Solveurs de Contraintes Autonomes

Hugues Wattez

UNIVERSITÉ D'ARTOIS

ÉCOLE DOCTORALE SCIENCES TECHNOLOGIES ET SANTÉ N°585

Solveurs de Contraintes Autonomes

THÈSE

présentée et soutenue publiquement le 9 décembre 2021

en vue de l'obtention du

Doctorat de l'Université d'Artois

(Spécialité Informatique)

par

Hugues Wattez

Composition du jury

Présidente :	Marie-José Huguet	INSA, Toulouse
Rapporteurs :	Thomas Schiex	INRAE, MIA, Toulouse
	Cyril Terrioux	Université d'Aix-Marseille
Directeurs :	Frédéric Koriche	Université d'Artois, Lens
	Christophe Lecoutre	Université d'Artois, Lens
Examinateur :	Rémy Bardenet	Université de Lille
Encadrante :	Anastasia Paparrizou	CNRS, Lens
Invité·e·s :	Sébastien Tabary	Université d'Artois, Lens
	Sonia Vanier	LIX, Université Paris1

Région Hauts-de-France

Mise en page avec memcril (B. Mazure, CRIL) et thloria (D. Roegel, LORIA).

Remerciements

Merci, tout d'abord, à ceux que nous appelons communément au sein des doctorants du CRIL : les parents académiques. Chacun pour leur domaine de recherche, merci Christophe [1] et Fred, mes directeurs de thèse, de m'avoir prodigué une part de vos larges connaissances dans les domaines de la programmation par contraintes et de l'apprentissage. Merci Anastasia, mon encadrante de thèse, de m'avoir partagé tes connaissances mixtes en ces deux domaines et plus largement à travers nos nombreux échanges multi-lingues ; heureux d'avoir été cet enfant académique « né » entre tes deux enfants biologiques. Merci à vous trois de m'avoir accueilli et accompagné dans ce monde de la recherche.

Evidemment, je souhaite remercier les membres de mon jury de thèse. Je remercie la présidente de ce jury, Marie-José Huguet, que j'ai eu la chance de rencontrer dans le monde virtuel des JFPC'21. Pour leurs riches et nombreux conseils pour « polir » ce manuscrit, je remercie les rapporteurs de ce jury : Thomas Schiex et Cyril Terrioux. Merci à Rémy Bardenet, examinateur de ce jury, pour avoir suivi cette thèse depuis les différents comités de suivi jusqu'à sa défense. Enfin, merci aux invités de ce jury : Sébastien pour ton soutien et tes idées au sujet de cette thèse ; Sonia pour me permettre de prolonger cette aventure au sein de ton équipe, pour ce magnifique accueil au sein de celle-ci et pour les courantes et futures idées de recherche à l'intersection de nos domaines.

Revenons quelques temps au préalable de ce doctorat pour remercier les différents professeurs m'ayant initié au domaine de l'informatique et par la suite au domaine de la recherche en intelligence artificielle. Ces mêmes professeurs, devenus collègues entre temps, avec qui j'ai eu la chance de partager la passion de l'enseignement.

Comment oublier de remercier ceux qui, au même titre que les liens académiques unissant encadrants et doctorants, représentent la sororité et fraternité académique : mes amis doctorants. Grâce à votre présence tout au long de cette aventure, la monotonie n'a finalement eu aucune chance de s'installer : de l'art plastique (pour enfants) aux gâteaux d'anniversaire [2], de la naissance d'une équipe pour participer aux compétitions algorithmiques au prolongement vers des travaux de recherche, des sorties dans notre pub local en passant par la cafétaria crilienne, les repas et échanges multi-culturels, . . . , jusqu'aux terrils jumeaux loossois [3] (de jour comme de nuit).

Immanquablement, je conclus cette liste non-exhaustive de remerciements en citant, cette fois-ci, ma famille biologique. Parents, grand-parents, frère, belle-sœur, marraine, parrain, tante, oncle, cousins, . . . , merci pour tout votre soutien, votre présence et votre aide sans qui et sans quoi ce périple [4] aurait été bien plus insurmontable. [*]

1. Ou plutôt *Cristophe*, en représailles et pour compenser tous ces *h* ajoutés à mon prénom.
2. Avec la croissance exponentielle du nombre de doctorants, bon courage pour le passage au prochain pallier des 2^{k+1} parts à découper.
3. Il s'agit du gentilé de la commune de Loos-en-Gohelle et non de la prononciation locale du gentilé lensois.
4. Un périple souvent associé à un marathon tant on y « crâche ses poumons ».

∗. Une dernière note de pied de page pour essayer d'obtenir le tant convoité modèle de l'œuf au sein de cette section. Il est vrai qu'il n'est pas courant d'utiliser cette extrêmité de page pour venir à bout de ce modèle, mais cela rappelle de loin la manière qu'a un ordinateur de remplir deux extrêmités de sa mémoire concernant la pile d'exécution et le tas. Bien que l'on travaille sur des solveurs permettant de contrecarrer les contraintes du quotidien, il est toujours aussi tentant d'en produire au sein d'un écrit : non seulement de part ce modèle de l'œuf visant à compléter entièrement cette section, mais aussi de part le fait que les premières lettres de chacun de ces paragraphes rappellent le mot clé le plus usité de ce passage du manuscrit.

« The most beautiful thing
we can experience is the mysterious.
It is the source of all true art and all science. »

- Albert Einstein

Table des matières

Annexes

Notations et Acronymes

Notation (Générale)

$1\,000\,000.1$	nous choisissons la représentation des nombres avec un espace insécable pour chaque ordre de grandeur (milliers, millions, *etc.*) et un point séparant la partie entière de la partie décimale (afin d'être en accord avec les figures produites dans ce document).
↻	lien vers une expérience reproductible.
$\mathbb{N}$	ensemble des entiers naturels.
$\mathbb{R}$	ensemble des réels. 36
$\mathbb{E}$	espérance mathématique. 38
$\mathbb{1}_{\alpha}$	ternaire retournant 1 si α est vrai, 0 sinon.
$O(x)$	borne supérieure asymptotique en x. 21
$\prod_{i=1}^{r} S_i$	produit cartésien correspondant à $S_1 \times S_2 \times ... \times S_r$. 11
`exp`	fonction exponentielle. 24
`rexp`	suite exponentielle basée sur la fonction exponentielle. 24
`luby`	suite de Luby. 24

Notation (Programmation par Contraintes)

x, y, z	(et u, v, w) variables. 10
a, b, c	valeurs. 10
τ	tuple de valeurs. 11
$\tau[i]$	$i$ème valeur du tuple τ. 11
$\tau[x]$	valeur de x du tuple τ. 20
c	contrainte. 12
$\texttt{dom}^{\texttt{init}}(x)$	domaine initial d'une variable x. 10
$\texttt{dom}(x)$	domaine courant d'une variable x. 10
$\texttt{fut}(X)$	ensemble de futures variables (non-fixées) d'un ensemble X de variables. 10
$\texttt{scp}(c)$	portée (ou *scope*) d'une contrainte c. 12
$\texttt{rel}(c)$	relation d'une contrainte c. 12
$\mathcal{P}$	réseau de contraintes. 14
$\mathcal{X} = \texttt{vars}(\mathcal{P})$	ensemble de variables du réseau de contraintes $\mathcal{P}$. 14
$\mathcal{C} = \texttt{ctrs}(\mathcal{P})$	ensemble de contraintes du réseau de contraintes $\mathcal{P}$. 14
$\mathcal{S} = \texttt{sols}(\mathcal{P})$	ensemble des solutions du réseau de contraintes $\mathcal{P}$. 15

$\mathcal{O} = \texttt{obj}(\mathcal{X})$	objectif du réseau de contraintes $\mathcal{P}$. 16
$\texttt{size}(\mathcal{P})$	taille réseau de contraintes $\mathcal{P}$. 14
I	instanciation. 14
$I[X]$	projection de I sur X. 15
δ	décision. 16
Δ	(et Σ) ensemble de décisions. 16
$\texttt{pos}(\Sigma)$	ensemble des décisions positives de Σ. 25
$\texttt{neg}(\Sigma)$	ensemble des décisions négatives de Σ. 25
$\mathcal{P}\|_\delta$	réseau de contraintes $\mathcal{P}$ restreint par la décision δ. 21
n	nombre de variables du réseau de contraintes $\mathcal{P}$. 17
e	nombre de contraintes du réseau de contraintes $\mathcal{P}$. 17
o	taille du plus grand domaine de variable du réseau de contraintes $\mathcal{P}$. 17
r	plus grande arité de contrainte du réseau de contraintes $\mathcal{P}$. 17
ϕ	opérateur appliquant une propriété de cohérence. 20
$\mathcal{T}$	arbre binaire de recherche. 22
t	nombre de *runs* courant lors d'une résolution. 24
$\texttt{restart}(t)$	nombre d'*étapes* avant le redémarrage du *run* t. 24
T	nombre de *runs* total lors d'une résolution. 24
h	une heuristique. 32
$h_1 \circ h_2$	aggrégation de deux heuristiques ($/$, $\times$, $+$, $-$). 32
$h_1 > \cdots > h_m$	chaînage d'heuristiques suivant le principe de *tie-breaking*. 32

Notation (Bandits Multi-Bras)

t	essai/tour actuel du bandit. 36
$[T] = \{1, ..., T\}$	ensemble allant de 1 à T. 36
T	nombre d'essais disponibles pour le bandit (horizon). 36
i_t	choix de l'action/bras lors de l'essai t. 36
$\mathcal{R}_t$	fonction de récompense au tour t. 36
$\mathcal{R}_t(i_t)$	récompense pour l'action i_t par la fonction $\mathcal{R}_t$. 36
$\mathcal{R}$	ensemble des fonctions de récompense. 36
$\mathcal{B}$	algorithme répondant au problème de MAB. 36
$n_t(i)$	nombre de fois où l'action i est sélectionnée durant les t premiers tours. 37
$\hat{\mathcal{R}}_t(i)$	moyenne empirique de $\mathcal{R}(i)$ sur les $n_t(i)$ sélections de i. 37
$\mu(i)$	moyenne réelle des récompenses de l'action i.
$\texttt{regret}_T$	regret cumulé pendant T tours. 37
$\texttt{pseudo-regret}_T$	pseudo-regret cumulé pendant T tours. 38

Acronymes

ABD	*Aggressive Bound Descent* (Descente Agressive de Borne). 110,
AC	*Arc-Consistency* (Cohérence d'Arc). 20,
auvr	*Average of Unassigned Variable Ratios* (Moyenne des Ratios des Variables Non-Assignées). 59,

CN *Constraint Network* (Réseau de Contraintes). 14,
COP *Constraint Optimization Problem* (Problème d'Optimisation sous Contraintes). 16,
CP *Constraint Programming* (Programmation par Contraintes). 9,
CSP *Constraint Satisfaction Problem* (Problème de Satisfaction de Contraintes). 9, 14,

EQ *EQuivalence policy* (Politique d'équivalence). 90,
esb *Explored Sub-Tree* (Sous-Arbre Exploré). 60,
EXP3 *Exponential-weight algorithm for Exploration and Exploitation* (Algorithme à Pondération Exponentielle pour l'Exploration et l'Exploitation). 41,

MAB *Multi-Armed Bandit* (Bandit Multi-Bras). 35,
MOSS *Minimax Optimal Strategy in the Stochastic case* (Stratégie Optimale Minimax dans le cas Stochastique). 39,

NOD *NODe-based policy* (Politique basée sur les nœuds (de la recherche)). 67,

PER *PERturbation policy* (Politique de PERturbation). 91,
prob *PROBing policy* (Politique d'échantillonnage). 90,
pts *Pruned Tree Sizes* (Taille des Arbres Élagués). 59,

RES *REStart-based policy* (Politique basée sur les redémarrages). 68,

SP *Static Policy* (Politique Statique). 92,
ST *Single Tournament* (Tournoi). 81,

TS *Thompson Sampling* (Échantillonnage de Thompson). 40,

UCB *Upper Confidence Bound* (Borne Supérieur de Confiance). 39,

VBS *Virtual Best Solver* (Meilleur Solveur Virtuel). 58,

Introduction Générale

La programmation informatique est le processus de conception et de réalisation d'un programme informatique exécutable afin d'obtenir un résultat spécifique pour un problème donné. En programmation, plusieurs paradigmes permettent de conceptualiser un programme. Ces paradigmes peuvent se partager en deux grandes familles : le *programmation impérative* et la *programmation déclarative*. Pour ce premier paradigme, l'utilisateur use de son expérience en algorithmique pour indiquer à la machine comment résoudre le problème ; la programmation procédurale et objet appartiennent à cette famille. Pour le second paradigme, l'utilisateur se contente de déclarer le problème et les propriétés du résultat souhaité, mais pas la manière de le calculer. La programmation fonctionnelle, la programmation logique et, notamment celle nous intéressant dans ce manuscrit, la *programmation par contraintes*, font partie de ce paradigme déclaratif.

Les langages correspondant au paradigme déclaratif n'indiquent pas l'ordre dans lequel les opérations doivent être exécutées. Au lieu de cela, ils fournissent un certain nombre d'opérations mises à disposition de l'utilisateur, ainsi que les conditions dans lesquelles chacune est autorisée à s'exécuter. Un programme générique, appelé *solveur*, interprète le modèle proposé par l'utilisateur et se charge de trouver une solution grâce aux mécanismes « intelligents » dont il est composé.

La programmation par contraintes (CP — *Constraint Programming*) [MONTANARI 1974, APT 2003, ROSSI *et al.* 2006, LECOUTRE 2009] permet la résolution de problèmes combinatoires de grande taille tels que les problèmes de planification et d'ordonnancement. Elle s'appuie sur un large éventail de techniques issues de l'*intelligence artificielle*, de l'*informatique* et de la *recherche opérationnelle*. En programmation par contraintes, une première étape correspond à la modélisation du problème en un *problème de satisfaction de contraintes*, puis une seconde étape correspond à la résolution de ce modèle par un *solveur de contraintes*. Le Graal suivi par la programmation par contraintes correspond à cette citation d'Eugene Freuder : « *the user states the problem, the computer solves it.* ».

Lors de la modélisation, l'utilisateur déclare un ensemble de contraintes et un ensemble de variables de décision (sur lesquelles sont appliquées les précédentes contraintes) menant vers une (possible) solution du problème. Les contraintes diffèrent des instructions courantes des langages de programmation impérative car elles ne cherchent pas à spécifier une étape ou une séquence d'étapes à exécuter, mais plutôt les propriétés d'une solution à trouver. Lors de la résolution, le solveur s'aide de mécanismes d'*inférence* permettant de filtrer et réduire l'espace de recherche à chaque décision (par exemple, l'assignation d'une valeur à une variable), mais aussi de mécanismes *heuristiques* permettant de prendre une décision et guider la recherche. En plus de la modélisation, l'utilisateur doit également spécifier chacun de ces mécanismes afin de résoudre efficacement son problème. Dans la philosophie émise par le Graal sus-mentionné, les *solveurs de contraintes autonomes* veulent alléger la charge cognitive supplémentaire causée par le manque d'autonomie actuel des solveurs.

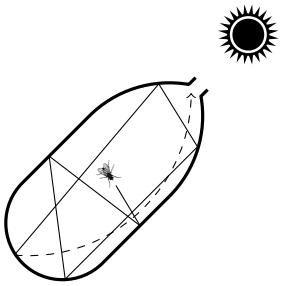
Dans ce manuscrit nous nous concentrons essentiellement sur l'automatisation du choix de l'heuristique à utiliser pour rendre la recherche d'une solution la plus efficace possible, et ce, en délégant cette tâche à un mécanisme d'apprentissage (décrit plus bas). Une heuristique, dans la généralité de sa définition, est une stratégie ou un processus mental simple que les humains, les animaux, ou les machines

utilisent pour former rapidement des jugements, prendre des décisions et trouver des solutions à des problèmes complexes. Lorsqu'une politique optimale de recherche est aussi, voire plus, compliquée que le problème à résoudre, ces méthodes heuristiques peuvent être utilisées pour accélérer le processus de recherche d'une solution satisfaisante. À travers sa définition, et nous le découvrons dans les résultats de ce manuscrit, certaines heuristiques se montrent plus efficaces et utiles que d'autres en fonction du problème donné ; nulle heuristique ne se montre continuellement meilleure que les autres.

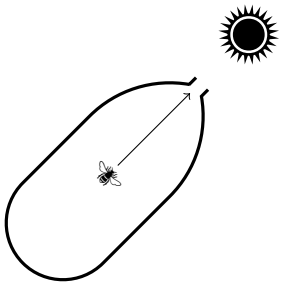
Décontextualisation avec l'analogie des heuristiques de l'abeille et de la mouche

Afin d'illustrer ce principe heuristique, nous proposons une analogie liée au règne animal. Cette analogie concerne deux insectes — la mouche et l'abeille, « emprisonnées » dans une bouteille translucide possédant une porte de sortie : son goulot. Pour ce problème de la bouteille, plusieurs instanciations de la bouteille sont possibles et correspondent simplement à différentes orientations de la bouteille (debout, couchée, renversée, *etc.*).

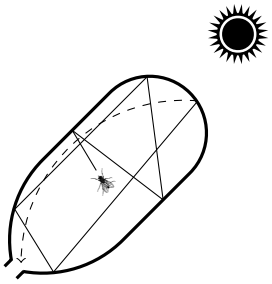
Les abeilles sont connues pour leur intelligence. Elles peuvent apprendre des tâches très complexes, comme soulever ou faire glisser un bouchon pour accéder à une solution sucrée dans un laboratoire, ou même enseigner ce qu'elles ont appris à leurs congénères. Dans le cadre spécifique du problème de la bouteille translucide, son comportement inné l'amène à s'orienter vers la plus grande source lumineuse ; comme le soleil dans son état naturel. Le cas de la mouche est bien plus basique car celle-ci aura tendance à adopter une stratégie plus chaotique et aléatoire quant à la direction à prendre dans le cas du problème de la bouteille. Ainsi, nous venons de décrire les deux heuristiques appartenant à l'abeille et la mouche : l'une s'oriente vers la plus grande source lumineuse tandis que l'autre, sans but fixe, opte pour une orientation aléatoire.



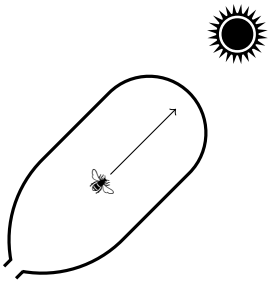
(a) La mouche prend 100 secondes pour résoudre l'instance du problème de la bouteille dont le goulot pointe vers le soleil



(b) L'abeille prend 1 seconde pour résoudre l'instance du problème de la bouteille dont le goulot pointe vers le soleil



(c) La mouche prend 100 secondes pour résoudre l'instance du problème de la bouteille dont le goulot pointe vers le sol



(d) L'abeille ne parvient pas à résoudre l'instance du problème de la bouteille dont le goulot pointe vers le sol

FIGURE 1 – Expérimentation des heuristiques de l'abeille et de la mouche

La figure 1 décrit les expériences (de pensée) proposées à l'abeille et la mouche pour différentes instances du problème de la bouteille. Les deux premières expérimentations correspondent à l'instance de la bouteille où le goulot est orienté vers le soleil et les deux dernières correspondent à la version où le goulot de la bouteille est orienté vers le sol (opposé au soleil).

Lors de la première expérience (figure 1a), la mouche, suivant son heuristique aléatoire, produit des mouvements chaotiques au sein de la bouteille, se cognant sur les rebords puis changeant arbitrairement de direction. L'ensemble des mouvements de la mouche n'est pas décrit dans le graphique, mais est schématisé par le dernier arc en pointillé et pointant vers la sortie. Après tous ces efforts, la mouche finit par trouver la solution au problème, en sortant par le goulot après une centaine de secondes de choix aléatoires. La deuxième expérience (figure 1b) correspondant à la même instance, mais faisant appel cette fois-ci à l'abeille et son heuristique bien plus déterministe. Suivant cette politique innée de suivre le soleil et le goulot étant orienté vers ce dernier, l'abeille trouve la solution au problème en une seule seconde.

La seconde instance donne une toute autre facette à ces expériences. La mouche, comme à son habitude, produit un mouvement aléatoire l'amenant à résoudre le problème en une centaine de secondes (figure 1c). Contre-instinctivement, cette instance amène l'abeille à se retrouver piégée dans le fond de la bouteille, continuant de voler inlassablement vers la lumière jusqu'à ce que, fatalement, celle-ci s'épuise à la tâche (figure 1d).

Usuellement, cette analogie est en lien avec les objectifs humains trop *strictement* définis où, tout comme l'abeille ayant un chemin *strictement* tracé vers un objectif, l'un et l'autre finissent aveuglés par cet objectif manquant de souplesse et de choix arbitraires venant le perturber. Dans notre cas, cette analogie nous permet aussi de mieux appréhender la place des heuristiques dans la recherche dans un espace combinatoire (que ce soit un environnement à trois dimensions ou un environnement à n dimensions/variables de décisions sous contraintes). Nous remarquons, selon le critère que nous observons, que l'une n'est pas strictement meilleure qu'une autre : là où la mouche trouve une solution sur l'ensemble des instances proposées et prend une centaine de secondes par instance, l'abeille a tout de même trouvé une solution sur les deux en une seule seconde. C'est à ce moment précis qu'il serait intéressant de prendre la main sur ces deux possibles heuristiques et de choisir intelligemment celle à suivre.

Une première politique naïve serait, à chaque seconde de l'expérience, d'alterner entre l'une et l'autre heuristique. Ainsi, sans avoir la connaissance du résultat des précédentes expériences, cela nous permettrait de résoudre les deux instances en à peine plus de temps que la meilleure des heuristiques actuelles. Il serait intéressant d'ajouter à cela une manière de discriminer la progression de ces deux heuristiques à mener l'insecte vers la bonne direction : par exemple, prendre en compte la distance séparant l'insecte et le goulot pourrait être un bon indicateur. Avec cette manière de discriminer, nous remarquerions dans le cas de la bouteille au goulot orienté vers le sol, que l'abeille est constamment à une distance extrême du goulot, là où la mouche serait en moyenne à mi-chemin de la sortie. Prenant en compte ce genre de discrimination, existerait-il une politique plus efficace que l'alternance naïve et uniforme entre heuristiques pour mener à bien ce choix d'heuristique et ainsi, la recherche de la sortie/solution ?

Recontextualisation avec les heuristiques au sein des solveurs de contraintes

Au sein des solveurs de contraintes actuels, nous nous retrouvons face à cette même problématique de choix d'heuristique. Au même titre que le problème de la bouteille, il existe une multitude de problèmes soumis au paradigme de la programmation par contraintes, déclinés en instances grâce au paramétrage de ces problèmes. Plusieurs heuristiques, plus ou moins efficaces, existent et aucune n'est continuellement meilleure qu'une autre. Sans entrer dans les détails, nous présentons cinq heuristiques composant les solveurs de contraintes modernes dans le table 1, notées h_i.

