 %.

Nous avons utilisé les 100 benchmarks (5 instances pour chacun des 20 problèmes proposés) du challenge MiniZinc 2016. La principale motivation de ce travail était clairement d'améliorer les résultats du solveur *Concrete* dans les conditions du challenge MiniZinc. *Concrete 3.2* s'était classé 19$^{\text{e}}$ des 22 solveurs en compétition en 2016, en utilisant un paramétrage classique, i. e., $^{\text{dom}}/_{\text{wdeg}}$, $p = 0$, Lex et redémarrages

TABLE 1 – Résultats sur les instances du MiniZinc Challenge 2016 (MZNC) et le problème Steel Mill Slab. Pour chaque catégorie, nous avons mis en valeur les trois meilleurs scores (si un score est inférieur de moins de 5 % au 3e meilleur score, il est également mis en valeur).

H_{var}	$^{\text{dom}}/_{\text{wdeg}}$, $p = 0$						$^{\text{dom}}/_{\text{wdeg}}$, $p = 20\,\%$					
Redémarrages	Geometric			Luby			Geometric			Luby		
H_{val}	Lex	PS	BS	Lex	PS	BS	Lex	PS	BS	Lex	PS	BS
carpet-cutting	9	6	9	7	7	7	37	38	**41**	**48**	**44**	**47**
celar	3	29	35	12	27	31	9	29	**52**	21	**37**	**45**
cryptanalysis*	8	0	8	**16**	0	**16**	9	0	9	**16**	0	**16**
depot-placement	17	**42**	32	8	**39**	29	19	**42**	34	13	**38**	17
diameterc-mst	0	0	0	0	0	0	7	**27**	10	**26**	**29**	**33**
elitserien	0	0	0	0	0	0	0	0	0	0	0	0
filters	23	23	19	23	30	19	27	28	28	**39**	**38**	**34**
gbac	7	29	25	5	25	22	6	**47**	**33**	6	**49**	31
gfd-schedule	20	24	26	**37**	16	26	33	17	**39**	29	13	**39**
java-auto-gen	24	22	26	**31**	16	19	26	11	**36**	**32**	**34**	**32**
mapping	29	**32**	**30**	**33**	24	28	**31**	11	**30**	**31**	26	24
maximum-dag	30	26	30	26	17	22	**35**	23	**40**	24	21	**36**
mrcpsp	18	12	14	13	7	6	31	**33**	30	**33**	30	**38**
nfc	25	25	20	26	26	19	**31**	**31**	**31**	**32**	**32**	**32**
oocsp-racks*	6	13	5	5	5	5	15	16	15	**23**	**25**	**22**
prize-collecting	28	6	31	32	13	**50**	26	3	**38**	35	8	**53**
rcpsp-wet	16	20	28	15	18	24	16	**43**	41	21	**45**	**44**
solbat*	**31**	**31**	**32**	22	14	22	28	20	27	12	10	12
tpp	28	32	**42**	13	16	32	32	**38**	19	29	**39**	10
zephyrus	26	**36**	**38**	25	28	**38**	21	29	31	15	19	23
overall MZNC	350	411	451	346	324	413	439	486	**586**	484	**537**	**587**
steel-mill-slab	1907	528	**2500**	2227	582	**2540**	1890	615	**2720**	2254	671	**2884**

* : Problème de décision

géométriques. *Concrete 3.3* avec la meilleure combinaison testée ($p = 20\,\%$, BS et redémarrages Luby) se serait classé 15e. Les résultats sont présentés sur la table 1. Pour les problèmes de décision indiqués, BS est équivalent à Lex.

Avec BS et $p = 20\,\%$, les stratégies de redémarrage choisies font finalement peu de différence, mais avec des redémarrages Luby le solveur est plus souvent parmi les 3 meilleurs d'une instance donnée : ceux-ci seraient donc plus robustes.

5.2 Problème du Steel Mill Slab

Le problème du Steel Mill Slab Design Problem (CSPLib #38 [4]) est une variante de problème de Bin Packing : les objets sont *colorés* et au plus deux couleurs peuvent être placées dans une même boite (appelée ici *slab*) ; la taille de chaque slab peut être choisie parmi un ensemble donné, et l'objectif du problème est de minimiser la capacité inutilisée des slabs. Bien que le problème présente d'importantes symétries (les slabs sont interchangeables), celles-ci ne sont pas traitées efficacement avec les traditionnelles contraintes d'inégalité. Les premiers travaux publiés résolvant le Steel Mill Slab se basaient sur LNS avec une heuristique de choix de fragment aléatoire, et une taille de fragment fixe [3]. Nous avons ajouté deux contraintes au modèle pour supprimer certaines symétries : si un slab b est vide, alors le slab $b + 1$ doit être vide ; et deux objets identiques (taille et couleur) $i_1 < i_2$ doivent être placés dans des slabs $b_1 \leq b_2$. Nous avons également désactivé le départage aléatoire des égalités de $^{\text{dom}}/_{\text{wdeg}}$ afin de départager en fonction du poids des objets. Des travaux plus récents ont montré qu'une résolution dynamique des symétries était bien plus efficace [19], mais il est impossible de modéliser de telles techniques avec le langage MiniZinc : il faut réaliser une implémentation spécifique au solveur.

Nous avons réalisé des expérimentation sur les 381 instances de SCHAUS et al. [15]. La table 1 montre que notre meilleur paramétrage, utilisant l'heuristique BS, les redémarrages Luby et la diversification aléatoire permet de résoudre les Steel Mill Slab bien plus efficacement qu'une DFS classique. Le Phase Saving est complètement inefficace sur ce problème.