La table 1 décrit les performances de chaque heuristique en terme du nombre de résolutions

	h_1	h_2	h_3	h_4	h_5
#RÉSOLUTIONS	422	384	407	529	**543**
TEMPS (S)	408 484	482 069	427 455	140 219	**123 546**
#MEILLEURS TPS	82	74	111	**158**	144
#EXCLUSIVITÉS	4	2	2	11	**15**

TABLE 1 – Performance des heuristiques d'un solveur de contraintes moderne sur 810 instances de 83 problèmes [section 3.4.2]

(#RÉSOLUTIONS), du temps cumulé de résolution (TEMPS (S)), du nombre de fois où l'heuristique obtient le meilleur temps (#MEILLEURS TPS) et enfin, du nombre de fois où seule cette heuristique résout une instance donnée (#EXCLUSIVITÉS). Nous remarquons que les deux dernières heuristiques (h_4 et h_5) sont celles montrant les meilleures contributions. Bien qu'une dominance apparaisse dans cet ensemble d'heuristiques, certains critères dévoilent que les trois premières heuristiques, moins efficaces, apportent tout de même une contribution avec de meilleurs temps de résolution pour une proportion d'instances, ainsi que quelques exclusivités. Nous avons calculé les résultats de la politique optimale, notée h^*, qui, tel un oracle, choisit la meilleure heuristique pour une instance donnée. h^* démontre des performances à hauteur de 569 résolutions tout en divisant par deux le temps de résolution de la meilleure politique actuelle.

À nouveau, nous faisons face au constat qu'aucune heuristique n'est continuellement meilleure. Cela nous amène donc à emprunter la précédente problématique et à l'appliquer à l'échelle de ce manuscrit : « Comment trouver la meilleure heuristique de choix de variable pour une instance de problème, étant donné un ensemble d'heuristiques fournies par le système de résolution ? ».

En réponse à cette problématique, plusieurs défis se profilent : (i) il s'agira, dans un premier temps, de gagner en autonomie et ainsi de réussir à retrouver et égaler les performances de la meilleure heuristique (qu'un expert aurait sélectionné à l'échelle d'un ensemble d'instances) ; (ii) au-delà, il s'agira de dépasser ces précédentes performances et approcher celles de la politique h^* ; (iii) enfin, le Graal de cette problématique serait d'obtenir, voire de dépasser, les résultats de la politique h^*.

Afin de répondre à la problématique et aux défis proposés, ce manuscrit propose un plan composé de deux parties divisé en sept chapitres au total. Une première partie propose l'état de l'art décrivant, dans un premier chapitre, l'avancée de la recherche concernant le paradigme de la programmation par contraintes [VAN BEEK 2006], puis dans un deuxième chapitre, une possible solution à notre problème de sélection d'heuristiques. Ce deuxième chapitre décrit le *problème du bandit multi-bras* [GITTINS 1989], semblable au problème que nous nous posons, et les politiques algorithmiques permettant de répondre efficacement à ces choix auxquels nous faisons face. Ce chapitre s'inscrit dans le domaine de l'*apprentissage automatique* et, plus précisément, celui de l'*apprentissage par renforcement*.

Dans une seconde partie, cinq chapitres sont proposés décrivant les contributions proposées dans ce manuscrit. Le troisième chapitre renforce l'idée de la problématique en mettant en évidence les différentes performances des heuristiques et leur complémentarité. Ce premier chapitre expérimental nous donne l'occasion de présenter notre outils d'extraction et d'analyse des résultats de campagnes expérimentales : *Metrics* [FALQUE *et al.* 2020, FALQUE *et al.* 2021b]. Cette étude des heuristiques préexistantes a permis de premières publications nationale et internationale en affinant le processus de pondération d'une des heuristiques [WATTEZ *et al.* 2019b, WATTEZ *et al.* 2021a]. Au-delà de l'observation des différentes performances de ces heuristiques, ce chapitre nous donne notamment l'occasion d'introduire de nouvelles fonctions capables d'estimer et de récompenser une heuristique donnée pour son efficacité pendant le processus de résolution du solveur : ces fonctions de récompense ont fait l'objet

de deux publications internationales [WATTEZ *et al.* 2020, PAPARRIZOU & WATTEZ 2020] et deux publications nationales [WATTEZ *et al.* 2019a, WATTEZ *et al.* 2021b]. Le quatrième chapitre décrit une première avancée dans la sélection autonome des heuristiques et la convergence vers la meilleure d'entre elles avec les politiques de sélection proposée par la bibliographie liée aux bandits multi-bras. Ce chapitre et le framework proposé font aussi l'objet de ces publications [WATTEZ *et al.* 2020, WATTEZ *et al.* 2019a]. Un cinquième chapitre prend en compte les remarques et limitations qu'imposent l'application stricte des politiques de bandit multi-bras proposées dans la bibliographie dans le cadre précis de nos travaux. Ce chapitre nous donne l'occasion de proposer un nouvel algorithme répondant aux problèmes du bandit multi-bras, au moins, dans le cadre particulier de la sélection d'heuristique en programmation par contraintes (en cours de soumission). Le sixième chapitre s'inspire de l'approche de l'heuristique de choix aléatoire (analogie de la mouche) afin de perturber des heuristiques apprenantes dont le problème est, parfois, leur manque de diversité (analogie de l'abeille). Ce chapitre a fait l'objet d'une publication internationale [PAPARRIZOU & WATTEZ 2020] et a aussi été proposée lors d'une conférence nationale [WATTEZ *et al.* 2021b]. Enfin, un dernier chapitre cherche à étendre ces précédents travaux à un domaine annexe à celui de la recherche de solution satisfaisante : les solveurs de contraintes sous optimisation. Ce chapitre a fait l'objet d'une publication nationale, pour une avancée annexe à celle de la thèse, concernant la progression *agressive* de la recherche de nouvelles solutions s'approchant de l'optimalité [FALQUE *et al.* 2021a]. Plus en lien avec le fil conducteur de la thèse, nous appliquons les précédents travaux d'exploitation de la meilleure heuristique dans ces solveurs cherchant l'optimalité et observons des résultats à l'image de ceux obtenus jusqu'ici.

Contributions

[WATTEZ *et al.* 2019a] H. WATTEZ, F. KORICHE, C. LECOUTRE, A. PAPARIZZOU, ET S. TABARY. « Heuristiques de recherche : un bandit pour les gouverner toutes ». Dans *Actes des 15es Journées Francophones de Programmation par Contraintes (JFPC'19)*, Albi, France, juin 2019.

[WATTEZ *et al.* 2019b] H. WATTEZ, C. LECOUTRE, A. PAPARIZZOU, ET S. TABARY. « Refining Constraint Weighting ». Dans *Proceedings of the 31st International Conference on Tools with Artificial Intelligence (ICTAI'19)*, novembre 2019.

[WATTEZ *et al.* 2020] H. WATTEZ, F. KORICHE, C. LECOUTRE, A. PAPARIZZOU, ET S. TABARY. « Learning Variable Ordering Heuristics with MultiArmed Bandits and Restarts ». Dans *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI'20)*, août 2020.

[PAPARRIZOU & WATTEZ 2020] A. PAPARIZZOU ET H. WATTEZ. « Perturbing Branching Heuristics in Constraint Solving ». Dans *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, septembre 2020.

[FALQUE *et al.* 2020] T. FALQUE, R. WALLON, ET H. WATTEZ. « Metrics : Towards a Unified Library for Experimenting Solvers ». Dans *11th International Workshop on Pragmatics of SAT (POS'20)*, juillet 2020.

[WATTEZ *et al.* 2021a] H. WATTEZ, F. KORICHE, C. LECOUTRE, A. PAPARIZZOU, ET S. TABARY. « Focus sur les heuristiques basées sur la pondération de contraintes ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021.

[WATTEZ *et al.* 2021b] H. WATTEZ, F. KORICHE, ET A. PAPARIZZOU. « Perturbation des heuristiques de branchement dans la résolution de contraintes ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021.

[FALQUE *et al.* 2021a] T. FALQUE, C. LECOUTRE, B. MAZURE, ET H. WATTEZ. « Descente Agressive de la Borne en Optimisation sous Contraintes ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021.

[FALQUE *et al.* 2021b] T. FALQUE, R. WALLON, ET H. WATTEZ. « Metrics : Mission Expérimentations ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021.

Première partie

État de l’art

Chapitre 1

Programmation par Contraintes

Sommaire

1.1 Introduction

La *programmation par contraintes* (CP — *Constraint Programming*) [APT 2003, ROSSI *et al.* 2006, LECOUTRE 2009] est reconnue depuis longtemps comme un robuste paradigme pour la modélisation et la résolution de problèmes combinatoires, avec de nombreuses applications allant de la configuration, la planification et l'ordonnancement à la bio-informatique, la conception de réseaux et la validation de logiciels. Le mot d'ordre de la programmation par contraintes est d'alléger autant que possible la charge cognitive de la recherche combinatoire en utilisant des solveurs de contraintes génériques. Idéalement, l'utilisateur ne doit se concentrer que sur la modélisation de sa tâche comme un ensemble de variables de décision, ainsi qu'un ensemble de contraintes spécifiant les relations entre les variables. Sur la base de cette description, le *problème de satisfaction de contraintes* (CSP — *Constraint Satisfaction Problem*) consiste à trouver une affectation pour toutes les variables satisfaisant l'ensemble des contraintes. La résolution de ce problème doit être dédiée au solveur de contraintes, qui alterne recherche et inférence afin d'explorer efficacement l'espace des instanciations possibles.

À travers ce chapitre, nous définissons dans un premier temps ce qu'est un réseau de contraintes puis la description globale du problème de satisfaction de contraintes. Enfin, nous décrivons les mécanismes mêlant recherche et inférence au sein d'un solveur. Comme prolongement de ce problème, nous décrivons ensuite le principe d'optimisation sous contraintes.

1.2 Réseau de Contraintes

1.2.1 Définitions et notations

La programmation par contraintes est introduite et commence à être formalisée dans les années 1960 et 1970 [MONTANARI 1974]. Dans cette section, nous donnons les définitions et notations usuellement employées pour décrire un réseau de contraintes. Nous commençons par décrire variables et domaines, puis venant lier ces derniers, les relations et les contraintes.

Définition 1 (Variable)

Une *variable* x est une entité à laquelle on associe une valeur. Cette valeur appartient au *domaine (courant)* de x, noté $\texttt{dom}(x)$.

Pour ce paradigme, les variables sont généralement considérées discrètes. Les lettres x, y, z (et parfois u, v, w), possiblement accompagnées d'un indice ou d'un exposant, sont utilisées pour désigner des variables. Par ailleurs, les lettres a, b, c, possiblement accompagnées d'un indice ou d'un exposant, sont utilisées pour désigner des valeurs. Le domaine d'une variable évolue au cours du temps mais est toujours inclus dans le *domaine initial* de la variable, noté $\texttt{dom}^{\texttt{init}}(x)$.

Exemple 1

Soient les variables x et y ayant pour domaine courant $\texttt{dom}(x) = \{a, b\}$ et $\texttt{dom}(y) = \{b, c\}$. Leurs domaines initiaux peuvent être $\texttt{dom}^{\texttt{init}}(x) = \texttt{dom}^{\texttt{init}}(y) = \{a, b, c\}$.

Une variable est dite *fixée* quand son domaine courant contient une seule valeur, et *non-fixée* sinon. La fonction $\texttt{fut}(X)$, pour *futures variables*, permet de récupérer l'ensemble des variables non-fixées (ou *libres*) dans un ensemble de variables X. Une variable peut être fixée *explicitement* ou *implicitement*. Lorsqu'une variable x reçoit explicitement une valeur a de son domaine courant $\texttt{dom}(x)$ au cours de la progression d'un algorithme, on considère que toute autre valeur $b \neq a$ est retirée de $\texttt{dom}(x)$. Dans ce cas, on dit que la variable x est *assignée* ; nous disons également que la valeur a est assignée à x ; dans le cas contraire, on dit que x est *non-assignée*. L'assignation d'une valeur à une variable est appelée *affectation* de variable. Les variables fixées implicitement apparaissent lorsque des mécanismes de déduction (inférence) sont utilisés.

Exemple 2

Considérons l'égalité $x = y$ entre deux variables x et y dont le domaine courant (et commun) est $\{1, 2\}$. Si la valeur 1 est attribuée à la variable x, en raisonnant à partir de l'égalité, nous pouvons déduire que y doit également être égale à 1, c'est-à-dire que la valeur 2 peut être retirée de $\texttt{dom}(y)$ par déduction. Les deux variables sont alors fixées, la première explicitement et la seconde implicitement.

Afin de définir les *contraintes*, les notions de *tuple*, de *produit cartésien* et de *relation* sont nécessaires.

Définition 2 (Tuple)

Un *tuple* τ est une séquence, entourées par des parenthèses, de valeurs séparées par des virgules. Un tuple contenant r valeurs est appelé r-tuple. La $i$ème valeur d'un r-tuple (avec $1 \leq i \leq r$) est notée $\tau[i]$.

Un *produit cartésien* est un ensemble composé de tous les tuples qui peuvent être construits à partir d'une séquence d'ensembles.

Définition 3 (Produit cartésien)

Soient $S_1, S_2, ..., S_r$ une séquence de r ensembles. Le *produit cartésien* $\prod_{i=1}^{r} S_i = S_1 \times S_2 \times ... \times S_r$ correspond à l'ensemble $\{(a_1, a_2, ..., a_r) \mid a_1 \in S_1, a_2 \in S_2, ..., a_r \in S_r\}$. Chaque élément de $\prod_{i=1}^{r} S_i$ correspond à un r-tuple.

Exemple 3

Soient x, y et z trois variables telles que $\texttt{dom}(x) = \texttt{dom}(y) = \{a, c\}$ et $\texttt{dom}(z) = \{a, b\}$. Le produit cartésien de ces trois domaines correspond à :

$$\texttt{dom}(x) \times \texttt{dom}(y) \times \texttt{dom}(z) = \left\{ \begin{array}{l} (a,a,a), \\ (a,a,b), \\ (a,c,a), \\ (a,c,b), \\ (c,a,a), \\ (c,a,b), \\ (c,c,a), \\ (c,c,b) \end{array} \right\}$$

Une *relation* est un sous-ensemble d'un produit cartésien.

Définition 4 (Relation)

Une *relation* R définie par une séquence de r ensembles $S_1, S_2, ..., S_r$ est un sous-ensemble du produit cartésien $\prod_{i=1}^{r} S_i$, autrement dit, nous avons : $R \subseteq \prod_{i=1}^{r} S_i$.

Exemple 4

Soit R_{xyz} définie sur $\texttt{dom}(x) \times \texttt{dom}(y) \times \texttt{dom}(z)$:

$$R_{xyz} = \left\{ \begin{array}{l} (a,a,a), \\ (a,c,a), \\ (a,c,b), \\ (c,c,a) \end{array} \right\} \subset \left\{ \begin{array}{l} (a,a,a), \\ (a,a,b), \\ (a,c,a), \\ (a,c,b), \\ (c,a,a), \\ (c,a,b), \\ (c,c,a), \\ (c,c,b) \end{array} \right\}$$

Nous pouvons maintenant introduire le concept central de *contrainte*.

Définition 5 (Contrainte)

Une *contrainte* c est définie par un ensemble de variables, appelé *portée* (ou *scope*) de c et noté $\texttt{scp}(c)$, et par une relation $\texttt{rel}(c)$ qui décrit l'ensemble des tuples autorisés par c pour les variables de sa portée. Nous avons :

$$\texttt{rel}(c) \subseteq \Pi_{x \in \texttt{scp}(c)} \texttt{dom}(x).$$

La lettre c, possiblement accompagnée d'un indice ou d'un exposant décrivant la séquence des variables de la portée, est utilisée pour indiquer une contrainte. La confusion avec l'usage de la lettre c en tant que valeur est évitée grâce au contexte.

Définition 6 (Arité)

L'*arité* d'une contrainte c correspond au nombre de variables impliquées dans c, c'est-à-dire, $|\texttt{scp}(c)|$. c est une contrainte :

- *unaire* si son arité est égale à 1 ;
- *binaire* si son arité est égale à 2 ;
- *ternaire* si son arité est égale à 3 ;
- *non-binaire* si son arité est strictement supérieure à 2.

Le *degré d'une variable* est une caractéristique qui peut être utile, par exemple, lors de la conception d'une heuristique de choix de variable (voir section 1.4).

Définition 7 (Degré)

Le *degré* d'une variable est le nombre de contraintes l'impliquant. Le *degré dynamique* d'une variable x est le nombre de contraintes impliquant x et au moins une variable non fixée distincte de x.

Bien que nous ayons défini une contrainte en termes de relation associée, cela n'impose aucune restriction sur la définition pratique des contraintes. En première instance, une contrainte peut être définie soit en *intention* ou soit en *extension*.

Définition 8 (Contrainte en intention)

Une contrainte c est en *intention* si elle est décrite par une formule booléenne (prédicat) qui représente une fonction qui est définie de $\prod_{x \in \texttt{scp}(c)} \texttt{dom}^{\texttt{init}}(x)$ vers $\mathbb{B} = \{\text{faux}, \text{vrai}\}$.

Exemple 5

Soient les variables v, w, x, y et z ayant pour domaines $\texttt{dom}(v) = \texttt{dom}(w) = \texttt{dom}(x) = \texttt{dom}(y) = \texttt{dom}(z) = \{0, 1, 2\}$:

- $v = w + 1$ est une contrainte binaire en intention ;
- $(x = y) \wedge (y = z)$ est une contrainte ternaire en intention.

Définition 9 (Contrainte en extension)

Une contrainte c est en *extension* si elle est décrite :

- soit positivement en listant les tuples autorisés par c ;
- soit négativement en listant les tuples interdits par c.

Exemple 6

Si $\texttt{dom}(x) \times \texttt{dom}(y) \times \texttt{dom}(z) = \{0, 1, 2\} \times \{0, 1, 2\} \times \{0, 1, 2\}$, alors la contrainte ternaire de l'Exemple 5 peut être définie positivement comme :

$$(x, y, z) \in \{(0, 0, 0), (1, 1, 1), (2, 2, 2)\}$$

Les *contraintes globales* ont été introduites dans l'optique de faciliter la modélisation et d'améliorer la résolution grâce à des algorithmes dédiés.

Définition 10 (Contrainte globale)

Une *contrainte globale* est un type de contrainte qui représente une sémantique mathématique précise et peut être appliquée sur un nombre arbitraire de variables.

Exemple 7

La sémantique de `allEqual` est telle que toutes les variables prennent la même valeur. La contrainte ternaire de l'exemple 5 peut être définie par :

$$\texttt{allEqual}(x, y, z)$$

Les contraintes globales ont pour vocation de faciliter la modélisation et d'améliorer la résolution grâce à leur algorithme de filtrage dédié. De nos jours, plus de 400 contraintes globales existent [BELDICEANU *et al.* 2007] telles que `allDifferent`, `cumulative`, *etc.*

Finalement, une structure composée de variables et de contraintes est appelée un *réseau de contraintes*.

Définition 11 (Instance CSP)

Une instance $\mathcal{P}$ du *problème (général) de satisfaction de contraintes* (CSP — *Constraint Satisfaction Problem*), aussi appelée *réseau de contraintes* (CN — *Constraint Network*), est définie par :

- un ensemble fini de variables, noté $\mathcal{X} = \texttt{vars}(\mathcal{P})$;
- un ensemble fini de contraintes, noté $\mathcal{C} = \texttt{ctrs}(\mathcal{P})$, chacune portant sur un sous-ensemble de $\mathcal{X}$ tel que $\forall c \in \mathcal{C}, \texttt{scp}(c) \subseteq \mathcal{X}$.

La taille d'un réseau de contraintes $\mathcal{P}$, correspondant aux nombres d'instanciations complètes du problème, est donnée par la fonction $\texttt{size}(\mathcal{P}) = |\prod_{x \in \mathcal{X}} \texttt{dom}(x)|$.

Avant d'en arriver à la définition d'une *solution* d'un réseau de contraintes, les notions de *support* et d'*instanciation* sont nécessaires.

Définition 12 (Support)

Pour une contrainte c :

- un *tuple autorisé*, ou *accepté*, par c est un élément de $T = \texttt{rel}(c)$;
- un *tuple valide* est un élément de $V = \prod_{x \in \texttt{scp}(c)} \texttt{dom}(x)$;
- un *support* (sur c) est un tuple qui est à la fois autorisé et valide, i.e., un élément de $T \cap V$.

Définition 13 (Instanciation)

Une *instanciation* I d'un ensemble $X = \{x_1, \ldots, x_q\}$ de q variables est un ensemble $\{(x_1, v_1), \ldots, (x_q, v_q)\}$ tel que $\forall i (1 \leq i \leq q), v_i \in \texttt{dom}^{\texttt{init}}(x_i)$ et toute variable x_i n'apparaît qu'une seule fois. Une instance I est valide si et seulement si $\forall (x, v) \in I, v \in \texttt{dom}(x)$

La fonction `vars` permet, dans le cas d'une instanciation I, d'y extraire un ensemble de variables tel que $\texttt{vars}(I) = \{x \mid \forall (x, v) \in I\}$.

Définition 14 (Projection d'une instanciation)

Soit un ensemble de variables X et une instanciation I. La *projection* de I sur X, notée $I[X]$, est définie par $I[X] = \{(x, a) | (x, a) \in I \wedge x \in X\}$.

Définition 15 (Instanciation satisfaisante/insatisfaisante)

Une instanciation I couvre une contrainte c si et seulement si $\texttt{scp}(c) \subseteq \texttt{vars}(I)$. Une instanciation I satisfait une contrainte c si et seulement si elle couvre c et que le tuple correspondant à la projection de I sur $\texttt{scp}(c)$ est un support dans c; on dit également que la contrainte c est satisfaite par I. A contrario, l'instanciation I ne satisfait pas la contrainte si elle couvre cette contrainte et que le tuple correspondant à la projection de I sur $\texttt{scp}(c)$ n'est pas autorisé par c (on dit aussi de ce tuple qu'il est en conflit dans c); la contrainte est alors insatisfaite (ou violée) par I.

Définition 16 (Instanciation localement cohérente)

Une instanciation I est *localement cohérente* sur un réseau de contraintes $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ si et seulement si I est valide et que toutes les contraintes couvertes par I sont satisfaites. Si $\texttt{vars}(I) = \mathcal{X}$, I correspond alors à une instanciation localement cohérente complète, sinon elle est partielle.

Définition 17 (Solution d'une instance CSP)

Une *solution d'une instance CSP* $\mathcal{P}$ est une instanciation localement cohérente complète sur $\mathcal{P}$. Autrement dit, cela correspond à l'assignation d'une valeur à chaque variable de $\mathcal{P}$ telle que toutes les contraintes de $\mathcal{P}$ sont satisfaites.

L'ensemble des solutions de $\mathcal{P}$ est noté $\mathcal{S} = \texttt{sols}(\mathcal{P})$. L'instance $\mathcal{P}$ est dite satisfaisable (ou cohérente) si elle admet au moins une solution, c'est-à-dire, si $\mathcal{S} \neq \emptyset$.

D'autres formes d'instanciations sont intéressantes à étudier lors du processus de résolution d'un CSP.

Définition 18 (Instanciation globalement incohérente)

Une instanciation est *globalement incohérente* si elle ne peut pas être étendue à une solution, sinon elle est globalement cohérente. Une instanciation globalement incohérente est appelée aussi *nogood*.

Identifier et enregistrer les *nogoods* permet généralement d'améliorer le processus d'exploration de l'espace de recherche, plus spécifiquement quand les *nogoods* sont petits.

Définition 19 (Décision positive/négative)

Soit $\mathcal{P}$ un réseau de contraintes et (x, a) un couple tel que $x \in \texttt{vars}(P)$ et $a \in \texttt{dom}(x)$. L'affectation $x = a$ est appelée *décision positive* tandis que $x \neq a$ est appelée *décision négative*, ou réfutation.

À l'aide de la précédente définition, nous proposons une définition alternative pour décrire les *nogoods*. Cette définition permet d'introduire plus simplement les nogoods extraits lors du redémarrage de la recherche (section 1.3.3).

Définition 20 (Nogood [DECHTER 1990])

Un *nogood* est un ensemble de d décisions positives δ, de la forme $\Delta = \{\delta_1, \ldots, \delta_d\}$, globalement incohérent.

Le problème classique de satisfaction de contraintes consiste à déterminer si un réseau de contraintes donné est satisfaisable en exhibant une solution le cas échéant. Décider si un CSP est satisfaisable est un problème NP-complet [MACKWORTH 1977]. D'autres tâches combinatoires peuvent présenter un intérêt : énumérer ou compter l'ensemble des solutions, calculer une solution optimale en fonction d'un objectif donné, *etc.* Nous concluons cette partie par la définition de l'une d'entre elles : le problème d'optimisation sous contraintes.

Définition 21 (Instance COP)

Une instance $\mathcal{P}$ du *problème (général) d'optimisation de contraintes* (COP — *Constraint Optimization Problem*) est composée de :

- un ensemble fini de variables, noté $\mathcal{X} = \texttt{vars}(\mathcal{P})$;
- un ensemble fini de contraintes, noté $\mathcal{C} = \texttt{ctrs}(\mathcal{P})$, tel que $\forall c \in \mathcal{C}, \texttt{scp}(c) \subseteq \mathcal{X}$;
- une *fonction objectif* $\mathcal{O} = \texttt{obj}(\mathcal{X})$ devant être maximisée ou minimisée.

Une instance COP peut être interprétée comme une instance CSP à laquelle on associe une fonction objectif. Cette fonction donne une valeur numérique à chacune des solutions de l'instance permettant ainsi de quantifier sa qualité. L'objectif est de trouver la solution maximisant ou minimisant cette fonction. Il s'agit, par exemple, d'une fonction maximisant (resp. minimisant) la somme ou le produit de certaines variables (possiblement associées à des coefficients) du problème.

Définition 22 (Solution d'une instance COP)

Une *solution d'une instance COP* $\mathcal{P}$ correspond à une solution du problème sous-jacent CSP. Pour la minimisation (resp. maximisation), une *solution optimale* de $\mathcal{P}$ est une solution pour laquelle la valeur objectif est inférieure ou égale (resp. supérieure ou égale) à la valeur de toute autre solution.

Un problème d'optimisation se montre au moins aussi difficile qu'un problème de décision ; il ne s'agit plus seulement de trouver une solution mais plutôt de trouver la meilleure solution selon la contrainte objectif. Ainsi, le problème d'optimisation s'inscrit dans la classe des problèmes NP-difficiles.

Avant de conclure cette section, quelques notations supplémentaires décrivant un réseau de contraintes sont proposées ci-après.

Notation 1 (n, e, o et r)

Pour un réseau de contrainte $\mathcal{P} = (\mathcal{X}, \mathcal{C})$, nous notons par :
- n le nombre de variables, $n = |\mathcal{X}|$;
- e le nombre de contraintes, $e = |\mathcal{C}|$;
- o la taille du plus grand domaine des différentes variables, $o = \max_{x \in \mathcal{X}} |\texttt{dom}(x)|$;
- r la plus grande arité des différentes contraintes, $r = \max_{c \in \mathcal{C}} |\texttt{scp}(x)|$.

1.2.2 Exemple du problème de coloration de carte

Le problème de coloration de carte est un exemple connu et emprunté d'un problème plus générique — le problème de coloration de graphe. Pour ce problème, une carte plane découpée en régions est mise à disposition. L'objectif est de colorier chaque région de façon à ce que deux régions adjacentes ne soient pas coloriées de la même couleur tout en minimisant le nombre de couleurs utilisées. Il est connu que ce problème est résolvable avec un maximum de quatre couleurs [WILSON & NASH 2003]. Sous ces contraintes et afin d'illustrer ce problème, nous proposons de colorer la carte d'un pays connu pour ses fêtes colorées : l'Inde [2].

La carte vierge présentée dans la figure 1.1a, accompagnée d'une palette de quatre couleurs, est une instance du problème de coloration de carte. Nous présentons cette instances sous forme d'un réseau de contraintes $\mathcal{P} = (\mathcal{X}, \mathcal{C})$. Comme le montre la figure 1.1a, chaque région est considérée comme étant une variable x_α où α représente le nom simplifié de la région concernée. Chaque variable est composée d'un domaine correspondant aux couleurs disponibles : $\{0, 1, 2, 3\}$. Chacune des valeurs de ces domaines correspond à une couleur précise. De manière plus formelle, les variables et leur domaine sont définis comme suit :

$$\mathcal{X} = \{x_\alpha | \alpha \in \texttt{regions}\}$$

où `regions` représente l'ensemble des régions de la carte et

$$\forall x \in \mathcal{X}, \texttt{dom}(x) = \{0, 1, 2, 3\}.$$

Il suffit maintenant de décrire l'ensemble des contraintes binaires définissant chacune des paires de régions adjacentes dont l'assignation des variables correspondantes doit différer. Les contraintes ci-dessous (↻) interdisent à deux régions voisines de posséder la même couleur. Elles sont représentées sous forme de contraintes binaires en intention avec le prédicat d'inégalité $\neq$:

2. Holi, aussi appelée *fête des couleurs*, est une fête hindoue célébrant le printemps.

$$\mathcal{C} = \left\{ \begin{array}{llllll} x_{ap} \neq x_{ka} & x_{ap} \neq x_{or} & x_{ap} \neq x_{py} & x_{ap} \neq x_{tn} & x_{ap} \neq x_{tn} & x_{ap} \neq x_{tg} \\ x_{ar} \neq x_{as} & x_{ar} \neq x_{nl} & x_{as} \neq x_{mn} & x_{as} \neq x_{ml} & x_{as} \neq x_{mz} & x_{as} \neq x_{nl} \\ x_{as} \neq x_{tr} & x_{as} \neq x_{wb} & x_{br} \neq x_{jh} & x_{br} \neq x_{up} & x_{br} \neq x_{wb} & x_{ch} \neq x_{hr} \\ x_{ch} \neq x_{pb} & x_{ct} \neq x_{jh} & x_{ct} \neq x_{mp} & x_{ct} \neq x_{mh} & x_{ct} \neq x_{or} & x_{ct} \neq x_{tg} \\ x_{ct} \neq x_{up} & x_{dn} \neq x_{gj} & x_{dn} \neq x_{mh} & x_{dd} \neq x_{gj} & x_{ga} \neq x_{ka} & x_{ga} \neq x_{mh} \\ x_{gj} \neq x_{mp} & x_{gj} \neq x_{mh} & x_{gj} \neq x_{rj} & x_{hr} \neq x_{hp} & x_{hr} \neq x_{dl} & x_{hr} \neq x_{pb} \\ x_{hr} \neq x_{rj} & x_{hr} \neq x_{up} & x_{hp} \neq x_{jk} & x_{hp} \neq x_{pb} & x_{hp} \neq x_{up} & x_{hp} \neq x_{ut} \\ x_{jk} \neq x_{pb} & x_{jh} \neq x_{or} & x_{jh} \neq x_{up} & x_{jh} \neq x_{wb} & x_{ka} \neq x_{kl} & x_{ka} \neq x_{mh} \\ x_{ka} \neq x_{tn} & x_{ka} \neq x_{tg} & x_{kl} \neq x_{py} & x_{kl} \neq x_{tn} & x_{mp} \neq x_{mh} & x_{mp} \neq x_{rj} \\ x_{mp} \neq x_{up} & x_{mh} \neq x_{tg} & x_{mn} \neq x_{mz} & x_{mn} \neq x_{nl} & x_{mz} \neq x_{tr} & x_{dl} \neq x_{up} \\ x_{or} \neq x_{tg} & x_{or} \neq x_{wb} & x_{py} \neq x_{tn} & x_{pb} \neq x_{rj} & x_{rj} \neq x_{up} & x_{sk} \neq x_{wb} \\ x_{up} \neq x_{ut} & & & & & \end{array} \right\}$$

Trois propositions d'instanciation sont proposées dans la figure 1.1 : deux cas conflictuels et une solution au problème. La figure 1.1b présente un conflit après l'assignation de trois variables correspondant à des régions adjacentes : deux d'entre elles sont assignées à la même valeur et violent ainsi la contrainte $x_{rj} \neq x_{up}$. Le second cas conflictuel de la figure 1.1c correspond à l'impossibilité d'assigner la variable x_{tg} car chacune des valeurs disponibles dans son domaine crée un conflit chez l'une des contraintes ayant x_{tg} dans sa portée. Autrement dit, la région correspondant à la variable x_{tg} est entourée de régions coloriées par les quatre couleurs disponibles. Le dernier cas de la figure 1.1d correspond à une solution de l'instance $\mathcal{P}$ où toutes les variables sont assignées tout en respectant les contraintes.

Pour trouver une solution à ce problème, nous avons eu besoin d'une recherche. Dans sa forme complète, la recherche effectue une exploration exhaustive de l'espace de recherche. L'espace de recherche correspond au produit cartésien des domaines des variables ; ici, comme nous avons 34 variables et 4 valeurs par domaine, nous obtenons un espace de recherche dont la taille est de 4^{34} ($> 10^{20}$). L'énumération de chaque instanciation complète à tour de rôle et la vérification de chacune d'entre elles pour déterminer si elle satisfait toutes les contraintes semblent être assez inefficaces ; c'est une méthode appelée *générer et tester*. Dans la partie suivante, il est montré comment certains algorithmes d'inférence et l'usage d'heuristiques permettent de réduire cette complexité en pratiquant une recherche moins naïve.

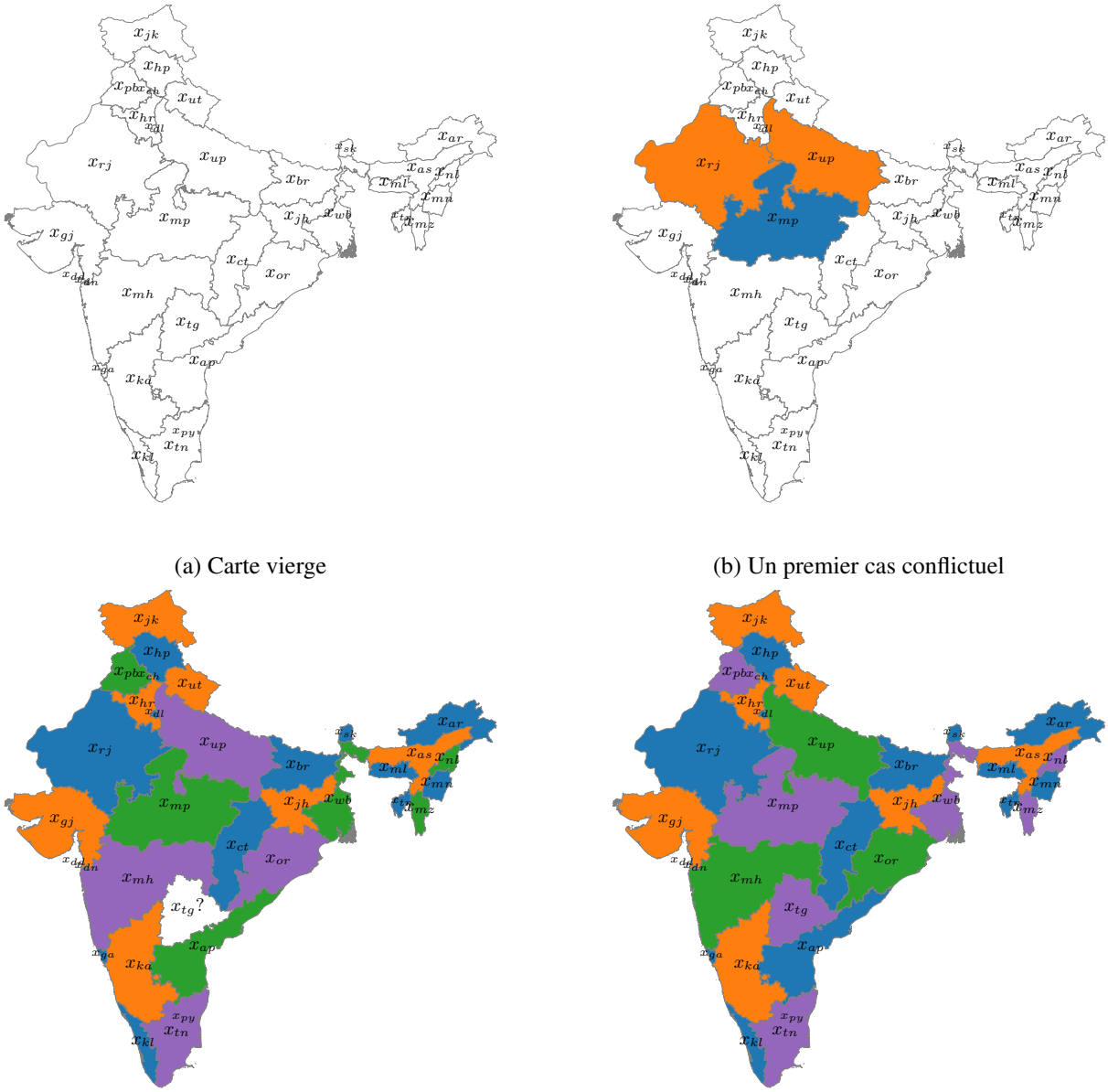
Avant de conclure cette partie, nous proposons une instances COP de ce problème sans la connaissance *a priori* que seules quatre couleurs sont nécessaires. Soit une instance COP $\mathcal{P}' = (\mathcal{X}', \mathcal{C}', \mathcal{O})$ composée d'un ensemble de variables semblable à la précédente instance $\mathcal{X}' = \mathcal{X}$ dont le domaine de chacune d'elles diffère en prenant autant de valeurs que de régions (variables) disponibles :

$$\forall x \in \mathcal{X}', \texttt{dom}(x) = \{0, 1, \ldots, n-1\}$$

où n est le nombre de variables. Dans le pire cas, et sans connaissance du nombre de couleurs nécessaires pour colorier cette carte, nous proposons autant de couleurs que de régions (c'est-à-dire, des domaines aussi larges que le nombre de variables disponibles). Dans le cas de cette instance COP, une fonction objectif est aussi définie pour minimiser le nombre de couleurs et ainsi permettre au solveur de trouver une solution équivalente à celle proposée dans la figure 1.1d :

$$\mathcal{O} = \texttt{minimize}(\texttt{maximum}(\mathcal{X}'))$$

La fonction objectif essaye de minimiser la plus grande valeur assignée aux variables du problème. La contrainte `maximum` peut, par exemple, être remplacée par la contrainte `card` (*cardinalité*) retournant le nombre de valeurs distinctes assignées aux variables $\mathcal{X}'$; cette valeur doit aussi être minimisée. `card` est plus générale et nécessaire si par exemple des contraintes unaires portent sur les variables. En définitif, la soumission de cette instance à un solveur COP (↻) montre que les domaines se réduisent progressivement à ceux de l'instance CSP précédente $\mathcal{P}$ et qu'une solution équivalente à celle proposée dans la figure 1.1d est ainsi trouvée.



(a) Carte vierge

(b) Un premier cas conflictuel

(c) Un second cas conflictuel

(d) Une solution au problème

FIGURE 1.1 – Problème de coloration de carte appliqué à la carte de l'Inde [↻]

1.3 Inférence et résolution

Comme mentionné dans la précédente section, la taille de l'espace de recherche d'une instance peut croître de manière exponentielle et il est alors compliqué de tester l'ensemble des instanciations complètes. C'est pourquoi, en programmation par contraintes, un mécanisme d'inférence (ou filtrage) est appliqué afin de se rendre compte le plus tôt possible qu'une instanciation partielle mène à une incohérence globale. Nous présentons ce principe permettant de filtrer le réseau de contraintes et évitant de parcourir des zones inutiles. Il est ensuite proposé un algorithme générique permettant de parcourir l'espace de recherche avec ce principe d'inférence et son implantation dans les solveurs modernes.

1.3.1 Cohérence et propagation

Il existe différents niveaux d'inférence proposés par différents opérateurs de *cohérences* ϕ. Ainsi, pour un réseau de contraintes donné, il est possible d'obtenir différents niveaux de cohérence permettant de filtrer plus ou moins largement l'espace de recherche. La complexité temporelle croît en fonction du niveau de cohérence appliqué. À travers la littérature, de nombreux opérateurs sont proposés [MACKWORTH 1977] dont l'arc-cohérence, la cohérence de chemin, la singleton arc-cohérence, *etc.* Dans cette section, nous nous concentrons sur l'opérateur le plus utilisé dans les solveurs actuels : la cohérence d'arc [MACKWORTH 1977].

Reprenant la notion de support (définition 12), la *cohérence d'arc* se définit comme suit :

Définition 23 (Contrainte arc-cohérente)

Une contrainte est dite *arc-cohérente* (AC) si et seulement si $\forall x \in scp(c), \forall a \in dom(x)$, il existe un support sur c pour (x, a), i.e., un support τ sur c tel que $\tau[x] = a$.

Définition 24 (Valeur arc-incohérente)

Une valeur (x, a) est *arc-incohérente* pour une contrainte c quand elle n'admet pas de support sur c.

Définition 25 (Algorithme AC)

Un *algorithme AC*, pour une contrainte c, est un algorithme qui supprime toutes les valeurs qui sont arc-incohérentes sur c ; on dit de cet algorithme qu'il force/établit l'arc-cohérence sur c.

Une fois la notion de cohérence d'arc définie pour les valeurs et les contraintes, son extension à l'ensemble des contraintes du réseau se définit comme suit.

Définition 26 (Réseau de contraintes AC)

Un *réseau de contraintes $\mathcal{P}$ est AC* si et seulement si chacune des contraintes de $\mathcal{P}$ est AC.

Définition 27 (AC-fermeture)

Calculer la *AC-fermeture* sur un réseau de contraintes $\mathcal{P}$ est le fait de supprimer toutes les valeurs arc-incohérentes de $\mathcal{P}$.

L'opérateur calculant la AC-fermeture d'un réseau de contraintes est appelé *opérateur de filtrage* et est noté ϕ. Dans un cadre général, de nombreuses implantations calculant l'AC-fermeture sont proposées dans l'état de l'art, essayant de minimiser la complexité temporelle, telles que AC3 [MACKWORTH 1977], AC2001 [BESSIÈRE *et al.* 2005], *etc.* Le réseau de contraintes arc-cohérent obtenu est le même peu importe l'algorithme sélectionné et l'ordre suivi pour l'obtenir (*unicité*).

1.3.2 Méthode de résolution

Pour un réseau de contraintes $\mathcal{P}$, la méthode *générer et tester* — générant l'ensemble des instanciations complètes et vérifiant linéairement chacune d'elles — revient à une complexité temporelle en $O(o^n er)$, où $O(x)$ correspond à une borne supérieure asymptotique en x. Une telle complexité n'est pas acceptable en pratique. Dans cette partie, nous voyons comment remédier à cela en appliquant la précédente méthode de propagation à travers un *algorithme de retour en arrière* (*backtracking algorithm*) permettant de parcourir l'espace de recherche tout en réduisant le coût temporel.

La recherche avec retour en arrière est une méthode de *recherche complète*. Elle consiste en un parcours en profondeur de l'arbre de recherche avec un mécanisme de retour en arrière et une succession de décisions et de propagations. Il existe aussi des méthodes de *recherche incomplète* n'assurant pas la complétude de l'algorithme mais une efficacité parfois accrue pour trouver rapidement des solutions.

Pour ce type d'algorithme complet, il existe plusieurs manières de parcourir l'arbre de recherche. L'une d'elles consiste en une recherche binaire. Chaque nœud de l'arbre est séparé en deux branches, couvrant deux décisions complémentaires. Ces décisions complémentaires peuvent correspondre à l'assignation d'une valeur à une variable et la réfutation de cette dernière en la supprimant du domaine de la variable, ou alors en scindant le domaine d'une variable. La recherche n'a pas pour obligation d'être binaire, d'autres algorithmes de retour en arrière emploient un branchage n-aire divisant chaque nœud de recherche (correspondant à la sélection d'une variable) en autant de branches que de valeurs disponibles dans le domaine de la variable sélectionnée.

En pratique, la méthode binaire avec assignation et réfutation est celle la plus utilisée dans les solveurs modernes. L'algorithme 1 présente cette dernière avec l'application de l'opérateur ϕ filtrant le réseau de contraintes à chaque appel.

Algorithme 1 : recherche-binaire-$\phi(\mathcal{P} = (\mathcal{X}, \mathcal{C})$: CN)

```
1 P ← φ(P)
2 si ∃x ∈ X, dom(x) = ∅ alors
3 |   retourner faux
4 fin
5 si ∀x ∈ X, |dom(x)| = 1 alors
6 |   retourner vrai
7 fin
8 sélection d'un couple variable/valeur (x, a) de P tel que |dom(x)| > 1
9 retourner recherche-binaire-φ(P|_{x=a}) ∨ recherche-binaire-φ(P|_{x≠a})
```

L'algorithme 1 prend en paramètre un réseau de contraintes $\mathcal{P}$. En premier lieu, l'algorithme filtre le réseau de contraintes en supprimant les valeurs incohérentes selon l'opérateur ϕ. La première condition (lignes 2 à 4) vérifie si le domaine de l'une des variables n'a pas été vidé par la précédente propagation. Si tel est le cas, le Booléen *faux* est retourné et témoigne d'une assignation partielle globalement incohérente. La seconde condition (lignes 5 à 7) vérifie si toutes les variables du réseau de contraintes sont fixées, c'est-à-dire, que la taille du domaine de chaque variable est égale à 1. Si tel est le cas, une solution vient d'être trouvée et le Booléen *vrai* est retourné. Enfin, les deux dernières lignes de l'algorithme montre l'opération de branchage. Dans un premier temps (ligne 8), l'algorithme sélectionne une variable non-fixée et une valeur de son domaine (x, a). Par la suite, nous verrons que ce processus est délégué à des heuristiques (section 1.4). Une fois sélectionnée, ce couple permet l'appel récursif de l'algorithme par la disjonction recherche-binaire-$\phi(\mathcal{P}|_{x=a}) \vee$ recherche-binaire-$\phi(\mathcal{P}|_{x\neq a})$, où $\mathcal{P}|_\delta$ correspond au réseau

de contraintes $\mathcal{P}$ restreint par la décision δ. La partie gauche correspond à l'application de l'assignation de x à a sur le réseau de contraintes $\mathcal{P}$. Si cet appel retourne *faux*, la partie droite de la disjonction est appelée et applique l'action complémentaire : la réfutation de a du domaine de x. Récursivement, l'algorithme de recherche binaire parcourt exhaustivement l'espace de recherche tout en évitant une récursivité trop importante (c'est-à-dire, un arbre de recherche trop profond) grâce à l'application de l'opérateur ϕ et la mise en évidence de solutions partielles globalement incohérentes.