6 Conclusion & perspectives

Dans cet article, nous avons montré qu'une DFS associée à une heuristique de choix de valeurs choisissant prioritairement les valeurs issues de la meilleure solution connue, une heuristique de choix de variables agrémentée d'une diversification aléatoire, et d'une stratégie de redémarrages améliore clairement le comportement d'une DFS classique dans un environnement « boite noire » en conservant, notamment, la complétude. Nous avons comparé le comportement de cette stratégie avec une LNS, ce qui a permis de rapprocher les deux méthodes de recherche.

Dans des travaux futurs, nous souhaitons développer la partie expérimentale afin de mieux analyser la proximité entre LNS et DFS+BS. Nous souhaitons également nous inspirer des travaux sur les heuristiques de choix de fragment en LNS pour dériver de nouvelles heuristiques de choix de variable pour DFS+BS. Ainsi, peut-être que cette nouvelle stratégie se révèlera aussi plus performante pour la résolution de problèmes industriels de très grande taille pour lesquels une approche purement complète reste inefficace.

Références

[1] A. Biere et A. Fröhlich. « Evaluating CDCL Restart Schemes ». In : *Pragmatics of SAT*. 2015.

[2] F. Boussemart, F. Hemery, C. Lecoutre et L. Saïs. « Boosting Systematic Search by Weighting Constraints ». In : *Proc. of the 16th European Conference on Artificial Intelligence (ECAI)*. 2004, p. 146–150.

[3] A. Gargani et P. Refalo. « An Efficient Model and Strategy for the Steel Mill Slab Design Problem ». In : *Proc. of the 13th Itl. Conf. on Principles and Practice of Constraint Programming (CP)*. Springer, sept. 2007, p. 77–89.

[4] I.P. Gent et T. Walsh. *CSPLib : a benchmark library for constraints*. Rapp. tech. APES-09-1999, 1999. url : http://www.csplib.org.

[5] C.P. Gomes, B. Selman, N. Crato et H. Kautz. « Heavy-tailed phenomena in satisfiability and constraint satisfaction problems ». In : *Journal of Automated Reasoning* 24 (2000), p. 67–100.

[6] J. Hwang et D.G. Mitchell. « 2-way vs d-way branching for CSP ». In : *Proc. of the 11th Itl. Conf. on Principles and Practice of Constraint Programming (CP)*. 2005, p. 343–357.

[7] C. Lecoutre, L. Saïs et J. Vion. « Using SAT Encodings to Derive CSP Value Ordering Heuristics ». In : *JSAT Special Issue on SAT/CP Integration* 1 (2007), p. 169–186.

[8] M. Lombardi et P. Schaus. « Cost Impact Guided LNS ». In : *Proc. of the 11th Itl. Conf. on Integration of AI and OR Techniques in CP (CPAIOR)*. Springer, 2014, p. 293–300.

[9] M. Luby, A. Sinclair et D. Zuckerman. « Optimal speedup of Las Vegas algorithms ». In : *Inform. Process. Lett.* 1993, p. 173–180.

[10] J.-B. Mairy, P. Schaus et Y. Deville. « Generic Adaptive Heuristics for Large Neighborhood Search ». In : *Proc. of the 7th Itl. Workshop on Local Search Techniques in Constraint Satisfaction (LSCS)*. 2010.

[11] J. P. Marques Silva et K. A. Sakallah. « GRASP – A new search algorithm for satisfiability ». In : *Proc. of Itl. Conf. on Computer Aided Design*. Nov. 1996, p. 220–227.

[12] O. Ohrimenko, P. J. Stuckey et M. Codish. « Propagation via Lazy Clause Generation ». In : *Constraints* 14.3 (sept. 2009), p. 357–391.

[13] L. Perron, P. Shaw et V. Furnon. « Propagation Guided Large Neighborhood Search ». In : *Proc. of the 10th Itl. Conf. on Principles and Practice of Constraint Programming (CP)*. 2004, p. 468–481.

[14] K. Pipatsrisawat et A. Darwiche. « A Lightweight Component Caching Scheme for Satisfiability Solvers ». In : *Proc. of the 10th Itl. Conf. on Theory and Applications of Satisfiability Testing (SAT)*. Springer, 2007, p. 294–299.

[15] P. Schaus, P. Van Hentenryck, J.-N. Monette, C. Coffrin, L. Michel et Y. Deville. « Solving Steel Mill Slab Problems with Constraint-Based Techniques : CP, LNS, and CBLS ». In : *Constraints* (2011).

[16] B. Selman, H. Kautz et D. McAllester. « Ten challenges in propositional reasoning and search ». In : *Proc. of the 15th Itl. Joint Conference on Artificial Intelligence (IJCAI)*. 1997.

[17] P. Shaw. « Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems ». In : *Proc. 4th Itl. Conf. on Principles and Practice of Constraint Programming (CP)*. 1998, p. 417–431.

[18] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack et J. Fischer. « The MiniZinc Challenge 2008–2013 ». In : *AI Magazine* 35.2 (2014), p. 55–60.

[19] P. Van Hentenryck et L. Michel. « The Steel Mill Slab Design Problem Revisited ». In : *Proc. of the 5th Itl. Conf. on Integration of AI and OR Techniques in CP*. T. 5015. 2008, p. 377–381.

[20] M. Veksler et O. Strichman. « Learning General Constraints in CSP ». In : *Proc. 12th Itl Conf on Integration of AI and OR techniques in CP (CPAIOR)*. 2015.

[21] J. Vion. *Concrete : a CSP Solving API for the JVM*. `http://github.com/concrete-cp`. 2006–2016.

[22] T. Walsh. « Search in a small world ». In : *Proc. 16th Itl Joint Conf on Artificial Intelligence (IJCAI)*. 1999.