De nouveau, il existe différentes implantations de l'algorithme 1 dépendant de l'opérateur de propagation. Par application de l'opérateur d'AC-fermeture, l'algorithme de recherche binaire correspondant s'appelle *Maintien de la Cohérence d'Arc* (MAC — *Maintaining Arc Consistency*) [SABIN & FREUDER 1994]. D'autres implantations existent telles que *Forward Checking* (FC) [HARALICK & ELLIOTT 1980], *Real Full Look-ahead* (RFL) [NADEL 1988], *etc.*

Avant de décrire l'exemple présenté dans la figure 1.2, nous proposons quelques métriques inspirées des définitions de l'étude [BESSIERE *et al.* 2004] et appliquées au parcours binaire tel que précédemment présenté. Soit un arbre $\mathcal{T}$ produit par un algorithme A, type recherche-binaire-ϕ, pour résoudre un CSP $\mathcal{P}$. La métrique la plus englobante que nous présentons correspond aux nombres de nœuds parcourus par l'algorithme.

Définition 28 (Nombre de nœuds)

Le *nombre de nœuds* va correspondre à la taille de l'arbre $\mathcal{T}$, c'est-à-dire, le nombre d'appel à A lors de la recherche binaire avec retour en arrière.

Il peut être aussi intéressant de comptabiliser le nombre de fois où l'algorithme retourne *faux* lors de la vérification des domaines vides.

Définition 29 (Nombre de conflits)

Le *nombre de conflits* correspond au nombre de nœuds pour lesquels la propagation a généré un domaine vide pour l'une des variables.

Une autre mesure utile pour ce type d'algorithme est de compter le nombre de retours en arrière.

Définition 30 (Nombre de retours en arrière)

Le *nombre de retours en arrière* correspond au nombre de nœuds menant à une instanciation globalement incohérente.

Le nombre de mauvaises décisions est une autre mesure intéressante.

Définition 31 (Nombre de mauvaises décisions)

Le *nombre de mauvaises décisions* compte le nombre de fois où une décision est prise au cours de la recherche, sauf si ce choix conduit finalement à une solution.

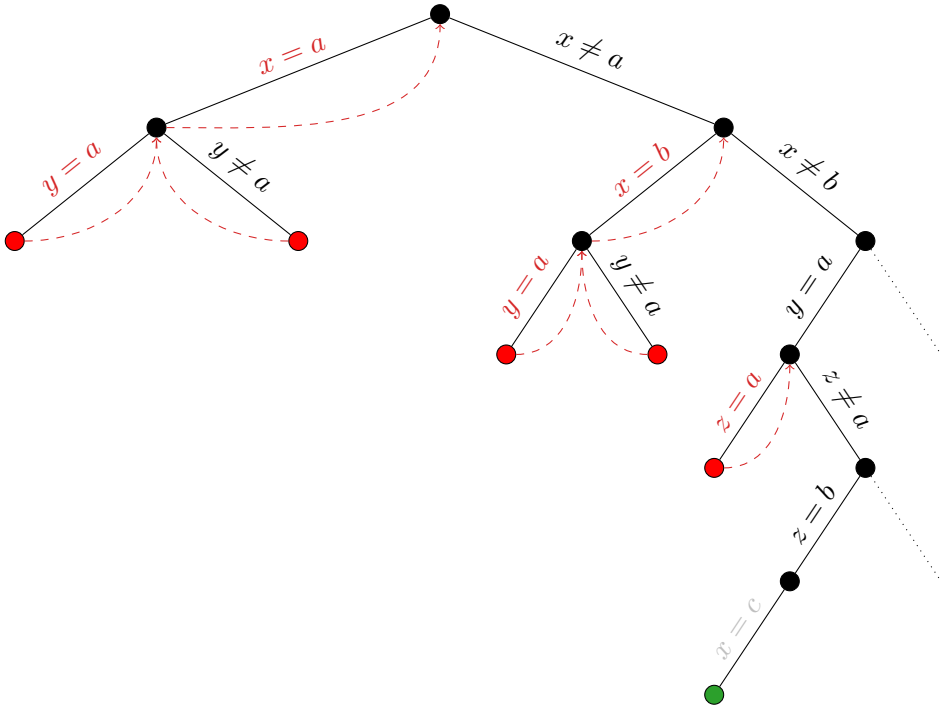


FIGURE 1.2 – Exemple d'exécution générique de recherche-binaire-$\phi(\mathcal{P})$

Exemple 8

Sur la figure 1.2, un arbre $\mathcal{T}$ produit par un algorithme A (type recherche-binaire-ϕ) est représenté, tentant de résoudre une instance CSP $\mathcal{P} = (\mathcal{X}, \mathcal{C})$ où $\mathcal{X} = \{x, y, z\}$ et $\texttt{dom}(x) = \texttt{dom}(y) = \texttt{dom}(z) = \{a, b, c\}$. A sélectionne séquentiellement les couples variable/valeur dans l'ordre dans lequel ils ont été définis (ordre *lexicographique*, voir section 1.4). Ainsi, nous observons que $\mathcal{T}$ est un arbre binaire où chaque nœud non-terminal (nœud noir) est composé de deux branches représentant l'assignation et la réfutation du couple de variable/valeur sélectionné.
Chaque nœud représente un appel à A :

- un nœud noir représente une exécution complète de A accompagnée du branchement (ligne 9 de l'algorithme 1) ;
- un nœud rouge représente un *conflit* ayant provoqué un *retour en arrière* jusqu'au nœud parent ;
- un nœud vert correspond à une solution.

Une assignation grisée correspond à l'assignation triviale d'une variable avec la valeur de son domaine singleton. Les autres assignations correspondent à des *décisions*. Enfin, les décisions en rouge sont de *mauvaises décisions*.
Pour résumer, l'arbre $\mathcal{T}$ est composé de 14 nœuds, 5 conflits, 7 retours en arrière et 7 décisions dont 5 mauvaises décisions.

1.3.3 Redémarrage

Les solveurs de contraintes modernes sont équipés d'une fonction de redémarrage de la recherche permettant de répondre au phénomène de queue longue (*heavy-tailed behavior*). Ce phénomène correspond à une perte importante de temps de recherche dans des impasses (notamment sur les instances aléatoires et les instances réelles [GOMES *et al.* 2000]). Les premières décisions sont primordiales pour obtenir des conflits le plus tôt possible afin d'éviter que de mauvaises décisions prises en début de recherche viennent agrandir inutilement l'arbre de recherche. Certaines heuristiques apprennent et réordonnent dynamiquement le choix des variables sur lesquelles branchées, c'est pourquoi il est notamment utile de reprendre régulièrement la recherche depuis la racine de l'arbre. Cette fonction correspond à l'application $\texttt{restart} : \mathbb{N}^+ \rightarrow \mathbb{N}^+$, où $\texttt{restart}(t)$ est le nombre maximal d'*étapes* pouvant être effectuées par l'algorithme de retour en arrière lors de la $t$ème exécution. Un solveur de contraintes, équipé de la procédure MAC et d'une stratégie de redémarrage `restart`, construit une séquence d'arbres de recherche binaires $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots \rangle$, où $\mathcal{T}_t$ est l'arbre de recherche exploré par MAC à l'exécution t.

Le *cutoff*, qui est le nombre d'étapes autorisées dans une exécution, peut être défini par le nombre de retours en arrière, le nombre de mauvaises décisions, ou toute autre mesure pertinente.

Dans une stratégie de redémarrage à *cutoff fixe*, le nombre T de *runs* est fixé à l'avance, et $\texttt{restart}(t)$ est constant pour chaque *run* t, sauf pour le Tème *run* qui autorise un nombre illimité d'étapes (afin de conserver un algorithme complet). Cette stratégie est connue pour être efficace en pratique [GOMES *et al.* 1998], mais une bonne valeur de *cutoff* $\texttt{restart}(t)$ doit être trouvée par essais et erreurs. Alternativement, dans une stratégie de redémarrage à *cutoff dynamique*, le nombre T de *runs* est inconnu, mais `restart` augmente le *cutoff* géométriquement [SÖRENSSON & EÉN 2005], ce qui garantit que l'espace entier des instanciations est exploré après un nombre polynomial de *runs*. Plusieurs fonctions de *cutoff* sont couramment utilisées où le nombre d'étapes par *run* est donné par $u \times l_t$, où u est une constante fixée par le solveur et l_t la suite utilisée.

$$l_t = \texttt{exp}(t) = 2^{t-1}$$

Cette première équation correspond à la classique fonction exponentielle `exp` (en base 2) garantissant que l'espace de recherche est entièrement exploré après $O(n)$ exécutions.

$$l_t = \texttt{rexp}(t) = \begin{cases} 2^{k-1}, & \text{if } t = \frac{k(k+1)}{2} \\ 2^{t-\frac{k(k+1)}{2}-1}, & \text{if } \frac{k(k+1)}{2} < t < \frac{(k+1)(k+2)}{2} \end{cases}$$

Dérivée de la fonction exponentielle, `rexp` correspond à une suite de `exp` régulièrement réinitialisée. Les premières valeurs de cette séquence sont : $1, 1, 2, 1, 2, 4, \ldots$. Pour cette suite, l'espace de recherche est garanti d'être exhaustivement exploré après $O(n^2)$ runs.

$$l_t = \texttt{luby}(t) = \begin{cases} 2^{k-1}, & \text{if } t = 2^k - 1 \\ \texttt{luby}(t - 2^{k-1} + 1), & \text{if } 2^{k-1} \leq t < 2^k - 1 \end{cases}$$

Une autre suite, couramment utilisée dans les politiques de redémarrage, est la suite de Luby [LUBY *et al.* 1993]. Les premières valeurs de la suite de `luby` sont : $1, 1, 2, 1, 1, 2, 4, \ldots$. Pour cette dernière suite, l'espace de recherche est garanti d'être exhaustivement exploré après $O(2^n)$ runs.

Il est important de noter que même si le solveur recommence depuis le début, il peut mémoriser certaines informations pertinentes sur la séquence $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{t-1} \rangle$ telles que des *nogoods* afin d'éviter un passage inutile dans la même portion de l'espace de recherche entre deux *runs*.

Afin d'exprimer plus précisément la création de ces *nogoods* de fin de *run*, nous reprenons la définition 20 donnant l'intuition de ce qu'est un *nogood* et définissons le principe de ces nogoods produits lors

de la fin d'un *run*, appelés *nld-nogoods* (*Negative Last Decision Nogoods*). Avant ceci, voyons ce que sont les *nld-sous-séquences* identifiant ces nogoods et dans une dernière définition, leur version *réduite*.

Définition 32 (nld-sous-séquence)

Soit $\Sigma = \langle \delta_1, \ldots, \delta_i, \ldots, \delta_m \rangle$ une séquence de décisions où δ_i est une décision négative. La séquence $\langle \delta_1, \ldots, \delta_i \rangle$ est appelée *nld-sous-séquence* de Σ. Les décisions précédant δ_i peuvent être positives ou négatives.

Les ensemble de décisions positives et négatives de Σ sont respectivement notés $\texttt{pos}(\Sigma)$ et $\texttt{neg}(\Sigma)$.

Définition 33 (nld-nogood)

Soit $\mathcal{P}$ un réseau de contraintes et Σ une séquence de décisions prises sur une branche de l'arbre de recherche menant à un sous-arbre incohérent. Pour toute nld-sous-séquence $\langle \delta_1, \ldots, \delta_i \rangle$ de Σ, l'ensemble $\Delta = \langle \delta_1, \ldots, \delta_{i-1}, \delta_i \rangle$ est un nogood de $\mathcal{P}$ appelé *nld-nogood* (à chaque décision négative de l'arbre de recherche, un *nld-nogood* peut être extrait).

Définition 34 (nld-nogood réduit [LECOUTRE *et al.* 2007])

Soit $\mathcal{P}$ un réseau de contraintes et Σ une séquence de décisions prises sur une branche de l'arbre de recherche. Pour toute nld-sous-séquence $\Sigma' = \langle \delta_1, \ldots, \delta_i \rangle$ de Σ, l'ensemble $\Delta = \texttt{pos}(\Sigma') \cup \{\delta_i\}$ est un nogood de $\mathcal{P}$ appelé *nld-nogood réduit*.

Par la suite, nous considérons que la mention de *nogood* correspond en fait à *nld-nogood* réduit.

Afin de ne pas pénaliser le solveur dans l'extraction de *nogoods*, nous remarquons que tous les *nogoods* extraits de la dernière branche d'un arbre de recherche subsument tous les *nogoods* extraits des branches précédemment explorées dans cet arbre.

Exemple 9

Afin d'illustrer l'extraction de *nld-nogoods* réduits au redémarrage de la recherche lors de la fin d'un *run*, nous reprenons l'exemple d'arbre de la figure 1.2, fixons un *cutoff* de 12 nœuds et obtenons la figure 1.3.
La dernière branche de cet arbre de recherche est $\Sigma = \{x \neq a, x \neq b, y = a, z \neq a\}$ où 3 *nld-nogoods* réduits sont extraits :

- $\Delta_1 = \{x = a\}$
- $\Delta_2 = \{x = b\}$
- $\Delta_3 = \{y = a, z = a\}$

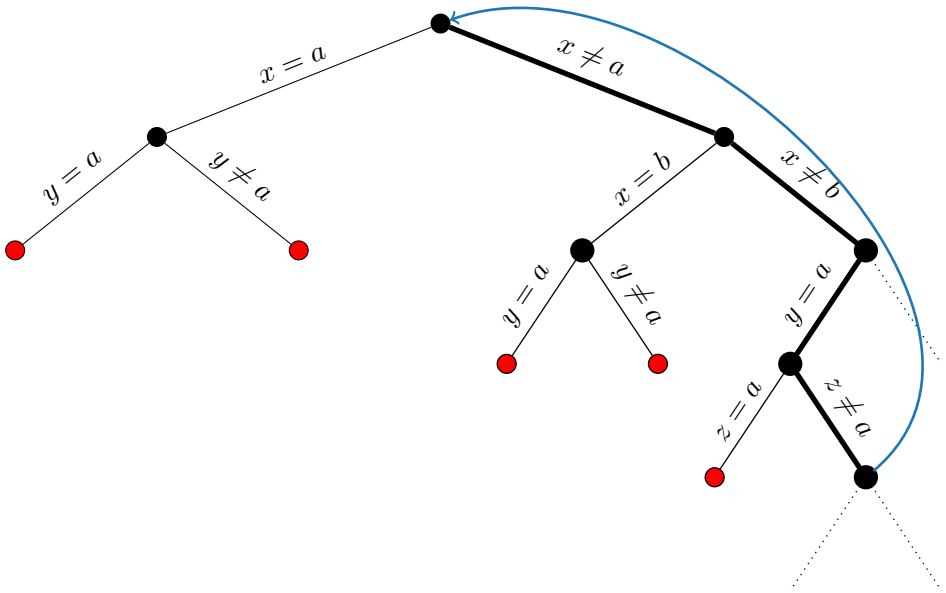


FIGURE 1.3 – Arbre illustrant le redémarrage de la recherche avec un *cutoff* fixé à 12 nœuds

1.4 Heuristiques de branchement

Dans cette quête visant à réduire la taille de l'arbre explorant l'espace de recherche, le mécanisme sélectionnant le prochain couple variable/valeur (x, a), grâce auquel brancher, est important. Pour cela, le précédent algorithme peut faire appel à des *heuristiques* sélectionnant ce couple. Quand trouver le parfait ordonnancement des variables et des valeurs est une tâche au moins aussi compliquée que de résoudre le CSP/COP lui-même [LIBERATORE 2000], les *heuristiques* sont une méthode de sélection sous-optimale permettant un choix approximatif, selon une philosophie qui lui est propre, et restant plus efficace que le choix aléatoire.

Dans notre cas, le but d'une heuristique est la minimisation de la taille de l'arbre de recherche. Il est important de savoir que l'ordre dans lequel les variables sont assignées est un élément clé. L'usage d'une heuristique plutôt qu'une autre peut mener à une différence drastique en terme d'efficacité et de robustesse de l'algorithme.

Pour chaque élément du couple (x, a) une heuristique spécifique est utilisée : une première heuristique est sollicitée afin d'ordonner les variables, et est associée à une seconde heuristique ordonnant les valeurs à sélectionner dans le domaine de la variable sélectionnée.

Une philosophie communément suivie lors de la création d'heuristiques pour ce type de recherche combinatoire est « Pour réussir, essaie en premier là où tu as le plus de chance de faillir. » [HARALICK & ELLIOTT 1980], aussi nommée le principe du *fail-first* [SMITH & GRANT 1998]. Cette citation est en lien direct avec le choix crucial des premières variables à assigner : il faut que celles-ci appartiennent à la partie compliquée du problème afin d'amener une détection rapide de l'incohérence (détectée par arc-cohérence) et le plus haut possible dans l'arbre de recherche — évitant ainsi un déploiement trop large de ce dernier. Plus en lien avec les heuristiques de choix de valeur et afin de trouver rapidement une solution, il est préférable d'affecter la première valeur permettant d'amener au plus vite à une solution. De manière globale, le choix minutieux du couple d'heuristiques sélectionnant le couple de variable et valeur (x, a) lors d'une recherche avec retour en arrière est particulièrement important quant à la réduction de l'arbre

de recherche.

Les deux sections suivantes décrivent les deux familles d'heuristiques présentes au sein des solveurs modernes.

1.4.1 Choix de variable

Dans l'exécution classique d'un solveur, l'heuristique de choix de variable est appelée en premier lieu pour choisir la prochaine variable sur laquelle brancher. Une heuristique associe à chacune des variables du problème un score calculé *statiquement*, *dynamiquement* ou de façon *adaptative* — représentant les trois grandes catégories d'heuristique de choix de variable. Selon la philosophie associée à cette heuristique et au score attribué à chacune des variables, celle-ci choisit la variable maximisant ou minimisant les scores tout en omettant les variables déjà fixées. À travers les trois prochaines sections, les trois catégories d'heuristiques de choix de variable sont décrites.

Heuristiques statiques

Les *heuristiques statiques* associent un score à chaque variable dès le début de la recherche. Ce score reste statique tout au long de la recherche. Ci-après, nous énumérons les heuristiques statiques les plus connues.

`rand` `rand` choisit aléatoirement la variable suivante sur laquelle brancher.

`lex` `lex` sélectionne la variable minimisant le rang de son emplacement lexicographique. Autrement dit, il s'agit de la sélection de la première variable libre en partant de l'ordre dans lequel les variables apparaissent dans la description de l'instance. Cette heuristique peut se montrer efficace si l'ordre proposé lors de la modélisation de l'instance est minutieusement choisi.

`deg` `deg` sélectionne la variable maximisant le score défini par son degré, c'est-à-dire, le nombre de contraintes où la variable est impliquée. Intuitivement, plus une variable est contrainte plus son assignation risque de provoquer un grand impact dans l'espace de recherche, voire de créer un conflit.

Heuristiques dynamiques

Les *heuristiques dynamiques* prennent en compte la dynamicité du réseau de contraintes. En effet, à chaque nœud de recherche, l'assignation de variables et le filtrage des domaines changent continuellement l'état du réseau de contraintes. Ainsi, à chaque état du réseau, il semble judicieux que l'heuristique reconsidère le score associé à chaque variable.

`dom` `dom` sélectionne la variable minimisant le score défini comme la taille de son domaine courant (existe aussi dans une version statique où le domaine initial est considéré). L'intuition derrière cette heuristique est que la falsification d'une valeur provenant d'un petit domaine a un plus grand impact sur la réduction de l'espace de recherche que sur un domaine plus grand.

`ddeg` `ddeg` sélectionne la variable maximisant le score égale à son degré dynamique, c'est-à-dire, le nombre de contraintes où la variable est impliquée et pour laquelle il existe une autre variable non-fixée.

Heuristiques adaptatives

Enfin, il existe les *heuristiques adaptatives* qui, en plus d'utiliser l'état courant des variables et du réseau de contraintes, prennent aussi en compte l'historique. Afin d'illustrer plus simplement ces heuristiques dont la complexité temporelle et spatiale augmente au même titre que leur robustesse, nous proposons une interface composée de plusieurs méthodes retraçant le cycle de vie de ces heuristiques au sein d'un solveur :

- INIT() : avant que ne démarre la phase d'exploration par le solveur (c'est-à-dire que le solveur n'entame sa série de *runs*), l'heuristique de choix de variable initialise ses structures par le biais de cette fonction ;
- AVANTRUN() : à chaque début de run, cette fonction permet la mise-à-jour des différents paramètres ou scores de l'heuristique ;
- APRÈSCONFLIT(c) : pendant le run courant, le solveur peut rencontrer des conflits (lorsque le domaine d'une variable est vidé lors de l'appel d'un algorithme de filtrage sur l'une des contraintes). À chaque conflit, la fonction APRÈSCONFLIT est appelée avec la contrainte c ayant causé le conflit ;
- APRÈSASSIGNATION($\mathcal{P}_{av}$, $\mathcal{P}_{ap}$) : après chaque assignation produite par le solveur, cette fonction est appelée avec l'*état du réseau de contraintes* $\mathcal{P}_{av}$ avant l'assignation et l'état du réseau de contraintes après assignation et propagation $\mathcal{P}_{ap}$. Plus précisément, au fur et à mesure que la recherche progresse (suppression de valeurs des domaines, assignation, propagation, *etc.*) le réseau s'en retrouve modifié, c'est pourquoi nous parlons d'état du réseau de contraintes ;
- SCORE(x) : cette fonction est appelée (sur toutes les variables non-fixées) lors de la sélection de la prochaine variable à assigner par l'heuristique. D'un point de vue global, le solveur cherche à identifier la variable dont le score est maximal (ou minimal) : $\max_{x \in \mathcal{X}} \text{SCORE}(x)$.

L'interface maintenant définie, la description de chacune des heuristiques adaptatives peut être donnée. Si nous considérons cette interface pour les précédentes heuristiques, seule la méthode SCORE est instanciée donnant directement l'information statique ou dynamique.

Heuristique I : `abs`($\mathcal{P} = (\mathcal{X}, \mathcal{C})$: CN)

Méthode INIT() :
  pour chaque $x \in \mathcal{X}$ **faire**
    $w_x \leftarrow 0$
  fin
  $\alpha \leftarrow 0.999$

Méthode AVANTRUN() :
  Aucune opération

Méthode APRÈSCONFLIT(c) :
  Aucune opération

Méthode APRÈSASSIGNATION($\mathcal{P}_{av}$, $\mathcal{P}_{ap}$) :
  pour chaque $x \in \texttt{fut}(\texttt{vars}(\mathcal{P}_{av}))$ **faire**
    $r \leftarrow \mathbb{1}_{\texttt{dom}(x_{\mathcal{P}_{ap}}) \subsetneq \texttt{dom}(x_{\mathcal{P}_{av}})}$
    $w_x \leftarrow w_x \times \alpha + r$;
  fin

Méthode SCORE(x) :
  retourner w_x

Les deux premières heuristiques, `abs` [MICHEL & HENTENRYCK 2012] et `ibs` [REFALO 2004], se

focalisent essentiellement sur le changement d'état du réseau de contraintes avant et après l'assignation (et la propagation) de variable pour mettre à jour et pondérer leurs variables. Les deux heuristiques suivantes ($\texttt{wdeg}^{\texttt{unit-2004}}$ et $\texttt{wdeg}^{\texttt{chs}}$) se concentrent sur les conflits occasionnés lors de la recherche pour mettre à jour la pondération des variables par le biais de la pondération des contraintes. Dans le but d'uniformiser les notations à travers l'ensemble des heuristiques adaptatives présentées, certaines descriptions diffèrent de leurs explications originelles mais conservent le comportement attendu.

`abs` et `ibs` Ces deux premières heuristiques basent la mise-à-jour progressive de la pondération des variables grâce à l'étude de l'assignation et de la propagation de cette action à travers le réseau de contraintes.

La première heuristique présentée par l'heuristique I, `abs` (*Activity-Based Search*) [MICHEL & HENTENRYCK 2012], nécessite l'initialisation des variables w_x à 0 pour chacune des variables du réseau de contraintes et d'un paramètre α permettant l'affaiblissement de ces poids au fur et à mesure de la recherche. α est empiriquement initialisée à 0.999 selon l'étude [MICHEL & HENTENRYCK 2012]. La méthode APRÈSASSIGNATION est instanciée afin de récupérer l'information nécessaire pour mettre à jour la pondération des variables de l'heuristique : à chaque nouvelle assignation, chaque future variable donnée par la fonction $\texttt{fut}(\mathcal{X})$ est mise à jour avec une récompense r égale à 1 si son domaine a changé après propagation de l'assignation, 0 sinon. Le poids de cette variable est mise à jour en affaiblissant le précédent poids par α et en ajoutant la récompense r. La méthode SCORE retourne simplement le poids w_x de la variable x.

Dans l'article [MICHEL & HENTENRYCK 2012], les poids w_x ne sont pas initialisés à 0 mais à une valeur calculée lors de *runs* préliminaires permettant de sonder le réseau de contraintes. Ce comportement est détaillé dans la section suivante. Cette heuristique peut être utilisée conjointement avec une heuristique de choix de valeur de même nature que nous décrivons plus bas.

Heuristique II : $\texttt{ibs}(\mathcal{P} = (\mathcal{X}, \mathcal{C}) : \text{CN})$

Méthode `INIT()` **:**
  pour chaque $x \in \mathcal{X}$ **faire**
    $w_x \leftarrow 0$
  fin
  $\alpha \leftarrow 7/8$

Méthode `AVANTRUN()` **:**
  Aucune opération

Méthode `APRÈSCONFLIT(`c`)` **:**
  Aucune opération

Méthode `APRÈSASSIGNATION(`$\mathcal{P}_{av}, \mathcal{P}_{ap}$`)` **:**
  $r \leftarrow 1 - \frac{\texttt{size}(\mathcal{P}_{ap})}{\texttt{size}(\mathcal{P}_{av})}$
  $w_x \leftarrow w_x \times \alpha + r \times (1 - \alpha)$

Méthode `SCORE(`x`)` **:**
  retourner w_x

`ibs` (*Impact-Based Search*) [REFALO 2004] telle qu'implantée dans [MICHEL & HENTEN-

RYCK 2012, PERRON 2010] possède une structure semblable à `abs` avec un tableau conservant la pondération de chacune de ses variables et un paramètre α permettant l'affaiblissement de cette pondération. La méthode APRÈSASSIGNATION est instanciée et présente comment `ibs` met à jour la pondération de ses variables : après chaque assignation et propagation de cette action, une récompense r est composée et attribuée à x — la variable assignée. r correspond à l'impact qu'a eu l'assignation sur la réduction de la taille du réseau de contraintes : il s'agit du complément à 1 du ratio entre la taille actuelle du réseau $\texttt{size}(\mathcal{P}_{ap})$ et la précédente taille du réseau $\texttt{size}(\mathcal{P}_{av})$. Le poids de x est actualisée en affaiblissant sa pondération actuelle par α et en ajoutant la récompense avec un poids de $1-\alpha$.

Au même titre qu'`abs`, `ibs` propose une initialisation des poids w_x par un sondage du réseau de contraintes, et fonctionne aussi conjointement avec une heuristique de choix de valeur que nous décrivons plus bas.

$\texttt{wdeg}^{\texttt{unit-2004}}$ et $\texttt{wdeg}^{\texttt{chs}}$ Chacune de ces heuristiques est basée sur une forme de pondération, et est donc notée `wdeg` (*Weighted DEGree*) en suivant l'appellation choisie initialement [BOUSSEMART *et al.* 2004]. Les différentes variantes de cette famille d'heuristiques sont alors identifiées par un terme placé en exposant, comme par exemple dans $\texttt{wdeg}^{\texttt{chs}}$. Il est également possible d'utiliser `dom/wdeg` à la place de `wdeg` comme base de calcul ; cela sera défini un peu plus loin.

$\texttt{wdeg}^{\texttt{unit-2004}}$ est l'heuristique originelle de la famille `wdeg`. Comme structure de données, cette heuristique n'a besoin que d'une variable w_c par contrainte c pour enregistrer la pondération de celle-ci. À chaque conflit issu d'une contrainte c (lors du processus de filtrage par celle-ci), cette variable, qui sert simplement de compteur, est incrémentée de 1 (*unit* dans l'intitulé faisant référence à une incrémentation unitaire). Ainsi, le score d'une variable x, selon $\texttt{wdeg}^{\texttt{unit-2004}}$, correspond à la somme du poids des contraintes ayant strictement plus d'une variable non-fixée.

Heuristique III : $\texttt{wdeg}^{\texttt{unit-2004}}(\mathcal{P}=(\mathcal{X},\mathcal{C}) : \text{CN})$

Méthode `INIT()` **:**
  pour chaque $c \in \mathcal{C}$ **faire**
    $w_c \leftarrow 0$
  fin

Méthode `AVANTRUN()` **:**
  Aucune opération

Méthode `APRÈSCONFLIT(`c`)` **:**
  $r \leftarrow 1$
  $w_c \leftarrow w_c + r$

Méthode `APRÈSASSIGNATION(`$\mathcal{P}_{av}$, $\mathcal{P}_{ap}$`)` **:**
  Aucune opération

Méthode `SCORE(`x`)` **:**
  retourner $\sum_{c \in \mathcal{C} : x \in \texttt{scp}(c) \wedge |\texttt{fut}(c)|>1} w_c$

Il est à noter que pour la neutralité des expériences de ce manuscrit, les variables w_c des heuristiques $\texttt{wdeg}^{\texttt{unit-2004}}$ et $\texttt{wdeg}^{\texttt{chs}}$ sont initialisées à 0 ; respectivement, dans leurs implantations d'origine, ces heuristiques initient ces variables à 1 et 10^{-4}.

En plus des variables w_c, comme pour $\texttt{wdeg}^{\texttt{unit-2004}}$, une variable t_c pour chaque contrainte c et correspondant au temps du dernier conflit impliquant la contrainte c est introduite pour l'heuristique $\texttt{wdeg}^{\texttt{chs}}$ [HABET & TERRIOUX 2021] : le *temps* correspond ici à un compteur de conflits et non à une horloge effective. Par la suite, `time` est introduit et correspond au temps du dernier conflit du solveur : $\texttt{time} = \max_{c \in \mathcal{C}} t_c$. Le tableau t sert par la suite à différencier les contraintes entrant souvent en conflit. La fonction AVANTRUN est utilisée dans l'implantation de $\texttt{wdeg}^{\texttt{chs}}$ afin d'affaiblir la pondération des contraintes en fonction du temps depuis lequel elle n'est pas entrée en conflit : plus ce temps est important, plus l'affaiblissement du score l'est aussi. Une variable α initialisée à $1/10$ et évoluant au cours des conflits est introduite afin de donner plus ou moins d'importance au score donné à une variable lors d'un conflit. La fonction APRÈSCONFLIT reprend ce principe de temps depuis lequel la contrainte a subi son dernier conflit pour calculer l'incrément : $r = \frac{1}{\texttt{time} - t_c + 1}$. La paramètre α est mis à jour à chaque conflit : $\alpha \leftarrow \max(6/100, \alpha - 10^{-6})$. Plus le nombre de conflits est important, plus α s'affaiblit et plus l'historique de la pondération de la contrainte c sera conservé : $w_c = (1 - \alpha) \times w_c + \alpha \times r$. Autrement dit, dans les premiers conflits du *run*, les incréments ont un impact plus grand sur la pondération des contraintes. Enfin, t_c est mis à jour en lui attribuant la valeur du dernier conflit. Les constantes de cette heuristique ont été calculées empiriquement dans l'étude [CHERIF *et al.* 2020b] se basant sur des travaux d'une heuristique de branchement pour les solveurs SAT [LIANG *et al.* 2016].

Heuristique IV : $\texttt{wdeg}^{\texttt{chs}}(\mathcal{P} = (\mathcal{X}, \mathcal{C}) : \text{CN})$

Méthode `INIT()` **:**
  $\texttt{time} \leftarrow 0$
  pour chaque $c \in \mathcal{C}$ **faire**
    $w_c \leftarrow 0$
    $t_c \leftarrow 0$
  fin

Méthode `AVANTRUN()` **:**
  pour chaque $c \in \mathcal{C}$ **faire**
    $w_c \leftarrow w_c \times 0.995^{\texttt{time} - t_c}$
  fin
  $\alpha \leftarrow 1/10$

Méthode `APRÈSCONFLIT(`c`)` **:**
  $r \leftarrow \frac{1}{\texttt{time} - t_c + 1}$
  $\alpha \leftarrow \max(6/100, \alpha - 10^{-6})$
  $w_c \leftarrow (1 - \alpha) \times w_c + \alpha \times r$
  $\texttt{time} \leftarrow \texttt{time} + 1$
  $t_c \leftarrow \texttt{time}$

Méthode `APRÈSASSIGNATION(`$\mathcal{P}_{av}$, $\mathcal{P}_{ap}$`)` **:**
  Aucune opération

Méthode `SCORE(`x`)` **:**
  retourner $\sum_{c \in : x \in \texttt{scp}(c) \wedge |\texttt{fut}(c)| > 1} w_c$

Agrégation et *Tie-break*

En cas de meilleurs scores identiques, il existe des mécanismes subsidiaires permettant de discriminer plus finement les variables. Une heuristique est notée h. Un premier mécanisme, communément utilisé, correspond à l'agrégation d'heuristiques. Une agrégation couramment utilisée est, par exemple, `dom/ddeg` [BESSIÈRE & RÉGIN 1996], choisissant la variable minimisant le score d'une variable associé au ratio de la taille du domaine courant divisé par le degré dynamique de cette variable, ou encore `dom/wdeg` étant la meilleure variante proposée dans [BOUSSEMART *et al.* 2004]. Les deux précédents exemples utilisent l'opérateur de division, dont la forme plus générale est $h_1 \circ h_2$, mais il est aussi possible d'additionner, soustraire et multiplier deux ou plusieurs heuristiques entre elles, formant ainsi une nouvelle heuristique.

Outre l'agrégation pour affiner le score d'une variable, il est aussi possible de demander à une heuristique indépendante h_2 de discriminer des variables aux scores identiques d'une heuristique h_1 — nous parlons dans ce cas de *tie-break*. Cette heuristique peut être arbitraire ; basée sur un générateur aléatoire (`rand`), sur l'ordre lexicographique du nom des variables (`lex`), ou encore `deg`. Plus généralement, m heuristiques peuvent être chaînées en cas d'égalités consécutives et sont décrites comme suit : $h_1 > \cdots > h_m$ où h_i délègue le choix entre les variables de l'ensemble X_i à h_{i+1} en cas d'égalité.

Ci-après, nous décrivons quelques mécanismes supplémentaires pouvant servir d'agrégation ou de *tie-break*.

$\texttt{abs}_{\texttt{prob}}$ Par défaut en association avec l'heuristique `abs`, $\texttt{abs}_{\texttt{prob}}$ (*Activity-Based Search Probing*) est un mécanisme exécutant des *runs* préliminaires parcourant aléatoirement l'espace de recherche en sélectionnant uniformément les couples de variable/valeur et appliquant le principe de mise-à-jour de l'heuristique `abs` après chaque assignation dans le but d'obtenir une première pondération des variables. L'heuristique complète, telle que décrite dans [MICHEL & HENTENRYCK 2012], se nomme ici $\texttt{abs}_{\texttt{prob}} + \texttt{abs/dom}$.

$\texttt{ibs}_{\texttt{prob}}$ Dans cette même idée, $\texttt{ibs}_{\texttt{prob}}$ (*Impact-Based Search Probing*) propose un mécanisme pondérant chacune de ses variables en testant, de façon indépendante, l'impact qu'a chacune des assignations (puis propagation) de l'ensemble des paires variable/valeur du réseau de contraintes — soit $O(nd)$ opérations. L'heuristique de choix de variable présentée dans [REFALO 2004] se nomme ici $\texttt{ibs}_{\texttt{prob}} + \texttt{ibs}$.

`lc` `lc` (*last-conflict*) [LECOUTRE *et al.* 2006] est une manière paresseuse de simuler des retours en arrière intelligents, en retenant les k derniers conflits produits par le solveur. Ainsi, un poids de k est donné à la dernière variable entrée en conflit, puis $k-1$ pour l'avant-dernière, ..., et un poids égale à 0 pour les variables n'étant pas concernées par les k derniers conflits. Cette heuristique complémentaire, utilisée conjointement avec une autre, sélectionne la variable maximisant ses scores et qui, en cas d'égalité (pour les variables pondérées à 0), délègue à l'heuristique associée correspondant au schéma du *tie-breaking* : $h_1 > h_2$ où h_1 correspond à `lc`.

1.4.2 Choix de valeur

La description des heuristiques de choix de valeur est plus sommaire car il s'avère que l'une des heuristiques les plus basiques de cette famille semble le mieux fonctionner en pratique : par exemple, l'heuristique sélectionnant la première valeur du domaine `min-dom`. Une raison pour laquelle ce type d'heuristiques de choix de valeur rend le solveur plus robuste est en lien avec leur staticité. Ainsi, l'heuristique de choix de variables se base sur la staticité du choix de la valeur des variables afin d'être

efficacement adaptative dans l'ordonnancement de ses variables. Plus basiquement, il existe aussi une heuristique choisissant aléatoirement la valeur à affecter à une variable donnée : `randval`.

En association avec les heuristiques de choix de variable `ibs` et `abs`, celles-ci peuvent étendre leur architecture afin de mesurer plus finement l'activité et l'impact non seulement à l'échelle de la variable, mais à l'échelle de l'ensemble des couples variable/valeur du réseau de contraintes. Ainsi, quatre nouvelles heuristiques de choix de valeur apparaissent et sont nommées : `min-impact`, `max-impact`, `min-activity` et `max-activity`. Pour ces précédentes heuristiques, `min-*` permettent de se rapprocher du principe de *valeur prometteuse* et `max-*` du principe de *fail-first*.

1.5 Conclusion

Dans ce premier chapitre, nous avons introduit le concept de problème de satisfaction de contraintes, le réseau de contraintes sous-jacent et l'ensemble de ses composants. En extension de ce problème, nous avons aussi introduit le problème d'optimisation sous contraintes. La résolution naïve de ce type de problème étant exclue, les solveurs modernes embarquent des mécanismes d'inférence, heuristiques et de redémarrage permettant une recherche bien plus efficace en pratique.

Malgré l'ambition d'alléger autant que possible la charge cognitive de l'utilisateur optant pour la programmation par contraintes, la réalité est différente. Les solveurs de contraintes modernes étant équipés de divers composants, le réglage minutieux peut augmenter considérablement leur efficacité. Parmi ces composants, l'heuristique de choix de variable occupe une place centrale dans la recherche avec retour en arrière en ordonnant les variables à visiter dans l'arbre de recherche. Le choix de la bonne heuristique, pour un réseau de contraintes donné, peut affecter considérablement le temps de résolution, puisque différentes heuristiques peuvent conduire à des arbres de recherche entièrement différents. Dans ce but, plusieurs heuristiques ont été proposées et analysées pour ordonnancer les variables. Par conséquent, la question qui se pose est la suivante : étant donné une instance de CSP et un ensemble d'heuristiques de choix de variable disponibles dans le solveur, quelle heuristique est la meilleure pour résoudre l'instance ? Comme nous le verrons dans le chapitre 3 aucune heuristique existante ne s'avère optimale pour toutes les instances de CSP possibles. Trouver la bonne heuristique pour la tâche à accomplir n'est pas un exercice simple, sauf peut-être pour un expert.

Chapitre 2

Problème du Bandit Multi-Bras

Sommaire

2.1 Introduction et généralité

Dans ce chapitre, nous nous intéressons à l'*apprentissage par renforcement*. Il s'agit de l'un des trois paradigmes de base de l'*apprentissage automatique*, avec l'*apprentissage supervisé* et l'*apprentissage non supervisé*. Ce domaine s'intéresse à la manière dont un *agent*, dit autonome, apprend de ses *actions* grâce à des expériences menées dans son *environnement*. À chaque expérience, l'agent choisit une action en fonction de son état et l'environnement lui attribue une récompense quantitative (positive ou négative). Ainsi, à travers les expériences et les récompenses associées, l'agent cherche à optimiser sa politique en maximisant la somme de ses récompenses.

Plus précisément, nous étudions dans ce chapitre le cas où l'agent se retrouve dans un seul et même état où il doit déterminer l'action qui lui est la plus profitable : le *problème du bandit multi-bras* [GITTINS 1989, LATTIMORE & SZEPESVÁRI 2020] (MAB — *Multi-Armed Bandit*) illustre ce dilemme. Cet intitulé est inspiré du bandit manchot, une autre dénomination des machines à sous. Dans le cas où un joueur (agent) fait face à un ensemble de ces bandits manchots (actions ou bras), dont la probabilité de gain est fixe mais inconnue du joueur, celui-ci fait face au dilemme exploration-exploitation de ces machines afin de maximiser son gain sur une séquence de jeux. Il s'agit de rechercher un équilibre

entre l'exploration de l'environnement – afin de trouver des actions profitables, tout en exploitant le plus souvent possible la, supposée, meilleure action.

Plus formellement, le problème général du MAB peut se définir comme un jeu entre un *joueur* et l'environnement considéré comme son *adversaire*. L'algorithme 2 décrit une abstraction de politique répondant au problème du bandit. À chaque tour $t \in [T] = \{1, ..., T\}$ où T correspond à l'*horizon* (c'est-à-dire, le nombre total de tours), le joueur choisit une action $i_t \in [K] = \{1, ..., K\}$ où K est le nombre d'actions disponibles (ligne 2). Simultanément, l'environnement choisit une fonction de récompense $\mathcal{R}_t : [T] \to \mathbb{R}$ (ligne 3) depuis $\mathcal{R}$, l'ensemble des fonctions de récompense. Ensuite, le joueur reçoit le *feedback* $\mathcal{R}_t(i_t)$ (ligne 4) pouvant être une récompense ou une punition. Par la suite, nous utilisons le terme récompense concernant le *feedback* du joueur. Il s'agit d'un jeu à somme nulle où le joueur essaye de maximiser la somme des récompenses $\mathcal{R}_t(i_t)$ par le choix de ses actions i_t à chaque tour t et où l'adversaire tente de minimiser la somme des punitions $-\mathcal{R}_t(i_t)$ par le choix d'une fonction de récompense. Il est à préciser que pour le choix d'un même couple action et fonction de récompense lors de deux tours distincts, la récompense n'est pas nécessairement la même : le tirage des récompenses se fait depuis une variable aléatoire suivant une distribution bien précise mais inconnue du joueur.

Algorithme 2 : abstraction-bandit(K : actions)

```
1 pour chaque t allant de 1 à T faire
2 |   Choix de i_t ∈ [K]
3 |   L'environnement choisit R_t ∈ R
4 |   Observation du feedback R_t(i_t)
5 fin
```

À travers cette introduction, nous formalisons ce qu'est le problème du bandit multi-bras. Par la suite, nous proposons différentes politiques $\mathcal{B}$ répondant à ce dilemme exploration-exploitation et introduisons le principe de *regret*. Puis, nous présentons deux applications de ces politiques sur des problèmes réels et académiques. Enfin, nous présentons des paradigmes dérivés de ce principe d'exploration-exploitation.

2.2 Politiques d'exploration-exploitation

Dans cette partie, nous introduisons le principe de *regret* afin de quantifier l'efficacité d'une politique de bandit [3]. Ensuite, nous proposons différentes politiques répondant au dilemme d'exploration-exploitation du problème de bandit multi-bras. Afin de faciliter l'expression des algorithmes que nous présentons dans les prochains chapitres, une interface est proposée pour décrire ces politiques :

— INIT$_{\mathcal{B}}$() : méthode initialisant la structure du bandit $\mathcal{B}$;
— SELECT$_{\mathcal{B}}$(t) : retourne l'action sélectionnée pour le tour t du bandit $\mathcal{B}$;
— MAJ$_{\mathcal{B}}$($\mathcal{R}_t$) : mise à jour du bandit $\mathcal{B}$ avec la fonction de récompense associée au tour t.

Cette interface présente les bandits avec leurs notations mathématiques usuelles. En pratique, l'implantation de ces algorithmes est simplifiable. Ci-après, nous présentons les principales notations et leur description.

3. Par abus de langage, nous appelons aussi *bandit* une politique répondant au problème du bandit multi-bras.

Notation 2

- K est le nombre total de bras/actions disponibles ;
- T est le nombre total de tours (horizon) ;
- $\mathcal{R}$ est l'ensemble des fonctions de récompense ;
- $\mathcal{R}_t(i)$ est la récompense du choix i avec la fonction de récompense $\mathcal{R}_t$ au tour t ;
- $n_t(i)$ est le nombre de fois où l'action i est sélectionnée durant les t premiers tours ;
- $\hat{\mathcal{R}}_t(i)$ est la moyenne empirique des récompenses de l'action i au tour t sur ses $n_t(i)$ sélections.

Les différentes politiques présentées par la suite se classent en deux groupes dépendant de l'environnement dans lequel évolue le bandit : *stochastique* ou *adversarial*.

2.2.1 Mesure du *regret*

Soit la politique avec la plus grande récompense espérée associée au bras i^*, appelée politique optimale. Si le joueur connaissait la politique optimale à l'avance, alors à chaque tour, il tirerait simplement l'action associée à la politique optimale et recevrait la plus grande récompense possible. L'apprenant doit conjecturer le meilleur bras à partir du feedback du bandit, c'est-à-dire les récompenses des actions précédemment choisies. En d'autres termes, l'objectif du joueur est de trouver une politique qui soit la plus proche de la politique optimale. Chaque fois que le joueur choisit un bras sous-optimal, il perd la différence de récompense entre ces deux bras. Puisque le but du joueur est de maximiser la récompense des actions choisies, il s'ensuit que l'apprenant devrait essayer de minimiser cette différence, c'est-à-dire le *regret*.

Dans ce cadre, la phase d'exploration et la phase d'exploitation s'entrelacent. La tâche du joueur est de trouver une politique qui sélectionne un bras à chaque tour de façon à ce que le *regret cumulé* soit minimisé.

Définition 35 (Regret [BUBECK & CESA-BIANCHI 2012])

Le *regret* est défini comme la différence entre l'espérance des meilleures récompenses récompenses cumulées à chaque tour et les récompense cumulées de la politique du joueur :

$$\begin{aligned}
\texttt{regret}_T &= \mathbb{E}\left[\max_{i\in[K]} \sum_{t=1}^{T} \mathcal{R}_t(i) - \sum_{t=1}^{T} \mathcal{R}_t(i_t)\right] \\
&= \mathbb{E}\left[\max_{i\in[K]} \sum_{t=1}^{T} \mathcal{R}_t(i)\right] - \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{R}_t(i_t)\right] \\
&= \mathbb{E}\left[\max_{i\in[K]} \sum_{t=1}^{T} \mathcal{R}_t(i)\right] - \sum_{t=1}^{T} \mu_{i_t}
\end{aligned}$$

Définition 36 (Pseudo-regret [BUBECK & CESA-BIANCHI 2012])

Le *pseudo-regret* est défini comme la différence entre l'espérance de la récompense cumulée de la politique optimale et l'espérance de la récompense cumulée de la politique du joueur :

$$\begin{aligned}\texttt{pseudo-regret}_T &= \max_{i\in[K]} \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{R}_t(i) - \sum_{t=1}^{T} \mathcal{R}_t(i_t)\right] \\ &= \max_{i\in[K]} \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{R}_t(i)\right] - \mathbb{E}\left[\sum_{t=1}^{T} \mathcal{R}_t(i_t)\right] \\ &= T\mu_{i^*} - \sum_{t=1}^{T} \mu_{i_t}\end{aligned}$$

Dans les deux définitions, l'espérance est nécessaire par rapport au tirage aléatoire des récompenses pour les actions du joueur. Notez que le *pseudo-regret* est une notion plus faible du regret, puisqu'on se compare à l'espérance de l'action optimale i^*. Le *regret*, au contraire, est l'espérance du regret par rapport à l'action qui est optimale à chaque tour de la séquence de réalisation des récompenses. Plus formellement, nous avons $\texttt{pseudo-regret}_T \leq \texttt{regret}_T$.

Ainsi, tout en explorant les bras, le joueur doit également exploiter ses connaissances sur les récompenses des bras, car les récompenses des bras sélectionnés sont prises en compte pour le calcul du (pseudo-)regret. Il s'agit ici du dilemme classique entre exploration et exploitation dans l'apprentissage par renforcement.

2.2.2 Environnement stochastique

L'élément essentiel d'un environnement dit *stochastique* est la présence de probabilité de récompense stationnaire associée à chaque bras. Il existe K distributions de probabilité (arbitraires) $\nu_1, \ldots, \nu_K$ dans $[0, 1]$ associées respectivement aux bras $1, \ldots, K$. Soient $\mu_1, \ldots, \mu_K$ sont les espérances mathématiques respectives de $\nu_1, \ldots, \nu_K$: $\mathbb{E}(\nu_1), \ldots, \mathbb{E}(\nu_K)$. Lorsqu'une action i_t est sélectionné au tour t, sa récompense $\mathcal{R}_t(i_t)$ est tirée de la distribution correspondante ν_a ; pour toute autre action $j \neq i_t$, $\mathcal{R}_t(j)$ retourne 0. Dans un environnement stochastique, $\mathcal{R}$ correspond donc à un singleton car la distribution des récompenses pour chaque action est fixe.

Bandit A : ε-`Greedy`(K : actions)

Méthode `INIT()` **:**
  pour chaque *action* $i \in [K]$ **faire**
    $n_1(i) \leftarrow 0$
    $\hat{\mathcal{R}}_1(i) \leftarrow 0$
  fin

Méthode `SELECT`(t) **:**
  Choix du paramètre $\varepsilon_t \in [0, 1]$
  retourner $i_t \in [K] \leftarrow \begin{cases} \text{aléatoirement,} & \text{avec la probabilité } \varepsilon_t \\ \text{maximisant } \hat{\mathcal{R}}_t(i_t), & \text{sinon} \end{cases}$

Méthode `MAJ`$(\mathcal{R}_t)$ **:**
  pour chaque *action* $i \in [K]$ **faire**
    $n_{t+1}(i) \leftarrow n_t(i) + \mathbb{1}_{i=i_t}$
    $\hat{\mathcal{R}}_{t+1}(i) \leftarrow \frac{n_t(i) \times \hat{\mathcal{R}}_t(i) + \mathcal{R}_t(i)}{n_{t+1}(i)}$
  fin

ε-Greedy ε-Greedy [SUTTON *et al.* 2018] (bandit A) correspond à la politique la plus simple et naïve afin de venir à bout du problème de MAB dans un environnement stochastique. Il suffit de préciser une valeur $\varepsilon_t \in [0, 1]$ renseignant dans quelle proportion nous souhaitons explorer ou exploiter au tour t.

Le bandit A présente l'initialisation des variables n et $\hat{\mathcal{R}}$ dans la méthode $\text{INIT}_{\varepsilon\text{-Greedy}}$ et la mise à jour de celles-ci (par application d'une moyenne empirique) à chaque nouvelle récompense pour la sélection de l'action i_t dans la méthode $\text{MAJ}_{\varepsilon\text{-Greedy}}$. $\mathbb{1}_\alpha$ retourne 1 si α est vrai, 0 sinon.

Afin de calculer le pseudo-regret, une borne inférieure d basée sur l'intervalle minimal doit être connue à l'avance. En admettant que $d < \min_{j \in [K]-\{i^*\}} \Delta_j$, où Δ_j est l'intervalle de sous-optimalité entre l'action optimale i^* et j, et qu'on initialise $\varepsilon_t = {}^{K}/_{(d^2 t)}$, ε-Greedy accomplit un pseudo-regret en $O({}^{K \ln(T)}/_{d^2})$.

Pour notre usage, nous simplifions et déclarons globalement la variable ε lors de l'initialisation du bandit. ε est souvent fixée à ${}^{1}/_{10}$.

UCB1 Les politiques basées sur la *borne supérieure de confiance* (UCB — *Upper Confidence Bound*) sont couramment utilisées dans les environnements stochastiques : [AUER *et al.* 2002a]. Nous présentons UCB1 qui est la politique la plus simple de cette famille de bandits. L'implantation (bandit B) présente les mêmes initialisation et mise à jour que ε-Greedy. À chaque tour t, $\text{SELECT}_{\text{UCB1}}$ sélectionne le bras qui maximise la moyenne $\hat{\mathcal{R}}_t(i_t)$ et le biais $\sqrt{{}^{8 \ln(t)}/_{n_t(i_t)}}$ correspondant à la borne supérieure de confiance associée à l'action : moins la proportion de sélection d'une action par rapport au nombre d'essais total est importante, plus cette borne est importante et donne des chances à l'action d'être tirée à nouveau.

Bandit B : UCB1(K : actions)

Méthode INIT() **:**
  pour chaque *action* $i \in [K]$ **faire**
    $n_1(i) \leftarrow 0$
    $\hat{\mathcal{R}}_1(i) \leftarrow 0$
  fin

Méthode SELECT(t) **:**
  retourner $i_t \in [K]$ maximisant $\hat{\mathcal{R}}_t(i_t) + \sqrt{\dfrac{8 \ln(t)}{n_t(i_t)}}$

Méthode MAJ($\mathcal{R}_t$) **:**
  pour chaque *action* $i \in [K]$ **faire**
    $n_{t+1}(i) \leftarrow n_t(i) + \mathbb{1}_{i=i_t}$
    $\hat{\mathcal{R}}_{t+1}(i) \leftarrow \frac{n_t(i) \times \hat{\mathcal{R}}_t(i) + \mathcal{R}_t(i)}{n_{t+1}(i)}$
  fin

Sous l'hypothèse que, pour chaque action i, $\mathcal{R}_t(i)$ est tiré (indépendamment au hasard) selon une distribution fixe sur $[0, 1]$, UCB1 obtient un pseudo-regret en $O(\sqrt{TK \ln K})$.

MOSS L'algorithme de *stratégie optimale Minimax dans le cas stochastique* (MOSS — *Minimax Optimal Strategy in the Stochastic case*) [AUDIBERT & BUBECK 2009] (bandit C) est une instance provenant de la famille d'UCB.

La seule différence avec UCB1 est liée au niveau de confiance accordé à chaque bras. MOSS ne prend pas seulement en compte le nombre de sélections de chaque bras, mais aussi le nombre total

Bandit C : MOSS(K : actions)

Méthode INIT() **:**
  pour chaque *action* $i \in [K]$ **faire**
    $n_1(i) \leftarrow 0$
    $\hat{\mathcal{R}}_1(i) \leftarrow 0$
  fin

Méthode SELECT(t) **:**
  retourner $i_t \in [K]$ maximisant $\hat{\mathcal{R}}_t(i_t) + \sqrt{\frac{4}{n_t(i_t)} \ln^+ \left(\frac{t}{K \times n_t(i_t)} \right)}$

Méthode MAJ($\mathcal{R}_t$) **:**
  pour chaque *action* $i \in [K]$ **faire**
    $n_{t+1}(i) \leftarrow n_t(i) + \mathbb{1}_{i=i_t}$
    $\hat{\mathcal{R}}_{t+1}(i) \leftarrow \frac{n_t(i) \times \hat{\mathcal{R}}_t(i) + \mathcal{R}_t(i)}{n_{t+1}(i)}$
  fin

de bras et le nombre d'essais effectués : $i_t \in [K]$ maximisant $\hat{\mathcal{R}}_t(i_t) + \sqrt{4/n_t(i_t) \ln^+ (t/K \times n_t(i_t))}$ où $\ln^+(x) = \ln \max\{1, x\}$. MOSS obtient un pseudo-regret en $O(\sqrt{TK})$.

TS L'échantillonnage de Thompson (TS — *Thompson Sampling*) [THOMPSON 1933, RUSSO *et al.* 2018, RUSSO & ROY 2016] (bandit D) maintient une distribution `beta` des récompenses de chaque action où α correspond à la somme des récompenses et β correspond à la somme du complément à 1 de chaque récompense (INIT_{TS} et MAJ_{TS}). Chaque action est sélectionnée aléatoirement suivant sa probabilité d'être optimale ($\text{SELECT}_{\text{TS}}$). TS obtient un pseudo-regret en $O(\sqrt{TK \ln K})$.

Bandit D : TS(K : actions)

Méthode INIT() **:**
  pour chaque *action* $i \in [K]$ **faire**
    $\alpha_i \leftarrow 1$
    $\beta_i \leftarrow 1$
  fin

Méthode SELECT(t) **:**
  retourner $i_t \in [K]$ maximisant $\texttt{beta}(\alpha_{i_t}, \beta_{i_t})$

Méthode MAJ($\mathcal{R}_t$) **:**
  pour chaque *action* $i \in [K]$ **faire**
    $\alpha_i = \alpha_i + \mathcal{R}_t(i)$
    $\beta_i = \beta_i + \mathbb{1}_{i=i_t}(1 - \mathcal{R}_t(i))$
  fin

2.2.3 Environnement adversarial

La différence avec la version stochastique est que le bandit adversarial prend en compte un environnement choisissant sa fonction de récompense à chaque tour de jeu. Ainsi, la probabilité de récompense stationnaire associée à chaque bras n'est plus assurée. Le bandit (ou joueur) joue donc contre un environnement dynamique.

`EXP3` Lorsque l'on en connaît très peu sur les fonctions de récompense $\mathcal{R}_1, \cdots, \mathcal{R}_t$, l'*algorithme à pondération exponentielle pour l'exploration et l'exploitation* (`EXP3` — *EXPonential-weight algorithm for EXPloration and EXPloitation*) [AUER *et al.* 2002b] est sans doute le meilleur candidat pour le problème du bandit à K bras. A savoir, `EXP3` peut fonctionner dans des environnements non-stochastiques, pour lesquels aucune hypothèse statistique n'est faite sur les fonctions de récompense. Plusieurs variantes d'`EXP3` ont été proposées dans la littérature, mais nous utilisons ici la version la plus simple définie dans [BUBECK & CESA-BIANCHI 2012].

`EXP3` maintient une distribution de probabilité π_t sur $\{1, \cdots, K\}$. Plus précisément, la procédure $\text{INIT}_{\texttt{EXP3}}$ définit le vecteur initial π_1 à la distribution uniforme $(1/K, \cdots, 1/K)$. Au cours de chaque tour t, la procédure $\text{SELECT}_{\texttt{EXP3}}$ tire simplement un bras i_t selon la distribution π_t. En fonction de la récompense observée $\mathcal{R}_t(i_t)$, la procédure $\text{MAJ}_{\texttt{EXP3}}$ met à jour la distribution π_t selon la règle multiplicative de pondération et de mise à jour :

$$\pi_{t+1}(i) = \frac{\exp(\eta_t \overline{R_t}(i))}{\sum_{j=1}^{K} \exp(\eta_t \overline{R_t}(j))}$$

où

$$\overline{R_t}(i) = \sum_{s=1}^{t} \frac{\mathcal{R}_s(i)}{\pi_s(i)} \mathbb{1}_{i \sim \pi_s}$$

et $\mathbb{1}_{i \sim \pi_s}$ indique si i était le bras choisi à l'essai s, ou non. En considérant que la fonction de récompense est dans $[0, 1]$, le paramètre de taille de pas η_t peut être fixé à $\sqrt{\ln K / tK}$ afin d'obtenir une limite de pseudo-regret en $O(\sqrt{TK \ln K})$.

Bandit E : `EXP3`(K : actions)

Méthode `INIT()` **:**
  pour chaque *action* $i \in [K]$ **faire**
    $\pi_1(i) = 1/K$
  fin

Méthode `SELECT`(t) **:**
  retourner $i_t \sim \pi_t$

Méthode `MAJ`($\mathcal{R}_t$) **:**
  pour chaque *action* $i \in [K]$ **faire**
    $\overline{R_t}(i) = \sum_{s=1}^{t} \frac{\mathcal{R}_s(i)}{\pi_s(i)} \mathbb{1}_{i \sim \pi_s}$
    $\pi_{t+1}(i) = \frac{\exp(\eta_t \overline{R_t}(i))}{\sum_{j=1}^{K} \exp(\eta_t \overline{R_t}(j))}$
  fin

2.2.4 Politique de choix uniforme

Afin de mettre en concurrence les précédentes politiques adaptatives grâce à leur capacité à apprendre de la fonction de récompense, la politique de choix uniforme est naturellement proposée à titre comparatif. Si la politique de choix uniforme est capable de produire des résultats équivalents ou meilleurs, cela permet de mettre en avant une fonction de récompense ou une politique mal adaptée pour un problème donné.

Bandit F : `UNI`(K : actions)

1 **Méthode** `INIT()` **:**
  Aucune opération

2 **Méthode** `SELECT`(t) **:**
3 **retourner** $i_t \in [K]$ aléatoirement et uniformément

4 **Méthode** `MAJ`($\mathcal{R}_t$) **:**
  Aucune opération

Ainsi, le bandit F ne prend pas en compte les récompenses proposées par le biais de sa méthode $\text{MAJ}_{\texttt{UNI}}$ et ne nécessite pas l'initialisation d'une structure particulière ($\text{INIT}_{\texttt{UNI}}$). Seule la méthode $\text{SELECT}_{\texttt{UNI}}$ est instanciée afin de choisir aléatoirement et uniformément l'action suivante i_t. Bien que nous employons l'interface algorithmique d'un bandit pour déclarer `UNI`, cette politique ne peut pas être considérée comme un bandit puisqu'aucun apprentissage n'est appliqué.

2.3 Applications

Afin de mieux appréhender l'intérêt de chaque politique en fonction de la nature du problème, nous proposons deux mises en situation inspirées du monde réel et académique. Une première application consistera en l'expérimentation de traitements médicaux pour lesquels il faudra converger le plus rapidement possible vers le traitement le plus optimal. Une seconde application consistera à jouer au jeu du *pierre-papier-ciseaux* contre un environnement changeant de technique avec le temps. Ces deux applications proposent respectivement des environnements stochastique et adversarial.

2.3.1 Traitement médical

Pour cette première application, un hôpital dispose d'un ensemble de traitements médicaux contre une maladie donnée. Chaque application du traitement auprès d'un patient donne lieu à un feedback de celui-ci. Ce feedback correspond à un score entre 0 et 1 correspondant à l'efficacité du traitement sur ce patient (vitesse de guérison, effet secondaire, *etc.*). Ainsi, le but de l'hôpital est de converger le plus rapidement possible vers le traitement maximisant la rémission des patients et le feedback associé.

Ainsi formulée, cette tâche peut être présentée comme un problème de bandit multi-bras à K bras, où chaque bras est un traitement médical. Chaque essai t va correspondre à la sélection d'un traitement i_t pour un nouveau patient et au retour de l'efficacité du traitement sur ce patient avec la récompense $\mathcal{R}_t(i_t)$. Le but est donc d'explorer l'ensemble des traitements disponibles afin de vite converger vers le traitement le plus efficace et l'exploiter.

Afin de matérialiser ce problème, nous prenons $K = 5$ traitements médicaux dont l'efficacité est représentée par 5 distributions de probabilité $\nu_1, \ldots, \nu_5$ dans $[0, 1]$ associées respectivement aux bras

$1, \ldots, 5$. Les distributions $\nu_1, \ldots, \nu_K$ suivent une loi normale tronquée correspondant aux moyennes suivantes :

$$\mu_1 = 0.4, \mu_2 = 0.87, \mu_3 = 0.5, \mu_4 = 0.9, \mu_5 = 0.85$$

avec un écart-type de 5%. Ces précédentes moyennes ne sont évidemment pas connues du bandit et celui-ci doit les estimer au plus vite afin de converger vers le bras le plus prometteur. T, l'horizon, est fixée à 10 000 essais.

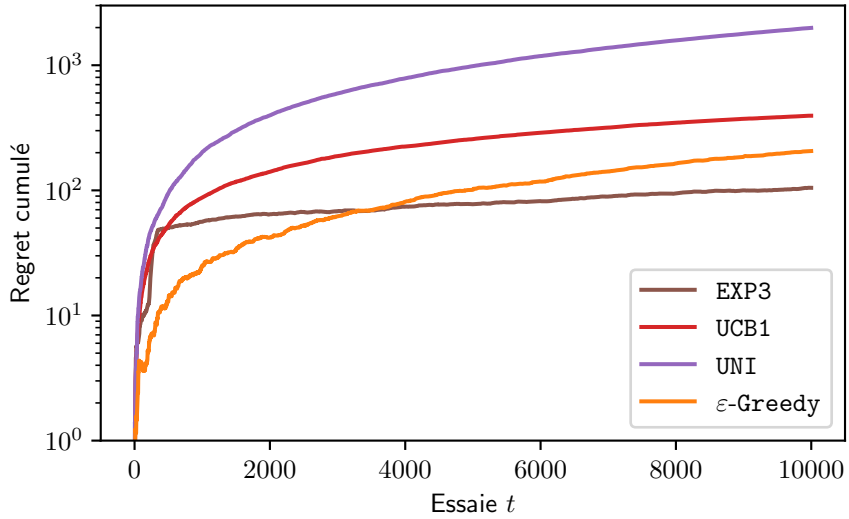


FIGURE 2.1 – Regret cumulé de différentes stratégies de sélection de traitements médicaux [♌]

La figure 2.1 présente le regret cumulé par les bandits ε-Greedy, UCB1, EXP3 et la stratégie de choix uniforme UNI à chaque essai t. Évidemment, la politique uniforme est la plus mal placée avec un regret cumulé constamment supérieur au regret des politiques de bandit. Au sein de ces politiques de bandit, nous remarquons que dans les 3 000 premiers essais, ε-Greedy est la stratégie la plus efficace. En effet, les distributions suivant la loi normale étant assez simple à déterminer, ε-Greedy n'a qu'à se baser sur les quelques premiers essais afin de déterminer sur quel bras converger. Malheureusement pour cette stratégie en continuelle exploration (10%), celle-ci se fait dépasser peu après les 3 000 essais par EXP3. Pour les mêmes raisons, cette politique adversariale est capable de converger très vite grâce aux récompenses suivant une distribution normale. UCB1, beaucoup plus prudent quant à la distribution qu'il peut rencontrer, poursuit son exploration et tend à être plus efficace qu'ε-Greedy au-delà des $T = 10\,000$ essais.

2.3.2 Pierre-papier-ciseaux

Pour cette seconde application, nous prenons l'exemple du jeu *pierre-papier-ciseaux*. Nous faisons face à un environnement possédant une stratégie adversariale à ce jeu. Le but du joueur est de déterminer, comme dans l'exemple précédent, quel est le coup qui lui permettra de gagner le plus souvent, et ainsi, le coup que l'adversaire/environnement a tendance à le plus utiliser. Cette fois-ci l'environnement est adversarial, c'est-à-dire, que la distribution de ses choix n'est pas forcément fixe à travers le temps. Par exemple, l'environnement peut majoritairement jouer les ciseaux en début de jeu, puis finalement choisir principalement le papier en fin de partie.

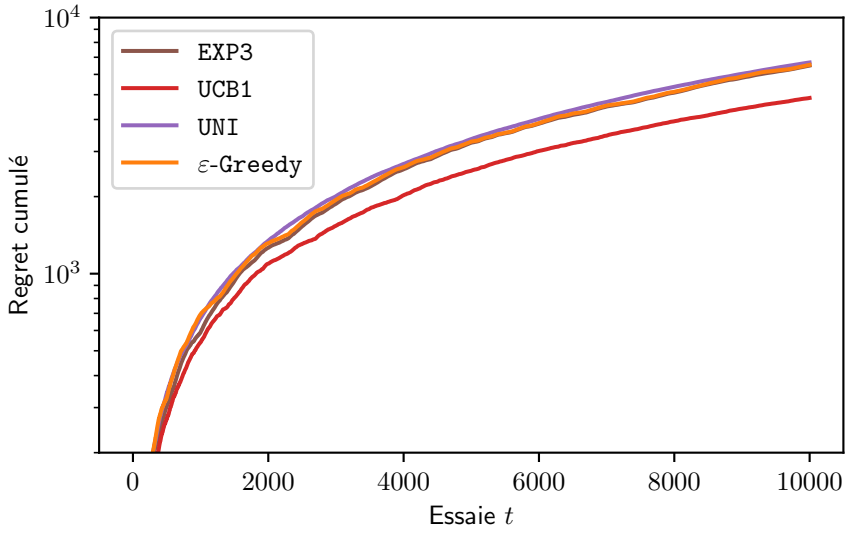


FIGURE 2.2 – Regret cumulé de différentes stratégies de sélection d'action au jeu *pierre-papier-ciseaux* [↻]

Afin de modéliser cette application, nous prenons $K = 3$ bras correspondant aux 3 actions possibles de l'environnement. T est fixé à 10 000 essais. L'environnement choisit avec une probabilité $3/5$ l'un des bras et de $1/5$ les deux autres bras. L'association de ces probabilités avec les actions possible du jeu est réaffectée aléatoirement tous les 100 essais.

La figure 2.2 montre les résultats des bandits `ε-Greedy`, `UCB1`, `EXP3` et de la stratégie de choix uniforme `UNI`. Cette fois-ci, nous remarquons que l'adaptabilité à l'environnement est très compliquée, que ce soit pour le bandit adversarial ou les bandits stochastiques. L'un des pièges dans lequel est tombé le bandit adversarial `EXP3` est qu'il a de nouveau bien trop vite convergé vers le bras lui apportant le plus en début de recherche. Ainsi, pour lui, comme pour le bandit `ε-Greedy`, leur regret cumulé est similaire à celui de la stratégie uniforme. En ce qui concerne `UCB1`, celui-ci est légèrement meilleur tout en restant dans l'ordre de grandeur du regret cumulé de la stratégie uniforme.

2.4 Bandit duelliste et autres variations du problème

Dans cette section, nous considérons le problème du bandit multi-bras avec un type particulier de feedback *préférentiel* [YUE & JOACHIMS 2009, RADLINSKI *et al.* 2008] entre deux actions. Dans le problème classique du bandit, l'apprenant reçoit un *feedback numérique* sur ses choix. Cependant, comme nous le verrons bientôt, un feedback numérique n'est pas toujours disponible dans les scénarios pratiques.

2.4.1 Motivation

Les humains trouvent qu'il est beaucoup plus facile de choisir parmi l'une des options proposées plutôt que de donner un avis absolu et numérique sur un seul choix. Par exemple, si les deux questions suivantes sont posées à un groupe de personnes :

— préférez-vous la dénomination *pain au chocolat* ou *chocolatine* ?

— quelle note sur 100 donneriez-vous à la dénomination *pain au chocolat* ?

La première question recevrait un plus grand nombre de réponses que la seconde. Par conséquent, le feedback préférentiel est naturellement adapté à de nombreuses applications pratiques où l'on attend des

humains qu'ils fournissent un feedback, comme les préférences perçues par l'utilisateur pour un produit, où une perception relative : « A est meilleur que B » est plus facile à obtenir que sa contrepartie absolue : « La valeur de A est 62, celle de B est 31 ».

Une application plus commerciale du feedback préférentiel provient des systèmes de tri d'informations [SUI *et al.* 2018] (par exemple, les moteurs de recherche [RADLINSKI & JOACHIMS 2007]) où les utilisateurs fournissent un feedback implicite sur les résultats fournis. Ce feedback implicite, et interprété sous forme de préférence, est recueilli au moment où l'utilisateur sélectionne un résultat plutôt qu'un autre.

2.4.2 Formalisation

Comme le problème du bandit classique, le problème du bandit duelliste peut lui aussi être subdivisé en deux catégories, en fonction de la stationnarité des récompenses : stochastique et adversarial.

Le problème du *bandit duelliste* à K bras est une variation du problème classique du bandit multi-bras introduit par Yue et Joachims [YUE & JOACHIMS 2009] pour formaliser le dilemme exploration/exploitation dans l'apprentissage à partir de feedbacks relatifs, aussi appelé feedback par préférence. Pour être en mesure de modéliser les scénarios pratiques décrits précédemment, l'apprenant doit choisir deux bras dans l'ensemble $[K]$ à chaque essai t. En retour, l'apprenant observe le gagnant du duel, c'est-à-dire quelle action a donné la meilleure récompense. Notez que l'apprenant n'a pas accès aux récompenses des actions sélectionnées, mais seulement au feedback de préférence. Cependant, la performance de l'apprenant est jugée sur la base de la récompense moyenne des actions sélectionnées et non du feedback. La difficulté de ce problème vient du fait que l'apprenant n'a aucun moyen d'observer directement la récompense des actions sélectionnées. L'apprenant doit donc trouver un moyen de déduire du feedback les informations nécessaires sur les récompenses.

Comme indiqué, cette formulation se caractérise par des préférences. Puisque la préférence de chaque paire d'actions doit être spécifiée, une façon naturelle de représenter l'ensemble de ces paires de préférence est une matrice carrée dont la dimension est égale au nombre d'actions : $K \times K$. On l'appelle une *matrice de préférence* P. Pour chaque paire (a, b) d'actions, l'élément $P_{a,b}$ contient la probabilité inconnue avec laquelle a gagne le duel contre b.

Un exemple de matrice de préférences est présenté ci-dessous :

$$P = \begin{bmatrix} 1/2 & P_{1,2} & P_{1,3} & \dots & P_{1,K} \\ P_{2,1} & 1/2 & P_{2,3} & \dots & P_{2,K} \\ P_{3,1} & P_{3,2} & 1/2 & \dots & P_{3,K} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ P_{K,1} & P_{K,2} & P_{K,3} & \dots & 1/2 \end{bmatrix}$$

La matrice P doit satisfaire la propriété de symétrie suivante :

$$\forall a, b \in [K], P_{a,b} + P_{b,a} = 1$$

En utilisant ces valeurs de préférence, le regret cumulé peut être défini comme suit :

$$\texttt{regret}_T = \sum_{t=1}^{T} \frac{P_{i_*,i_t} + P_{i_*,j_t} - 1}{2}$$

où i_t et j_t sont les deux bras tirés au temps t. Par la suite, nous parlons de regret de Condorcet car il coïncide avec la notion de gagnant de Condorcet i_*. Un gagnant de Condorcet est le bras, noté i_* qui est préféré à tous les autres bras, c'est-à-dire, $\forall i \in [K] - \{i_*\}, Pi_*, i > 1/2$. Notons que si $\mu_i > \mu_j$ pour

certains bras i et j, alors $P_{i,j} > 1/2$. Le bras optimal coïncide donc avec le gagnant de Condorcet dans la formulation des préférences.

2.4.3 Autres variantes

Au même titre que nous venons de faire varier le type d'action et le type de récompense, passant d'un feedback absolu à un feedback relatif, il existe encore bien d'autres variantes du problème du bandit multi-bras.

L'une d'elles est le *bandit à pure exploration*. Dans ce cadre, introduit par Bubeck et al. [BUBECK *et al.* 2009], l'apprenant explore d'abord les bras dans la phase d'exploration pour acquérir des connaissances sur les récompenses des bras, puis il exploite les connaissances acquises pour recommander un bras. Les phases d'exploration et d'exploitation ne se chevauchent pas. La mesure du regret change aussi. Nous parlons de regret simple et est défini comme la différence entre la récompense attendue du bras optimal (*a posteriori*) et la récompense attendue du bras recommandé par l'apprenant. Les récompenses des actions sélectionnées par l'apprenant pendant la phase d'exploration ne sont pas prises en compte pour le calcul du regret simple ; seule la récompense de l'action recommandée est prise en compte.

Beaucoup de paradigmes de bandit multi-bras existent, faisant varier le type d'action : sélection d'un bras, d'un couple dans le cadre duelliste ou même des combinaisons plus importantes de bras. En lien, le feedback peut être absolu, relatif à un duel ou même être un vecteur de récompenses lorsqu'une combinaison de bras est sélectionnée.

2.5 Conclusion

La littérature propose une grande diversité de paradigmes et politiques répondant au problème du bandit multi-bras. Ici, nous avons exprimé deux grandes familles. Dans le chapitre 4, nous proposons un nouveau framework au sein des solveurs de contraintes permettant d'accueillir les politiques proposées dans la section 2.2. Afin de contrecarrer certains défauts de ces politiques, nous proposons dans le chapitre 5 un nouveau bandit en lien avec la seconde famille de bandits présentée dans la section 2.4. Avant ceux-ci, des fonctions de récompense, récompensant l'efficacité d'un solveur, sont proposées dans le chapitre 3.

Deuxième partie

Contributions

Chapitre 3

Évaluation des Heuristiques de Choix de Variable

Sommaire

3.1 Introduction

Dans ce chapitre, les heuristiques de choix de variable sont évaluées afin de comparer leur efficacité et la capacité de chacune à résoudre plus rapidement certaines instances. L'évaluation est produite *a posteriori* des expériences. En plus de cette classique évaluation, nous essayons d'extraire différentes métriques permettant de discriminer leurs performances *en ligne*, c'est-à-dire, au cours de leur exécution.

Pour ce faire, nous décrivons dans un premier temps l'environnement d'évaluation utilisé tout au long de ce manuscrit. Nous argumentons le choix d'une configuration pour le solveur ACE. Une seconde partie propose une amélioration des heuristiques basées sur la pondération de contraintes. Les deux dernières parties correspondent à l'évaluation classique (a posteriori) d'une sélection de base d'heuristiques de choix de variable, et d'une autre évaluation discriminant celles-ci en ligne suite à la proposition de *fonctions de récompenses*.

3.2 Environnement expérimental

3.2.1 Vocabulaire

Cette section introduit quelques notions de vocabulaire utilisées pour identifier les données manipulées lors des différentes campagnes des chapitres qui suivent.

Une *campagne* contient toutes les données expérimentales qui ont été collectées, et définit la configuration de l'environnement d'exécution des solveurs (limites temporelle et spatiale, configuration des machines, *etc.*). Au cours d'une campagne, des *solveurs* sont évalués. Notons que différentes configurations d'un même solveur sont considérées comme différents solveurs, de même pour des solveurs complètement différents. Une campagne se caractérise par l'ensemble des fichiers d'entrée utilisés pour celle-ci. Dans ce contexte, *tous* les solveurs sont exécutés sur le *même* ensemble de fichiers. Enfin, une *expérimentation* correspond à l'exécution d'un solveur donné sur un fichier d'entrée donné, de sorte à ce que l'ensemble des expérimentations correspond au produit cartésien de l'ensemble des fichiers d'entrée et de celui des solveurs. Chaque expérimentation est caractérisée par les mesures qui sont pertinentes au vu des analyses à réaliser, telles que le temps d'exécution ou la mémoire utilisée par le solveur (et bien d'autres, dépendant de la configuration fournie par le solveur).

Exemple 10

Considérons une *campagne* dans laquelle nous souhaitons comparer les solveurs $\texttt{ACE}^{\alpha}$ et $\texttt{ACE}^{\beta}$. Supposons que nous souhaitons les comparer sur deux *entrées*, à savoir `CarSequencing.xml` et `RLFAP.xml`. L'ensemble des fichiers d'entrée se compose de ces deux fichiers. Les expérimentations de cette campagne sont alors :

- l'exécution d'$\texttt{ACE}^{\alpha}$ sur `CarSequencing.xml` ;
- l'exécution d'$\texttt{ACE}^{\alpha}$ sur `RLFAP.xml` ;
- l'exécution d'$\texttt{ACE}^{\beta}$ sur `CarSequencing.xml` ;
- l'exécution d'$\texttt{ACE}^{\beta}$ sur `RLFAP.xml`.

3.2.2 Expérimentation sur cluster

Les différentes campagnes décrites par la suite sont exécutées dans les mêmes conditions. Nous disposons d'un cluster composé de 50 nœuds de calcul ayant pour configuration un CPU cadencé à 3.3 GHz Intel XEON E5-2643 et 32 GB RAM ; pour un nœud donné, seule une expérimentation est exécutée à la fois. Pour chaque expérimentation, le temps maximal d'exécution (*timeout*) est fixé à 2 400 secondes – il s'agit du temps communément imposé lors des compétitions XCSP [Boussemart *et al.* 2020, Boussemart *et al.* 2016, Lecoutre & Szczepanski 2020]. L'espace est naturellement limité par les 32 GB de RAM disponibles. Les différentes campagnes sont exécutées sur une large diversité de problèmes de contraintes venant de la distribution XCSP. Nous utilisons deux benchmarks. Un premier, $\mathcal{I}_{\texttt{CSP}}$, correspond à l'ensemble des instances CSP des trois compétitions XCSP'17, XCSP'18 et XCSP'19, pour un total de 83 familles de problèmes et de 810 instances. Le second benchmark, $\mathcal{I}_{\texttt{COP}}$, est composé de l'ensemble des instances COP des compétitions XCSP'18 et de nouvelles instances mises à disposition en 2021, pour un total de 51 familles de problèmes et 697 instances.

Dans le cadre de la recherche en informatique (et plus généralement, dans n'importe quel domaine nécessitant la conception de logiciels), il est nécessaire de réaliser des expérimentations pour s'assurer que les programmes développés fonctionnent comme prévu. En particulier, il est important de vérifier que les ressources utilisées par ces programmes restent dans un périmètre contrôlé. Dans ce but, différentes

solutions logicielles telles que *runsolver* [ROUSSEL 2011] ont été proposées, à la fois pour mesurer et limiter l'utilisation des ressources temporelles et spatiales par le programme en cours d'évaluation. Afin d'appliquer les contraintes précédemment citées, *runsolver* est utilisé lors de chacune des expérimentations.

3.2.3 Analyse et reproductibilité

Cependant, respecter les limites fixées est en général insuffisant pour évaluer le comportement du programme. Il est souvent nécessaire de collecter des statistiques supplémentaires, qui peuvent être fournies par le programme lui-même (par exemple, via ses *logs*) ou par l'environnement d'exécution (par exemple, *runsolver*). Les données collectées doivent ensuite être agrégées pour évaluer la qualité des résultats du programme au travers d'une analyse statistique.

Dans ce domaine, de nombreux outils mathématiques peuvent être utilisés, et faire un choix parmi ceux-ci peut introduire des biais dans les résultats ou leur analyse. Il est donc important d'appliquer les principes de *science ouverte* et de *reproductibilité* pour permettre de rejouer l'analyse des résultats. Ainsi, ces principes ont fait l'objet d'une recommandation de l'OCDE [PILAT & FUKASAKU 2007], et des chercheurs ont déjà introduit diverses approches favorisant la reproductibilité dans le contexte d'expérimentations logicielles [FREIRE *et al.* 2016, KIM *et al.* 2018]. En particulier, il est recommandé de rendre disponible le code source des logiciels étudiés (ou, *a minima*, leurs exécutables sous forme binaire), et de diffuser les données utilisées pour les évaluer (par exemple, dans des forges logicielles). Il est également important de rendre l'analyse des résultats reproductibles, en utilisant des outils tels que le *RMarkdown* [4] ou les *notebooks Jupyter* [5].

Dans la communauté CP (tout comme dans les communautés SAT, PB, MaxSAT ou QBF par exemple), il y a souvent peu de différences sur la manière d'exécuter les solveurs. En effet, ces derniers doivent proposer des interfaces en lignes de commande qui respectent les règles imposées par l'environnement dans lequel ils sont exécutés (par exemple, durant les compétitions). De plus, la plupart des données collectées lors de l'exécution des solveurs restent le plus souvent les mêmes (par exemple, le temps d'exécution, l'utilisation de la mémoire, *etc.*). Dans ce contexte, la création d'un outil permettant d'exécuter le programme, de collecter les données qu'il produit et de les analyser présentent plusieurs avantages : tester de nouvelles fonctionnalités est plus simple, à la fois en termes d'exécution et d'analyse, et la reproductibilité des résultats est systématiquement assurée.

Partant de ces observations, nous avons développé *Metrics* [FALQUE *et al.* 2021b] (*mETRICS* signifie *rEproducible sofTware peRformance analysIs in perfeCt Simplicity*), une bibliothèque Python visant à unifier et faciliter l'analyse des expérimentations réalisées sur des solveurs. L'ambition de *Metrics* est de fournir une chaîne complète d'outils de l'exécution du solveur à l'analyse de ses performances. Actuellement, cette bibliothèque possède deux composants principaux : *Scalpel* (*sCAlPEL* signifie *extraCting dAta of exPeriments from softwarE Logs*) et *Wallet* (*wALLET* signifie *Automated tooL for expLoiting Experimental resulTs*). D'une part, *Scalpel* est conçu pour simplifier la récupération des données expérimentales. Ce module est capable de gérer une grande variété de fichiers, incluant les formats CSV, XML, JSON ou encore la sortie produite par le solveur, qui peut être décrite par l'utilisateur dans un fichier de configuration. Cette approche fait de *Scalpel* un outil à la fois flexible et simple à configurer. D'autre part, *Wallet* propose une interface simple d'utilisation pour tracer les graphiques les plus communément utilisés pour l'analyse de solveurs (tels que les *scatter plots* et les *cactus plots*) et pour calculer diverses statistiques relatives à leur exécution (en particulier, leurs scores en utilisant différentes mesures classiques). La conception de *Wallet* facilite l'intégration de l'analyse dans des *notebooks Jupyter* qui

4. https://rmarkdown.rstudio.com
5. https://jupyter.org

peuvent être facilement partagés en ligne (par exemple, *GitHub* ou *GitLab* sont capables d'afficher de tels fichiers), favorisant également la reproductibilité de l'analyse.

Dans les chapitres qui suivent, nous présentons l'utilisation de *Metrics* de la manière suivante. Pour chaque campagne donnant lieu à la production de statistiques, de tableaux ou de graphiques, celle-ci est accompagnée du logo cliquable « ⟳ »[6] menant vers l'analyse de la campagne correspondante. Une campagne comporte l'ensemble des informations nécessaires afin que le lecteur puisse, s'il le souhaite, reproduire les expérimentations, l'extraction des données, l'analyse et la production des figures présentées dans cette thèse.

3.2.4 Solveur de contraintes

Dans le cadre de la programmation par contraintes, nous utilisons un programme appelé solveur de contraintes pour résoudre un réseau de contraintes. Les solveurs sont des programmes qui traitent des réseaux de contraintes et qui prouvent soit la satisfaisabilité d'un réseau en retournant au moins une solution, soit son insatisfaisabilité. Dans le cadre des contributions présentées dans ce manuscrit, le solveur `ACE`[7], le nouvel alias d'AbsCon [MERCHEZ *et al.* 2001], est utilisé. Il s'agit d'une plateforme générique de satisfaction/optimisation sous contraintes développée en Java. Le format de modélisation pour les problèmes traitables par `ACE` est le XCSP [BOUSSEMART *et al.* 2020, BOUSSEMART *et al.* 2016, LECOUTRE & SZCZEPANSKI 2020]. Le solveur `ACE` permet de traiter des problèmes des cadres CSP et COP. `ACE` dispose de l'ensemble des heuristiques présentées dans la section 1.4.1. Son principe de résolution est celui présenté dans la section 1.3.2 : le parcours d'un arbre binaire avec assignation et réfutation d'une paire variable/valeur à chaque nœud ainsi que la propagation de la propriété de cohérence d'arc (sous réserve que chaque contrainte puisse assurer cette propriété). Par défaut, le solveur utilise l'heuristique de choix de variable $\texttt{wdeg}^{\texttt{ca}\cdot\texttt{cd}}$ présentée dans la section suivante et `min-dom` pour l'heuristique de choix de valeur. Plus précisément, il s'agit de la politique de choix de variable $\texttt{lc=2} > \texttt{wdeg}^{\texttt{ca}\cdot\texttt{cd}} > \texttt{lex}$: la politique de variables prioritaires *last-conflict* retient les deux dernières variables entrées en conflit, puis laisse la main à l'heuristique $\texttt{wdeg}^{\texttt{ca}\cdot\texttt{cd}}$ qui, en cas de variables au meilleur score égal, renvoie vers l'heuristique de choix *lexicographique*. Dans la configuration par défaut, `def`, le score des variables est régulièrement réinitialisé et la fonction de redémarrage est basée sur la suite exponentielle et autorise $\texttt{rexp}(\texttt{t})$ mauvaises décisions au run t avant que la recherche ne redémarre à sa racine. Plus précisément, `rexp` est aménagé par le solveur pour que la valeur minimale de la suite soit égale à 10 (par exemple, $\texttt{rexp}(1) = 10$) et son facteur de croissance est fixé à 1.1.

Cette section présente une première campagne recensant les résultats expérimentaux de la configuration par défaut $\texttt{ACE}^{\texttt{def}}_{\texttt{lc=2}}$ et la configuration pour laquelle nous allons opter pour les prochaines campagnes : $\texttt{ACE}^{\texttt{luby}}_{\texttt{lc=0}}$. Cette dernière configuration possède un comportement simplifié au niveau de l'heuristique de choix de variable : `lc` est initialisée à 0 et n'interfère donc pas avec la seconde (et la plus importante) heuristique sélectionnée – le but étant d'obtenir l'heuristique la plus pure possible afin de les comparer avec une plus grande justesse par la suite. Nous changeons aussi la politique de redémarrage et optons pour la suite de Luby ($u \times \texttt{luby}(t)$ mauvaises décisions où $u = 150$), afin d'obtenir un plus grand nombre de redémarrages sans nécessité de réinitialiser la suite : nous justifions ce choix dans le chapitre 4. Nous expérimentons l'ensemble Ψ_1 de solveurs suivant :

$$\left\{ \texttt{ACE}^{r}_{\texttt{lc=}k} \;\middle|\; \begin{array}{l} r \in \{\texttt{def}, \texttt{luby}\} \\ k \in \{0, 2\} \end{array} \right\} \qquad (\Psi_1)$$

6. L'ensemble des campagnes mène vers le répertoire *git* suivant : https://gitlab.com/productions-hwattez/solveurs-de-contraintes-autonomes/doctorat/experimentations

7. https://www.cril.univ-artois.fr/~lecoutre/#/softwares

La table statistique 3.1 présente les quatre configurations Ψ_1, testées sur l'ensemble des instances $\mathcal{I}_{\text{CSP}}$, partant du solveur par défaut $\text{ACE}^{\text{def}}_{\text{lc=2}}$ jusqu'au solveur que nous considérons « par défaut » pour la suite de ce manuscrit : $\text{ACE}^{\text{luby}}_{\text{lc=0}}$. Plusieurs statistiques sont présentées à travers ce tableau :

— #RÉS. correspond au nombre d'instances résolues ;
— TPS (S) correspond au temps de résolution en secondes (comprend les instances résolus par au moins un solveur) ;
— #RÉS. COMM. correspond au nombre d'instances résolues par l'ensemble des solveurs ;
— TPS COMM. (S) correspond au temps en secondes pour résoudre les instances communes ;
— #TOT. correspond au nombre total d'instances de cette campagne : cette métrique est différente de celle déclarée pour $\mathcal{I}_{\text{CSP}}$ (810 instances) car les instances n'étant pas résolues par au moins un solveur sont ignorées afin de supprimer une partie du bruit.

Le bruit supprimé permet notamment d'y voir plus clair au niveau du temps de résolution ; dans le cas où nous incluons les instances irrésolues, la différence dans les temps de résolution paraît minime et négligeable.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
$\text{ACE}^{\text{def}}_{\text{lc=2}}$	**558**	**73 219**	528	25 127	571
$\text{ACE}^{\text{luby}}_{\text{lc=2}}$	552	88 042	528	29 295	571
$\text{ACE}^{\text{def}}_{\text{lc=0}}$	551	85 157	528	**24 886**	571
$\text{ACE}^{\text{luby}}_{\text{lc=0}}$	543	113 915	528	31 457	571

TABLE 3.1 – Comparaison de différentes configurations basiques du solveur ACE [Ψ_1, $\mathcal{I}_{\text{CSP}}$, ↻]

Nous constatons à travers cette figure que la configuration retenue décroît légèrement l'efficacité de résolution : $\text{ACE}^{\text{luby}}_{\text{lc=0}}$ résout 15 instances de moins que la configuration $\text{ACE}^{\text{def}}_{\text{lc=2}}$. De même, le temps de résolution sur les 528 instances communément résolues est allongé.

Ainsi, la simplification de la configuration du solveur ACE montre des performances légèrement affaiblies, mais nous permet par la suite d'interpréter plus simplement les résultats. La plupart des techniques utilisées dans les chapitres suivants sont aussi applicables aux configurations précédemment citées. Les principales implantations dans le solveur ACE sont décrites dans l'annexe A.1.

3.3 Affinement de la pondération de contraintes

3.3.1 Une amélioration native de $\texttt{wdeg}^{\texttt{unit-2004}}$ dans ACE

En extension de l'heuristique III, ACE propose nativement l'implantation définie par l'heuristique V. Cette première amélioration consiste à affiner la façon dont le poids évolue : non plus globalement pour toutes les variables d'une contrainte, mais seulement pour celles qui sont futures. Il est alors nécessaire d'introduire un tableau pour chaque contrainte c afin d'enregistrer le score (poids) local w_c^x d'une variable x impliquée dans c. La particularité réside ainsi dans la manière d'interpréter un conflit : au lieu d'augmenter globalement le poids de la contrainte ayant causé le conflit, seules les variables futures (non encore assignées) dans la portée de la contrainte voient leur poids incrémenté.

À travers la table 3.2, nous observons comment se placent $\texttt{wdeg}^{\texttt{unit-2004}}$ et $\texttt{wdeg}^{\texttt{unit}}$ et leur agrégation correspondante dom/wdeg. De 9 à 21 instances additionnelles sont résolues par rapport à la version originelle de l'heuristique. Le temps commun de résolution est aussi réduit.

Heuristique V : $\texttt{wdeg}^{\texttt{unit}}(\mathcal{P} = (\mathcal{X}, \mathcal{C}) : \text{CN})$

1 **Méthode** INIT() **:**
2 | **pour chaque** $c \in \mathcal{C}$ **faire**
3 | | **pour chaque** $x \in \texttt{scp}(c)$ **faire**
4 | | | $w_c^x \leftarrow 0$
5 | | **fin**
6 | **fin**

7 **Méthode** AVANTRUN() **:**
| *Aucune opération*

8 **Méthode** APRÈSCONFLIT(c) **:**
9 | **pour chaque** $x \in \texttt{fut}(c)$ **faire**
10 | | $r \leftarrow 1$
11 | | $w_c^x \leftarrow w_c^x + r$
12 | **fin**

13 **Méthode** APRÈSASSIGNATION($\mathcal{P}_{av}$, $\mathcal{P}_{ap}$) **:**
| *Aucune opération*

14 **Méthode** SCORE(x) **:**
15 | **retourner** $\sum_{c \in \mathcal{C} : x \in scp(c) \wedge |\texttt{fut}(c)| > 1} w_c^x$

3.3.2 Affinement de l'heuristique $\texttt{wdeg}^{\texttt{unit}}$

Même si $\texttt{wdeg}^{\texttt{unit}}$ surpasse légèrement $\texttt{wdeg}^{\texttt{unit-2004}}$, on peut regretter que la pondération des contraintes reste très simpliste et ne fasse pas de différence entre les contraintes. Par exemple, des caractéristiques comme l'arité des contraintes et l'état des domaines des variables participantes sont totalement ignorées puisque l'incrément est statique (c'est-à-dire, 1). C'est pourquoi nous proposons d'affiner la pondération des contraintes en exploitant ces informations.

Plus précisément, nous introduisons quatre variantes à travers la déclaration générique de l'heuristique VI. Un paramètre α est introduit à travers l'intitulé de l'heuristique ($\texttt{wdeg}^{\alpha}$) accompagné d'une fonction $f^{\alpha}(c, x)$ (ligne 10) correspondant à la récompense attribuée à une paire contrainte/variable (c et x, où c est la contrainte conflictuelle et $x \in \texttt{scp}(c)$) donnée :

$\texttt{wdeg}^{\texttt{ia}}$ ia est la variante pour laquelle l'arité *initiale* de la contrainte est utilisée :

$$f^{\texttt{ia}}(c, x) = \frac{1}{|\texttt{scp}^{\texttt{init}}(c)|}$$

$\texttt{wdeg}^{\texttt{ca}}$ ca est la variante pour laquelle on utilise l'arité *courante* de la contrainte (c'est-à-dire, le nombre de variables futures de la contrainte). Nous rappelons que les variables futures d'une contrainte sont les variables de la portée dont les domaines ont une taille supérieure à un :

$$f^{\texttt{ca}}(c, x) = \frac{1}{|\texttt{scp}(c)|}$$

wdeg$^{\texttt{id}}$ `id` est la variante pour laquelle on utilise la taille du *domaine initiale* de la variable :

$$f^{\texttt{id}}(c,x) = \frac{1}{|\texttt{dom}^{\texttt{init}}(x)|}$$

wdeg$^{\texttt{cd}}$ `cd` est la variante pour laquelle on utilise la taille du *domaine courant* de la variable :

$$f^{\texttt{cd}}(c,x) = \frac{1}{|\texttt{dom}(x)|}$$

wdeg$^{\texttt{ca.cd}}$ `ca.cd` combine à la fois l'arité courante et le domaine courant :

$$f^{\texttt{ca.cd}}(c,x) = \frac{1}{|\texttt{scp}(c)| \times |\texttt{dom}(x)|}$$

Heuristique VI : $\texttt{wdeg}^{\alpha}(\mathcal{P} = (\mathcal{X}, \mathcal{C}) : \text{CN})$

Méthode INIT() **:**
  pour chaque $c \in \mathcal{C}$ **faire**
    pour chaque $x \in \texttt{scp}(c)$ **faire**
      $w_c^x \leftarrow 0$
    fin
  fin

Méthode AVANTRUN() **:**
  Aucune opération

Méthode APRÈSCONFLIT(c) **:**
  pour chaque $x \in \texttt{fut}(c)$ **faire**
    $r \leftarrow f^{\alpha}(c,x)$
    $w_c^x \leftarrow w_c^x + r$
  fin

Méthode APRÈSASSIGNATION($\mathcal{P}_{av}$, $\mathcal{P}_{ap}$) **:**
  Aucune opération

Méthode SCORE(x) **:**
  retourner $\sum_{c \in \mathcal{C} : x \in scp(c) \wedge |\texttt{fut}(c)|>1} w_c^x$

3.3.3 Expérimentation des heuristiques wdeg et des variantes affinées

Afin de tester en largeur ces nouvelles variantes, nous proposons d'expérimenter l'ensemble Ψ_2 de solveurs suivant :

$$\left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc=0}}(h^{\alpha}) \;\middle|\; \begin{array}{l} h \in \{\texttt{wdeg}, \texttt{dom/wdeg}\} \\ \alpha \in \{\texttt{unit-2004}, \texttt{unit}, \texttt{chs}, \texttt{ia}, \texttt{ca}, \texttt{id}, \texttt{cd}, \texttt{ca.cd}\} \end{array} \right\} \qquad (\Psi_2)$$

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
$\texttt{wdeg}^{\texttt{ca.cd}}$	**543**	129 037	444	10 829	577
$\texttt{dom/wdeg}^{\texttt{ia}}$	537	**125 007**	444	9 061	577
$\texttt{dom/wdeg}^{\texttt{ca.cd}}$	537	127 201	444	**8 481**	577
$\texttt{dom/wdeg}^{\texttt{ca}}$	535	137 356	444	10 508	577
$\texttt{wdeg}^{\texttt{chs}}$	534	137 795	444	10 733	577
$\texttt{dom/wdeg}^{\texttt{chs}}$	529	145 235	444	9 289	577
$\texttt{wdeg}^{\texttt{ca}}$	527	156 464	444	10 417	577
$\texttt{wdeg}^{\texttt{ia}}$	526	153 467	444	11 181	577
$\texttt{dom/wdeg}^{\texttt{id}}$	525	157 617	444	13 189	577
$\texttt{dom/wdeg}^{\texttt{unit}}$	522	169 710	444	9 647	577
$\texttt{wdeg}^{\texttt{id}}$	516	181 194	444	12 001	577
$\texttt{wdeg}^{\texttt{unit}}$	510	192 895	444	15 896	577
$\texttt{dom/wdeg}^{\texttt{cd}}$	510	194 795	444	9 330	577
$\texttt{wdeg}^{\texttt{unit}-2004}$	501	223 126	444	20 244	577
$\texttt{dom/wdeg}^{\texttt{unit}-2004}$	499	217 489	444	12 757	577
$\texttt{wdeg}^{\texttt{cd}}$	487	248 400	444	16 916	577

TABLE 3.2 – Comparaison des variantes de $\texttt{wdeg}^{\alpha}$ et $\texttt{dom/wdeg}^{\alpha}$ [Ψ_2, $\mathcal{I}_{\texttt{CSP}}$, ↻]

En plus de tester les différentes variantes sur la simple heuristique basée sur la pondération, nous prenons aussi en compte sa version agrégée avec dom : dom/wdeg.

La table 3.2 présente exhaustivement l'ensemble des variantes de wdeg Ψ_2 présentés dans ce manuscrit. Nous observons rapidement que $\texttt{wdeg}^{\texttt{ca.cd}}$ est la variante s'en sortant le mieux dans la configuration actuelle du solveur, en accord avec l'étude [WATTEZ *et al.* 2019b]. $\texttt{wdeg}^{\texttt{ca.cd}}$ résoud 9 instances supplémentaires par rapport à la meilleure version actuelle de cette heuristique ($\texttt{wdeg}^{\texttt{chs}}$) et 21 supplémentaires par rapport à la variante de wdeg proposée par ACE.

La figure 3.1 propose une autre perspective pour ces résultats. Ceux-ci sont présentés sous forme d'un cactus plot. Ce graphique considère toutes les instances résolues par chaque solveur. Chaque courbe du graphique représente un solveur. Les entrées sont classées par temps de résolution pour chaque solveur afin de construire cette figure : l'axe des x correspond au rang de l'instance résolue et l'axe des y au temps pris pour résoudre l'instance, de sorte que plus la courbe est positionnée à droite, meilleur est le solveur.

Afin de faciliter la lecture de la figure 3.1 les solveurs $\texttt{wdeg}^{\alpha}$ et $\texttt{dom/wdeg}^{\alpha}$ sont, respectivement, tracés en pointillés et en ligne continue. Une même couleur est utilisée pour une variante α donnée. Au sein de la légende, les solveurs sont classés par ordre décroissant de résolution au moment du timeout. Ce genre de figure donne une information complémentaire : l'ordre des solveurs est conservé quant au timeout, mais nous observons que si le timeout avait été à 1 200 secondes, le solveur $\texttt{dom/wdeg}^{\texttt{ia}}$ aurait été le mieux classé.

3.4 Sélection de l'ensemble $\mathcal{H}^{\texttt{base}}$ d'heuristiques de choix de variable

Cette section décrit la sélection d'heuristiques de choix de variable utilisées, par la suite, comme ensemble de base pour nos expérimentations : $\mathcal{H}^{\texttt{base}}$. Cette sélection est testée avec différents tie-breakers et est analysée afin de mettre en évidence la robustesse complémentaire de ces mécanismes.

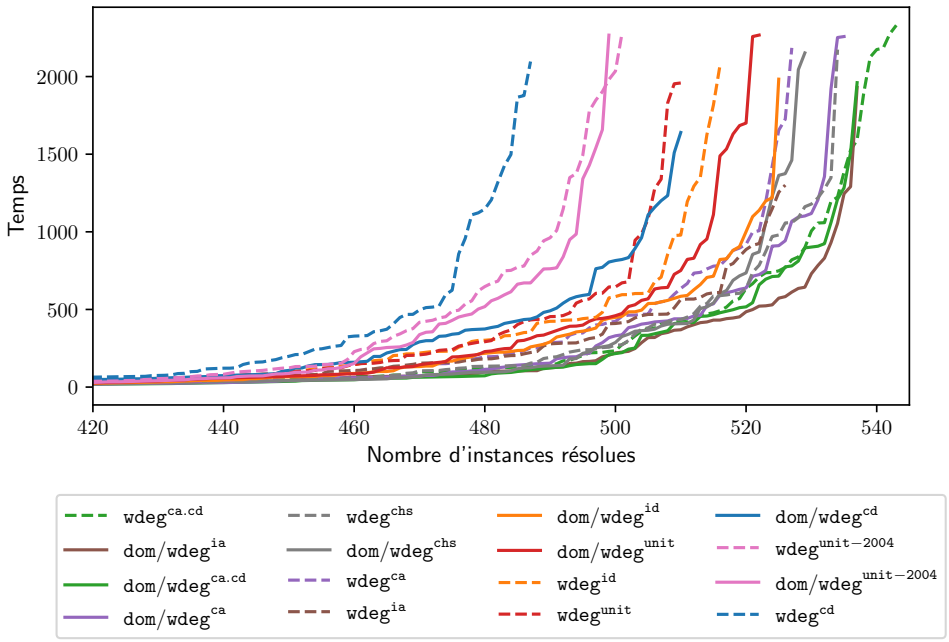


FIGURE 3.1 – Cactus plot des variantes de $\texttt{wdeg}^{\alpha}$ et $\texttt{dom/wdeg}^{\alpha}$ [Ψ_2, $\mathcal{I}_{\texttt{CSP}}$, ⟳]

3.4.1 Sélection des heuristiques

Soit $\mathcal{H}^{\texttt{base}}$, une sélection d'heuristiques de choix de variables, provenant de l'état de l'art présenté dans la section 1.4.1, utilisées comme base pour l'analyse qui suit et dans de futures études de ce manuscrit :

$$\mathcal{H}^{\texttt{base}} = \left\{\texttt{abs}, \texttt{ibs}, \texttt{dom/ddeg}, \texttt{dom/wdeg}^{\texttt{chs}}, \texttt{wdeg}^{\texttt{ca.cd}}\right\}$$

Cet ensemble est composé de quatre heuristiques adaptatives et d'une heuristique dynamique :
— `abs` est une forme simplifiée de l'heuristique proposée dans [MICHEL & HENTENRYCK 2012] où l'échantillonnage n'est pas appliquée — nous voyons par la suite que cela complique l'implantation de stratégies de recherche de la meilleure heuristique ;
— `ibs` est, pour la même raison que précédemment, une version sans échantillonnage de l'heuristique présentée dans l'étude [REFALO 2004] ;
— `dom/ddeg` est l'une des heuristiques dynamiques les plus efficaces de sa catégorie ;
— $\texttt{dom/wdeg}^{\texttt{chs}}$ est la meilleure variante extraite de l'étude [CHERIF *et al.* 2020b] et en accord avec les résultats de la section 3.3.3 ;
— $\texttt{wdeg}^{\texttt{ca.cd}}$ est la meilleure des variantes de `wdeg` proposée précédemment et aussi dans l'étude [WATTEZ *et al.* 2019b].

3.4.2 Analyse classique des heuristiques de $\mathcal{H}^{\texttt{base}}$

Afin de se familiariser avec l'ensemble $\mathcal{H}^{\texttt{base}}$ d'heuristiques de choix de variable, nous appliquons une classique analyse de celui-ci avec la campagne d'expérimentation Ψ_3 :

$$\left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc=0}}(h_1 > h_2) \;\middle|\; \begin{array}{l} h_1 \in \mathcal{H}^{\texttt{base}} \\ h_2 \in \{\texttt{lex}, \texttt{deg}, \texttt{rand}\} \end{array} \right\} \qquad (\Psi_3)$$

Ψ_3 correspond donc à une campagne présentant les performances des heuristiques de $\mathcal{H}^{\texttt{base}}$ avec divers tie-breakers.

	lex	deg	rand	VBS
abs	#422 (408 484s)	#415 (421 623s)	#413 (425 182s)	#433 (370 132s)
ibs	#384 (482 069s)	#375 (507 395s)	#353 (573 304s)	#406 (434 600s)
dom/ddeg	#407 (427 455s)	#407 (427 241s)	#407 (427 649s)	#407 (427 110s)
dom/wdeg$^{\texttt{chs}}$	#529 (140 219s)	#529 (140 009s)	#529 (140 628s)	#529 (139 533s)
wdeg$^{\texttt{ca.cd}}$	#543 (123 546s)	#540 (123 235s)	#525 (154 278s)	#554 (88 268s)
VBS	#569 (52 355s)	#569 (50 108s)	#564 (56 463s)	

TABLE 3.3 – Comparaison des heuristiques $\mathcal{H}^{\texttt{base}}$, de tie-breakers et de leurs VBS [Ψ_3, $\mathcal{I}_{\texttt{CSP}}$, ↻]

Une première table (3.3) présente les résultats de la campagne Ψ_3 : en ligne nous avons une même heuristique expérimentée avec différents tie-breakers (exprimés en haut de colonne). Sur les extrémités sont représentés les VBS correspondant. Un *meilleur solveur virtuel* (VBS — *Virtual Best Solver*) est un solveur fictif créé à partir des meilleurs résultats d'un ensemble d'heuristiques. Par exemple, le VBS en haut à droite du tableau correspond aux meilleurs résultats obtenus par abs tous tie-breakers compris : pour chaque instance de la campagne, le VBS obtient le temps cpu de la meilleure variante de abs. Les VBS en bout de ligne ou en bas de colonne correspondent aux meilleures résultats des heuristiques de ces lignes ou colonnes. Excepté pour le VBS des lignes correspondants aux heuristiques dom/ddeg et dom/wdeg$^{\texttt{chs}}$, tous les VBS obtiennent de biens meilleurs résultats que la meilleure des heuristiques les composant. En ce qui concerne les tie-breakers, ce sont ceux basés sur lex et deg qui obtiennent les meilleurs résultats. Par la suite, et en accord avec la configuration par défaut de ACE, nous nous concentrons plus en détail sur la sélection d'heuristiques $\mathcal{H}^{\texttt{base}}$ liée au tie-breaker lex.

	$\Delta_{tps} > 0$s	$\Delta_{tps} > 1$s	$\Delta_{tps} > 10$s	$\Delta_{tps} > 100$s	$\Delta_{tps} > 1\,000$s	$\Delta_{tps} = \infty$
dom/wdeg$^{\texttt{chs}}$	**158**	**59**	43	25	13	11
wdeg$^{\texttt{ca.cd}}$	144	**59**	**46**	**32**	**16**	**15**
dom/ddeg	111	49	24	11	2	2
abs	82	27	14	7	6	4
ibs	74	7	5	4	2	2

TABLE 3.4 – Contribution de chacun des solveurs [Ψ_3, $\mathcal{I}_{\texttt{CSP}}$, ↻]

La *table des contributions* 3.4 présente la contribution de chacune des heuristiques de la sélection $\mathcal{H}^{\texttt{base}}$ (uniquement en lien avec lex). La contribution d'une heuristique correspond au nombre de fois où cette heuristique obtient le meilleur résultat en terme de temps de résolution. Une information plus fine serait de comptabiliser le nombre de fois où cette heuristique obtient un résultat dont le temps de résolution est meilleur de x seconde(s) en comparaison avec la seconde meilleure heuristique, noté $\Delta_{tps} > x$s. La table 3.4 présente différents Δ_{tps} allant de 0 à 1 000 secondes. Le $\Delta_{tps} = \infty$ correspond au cas où un et un seul solveur a résolu l'instance (les autres solveurs ont produit un timeout considéré comme un *temps infini*).

À travers la table 3.4, et en accord avec les précédents résultats, nous remarquons à quel point les deux meilleures heuristiques — dom/wdeg$^{\texttt{chs}}$ et wdeg$^{\texttt{ca.cd}}$ — contribuent aux meilleurs résultats peu importe le Δ_{tps}. Néanmoins, nous remarquons aussi que les autres heuristiques contribuent un minimum peu importe le Δ_{tps}. C'est aussi l'occasion d'insister sur le fait qu'aucune heuristique n'est toujours

meilleure que les autres.

3.5 Évaluation en ligne des heuristiques de $\mathcal{H}^{\texttt{base}}$

Dans les chapitres suivants, nous exprimons le besoin de discriminer la capacité des heuristiques à être efficace lors de l'exécution du solveur (en ligne). Pour ce faire, nous proposons dans cette section différentes métriques, que nous appelons fonction de récompense, permettant de noter la qualité actuelle de la recherche. Ces métriques prélèvent à chaque fin de run les informations nécessaires afin de noter, plus précisément, la qualité du run. Une fois ces fonctions de récompense définies, nous proposons une analyse des récompenses des différentes fonctions sur un échantillon d'instances.

3.5.1 Fonctions de récompenses

Ci-après, nous décrivons trois fonctions de récompense proposées dans la littérature dans l'ordre chronologique de leur apparition.

`npts` Une première fonction de récompense considère la *taille des arbres élagués* (`pts` — *Pruned Tree Sizes*) provenant de l'étude [WATTEZ *et al.* 2020]. Intuitivement, cette mesure est donnée par la somme des tailles des sous-arbres trouvant leur racine sur les nœuds ayant donné lieu à une réfutation. Puisque tout nœud de ce type correspond à une impasse ne menant à aucune solution, `pts` représente la capacité du solveur à élaguer rapidement de grandes portions de son espace de recherche. Pour calculer cette mesure, nous ne considérons que la dernière branche produite par le run composée d'affectations et de réfutations depuis laquelle nous extrayons les nogoods. Grâce à la définition et l'utilisation des nogoods dans la formulation qui suit, celle-ci se retrouve légèrement simplifiée comparativement à la formulation déclarée dans [WATTEZ *et al.* 2020]. En termes formels, étant donné un arbre de recherche binaire $\mathcal{T}_t$ généré par le solveur lors du run t, $\Sigma(\mathcal{T}_t)$ représente l'ensemble des nogoods extraits de la dernière branche du run t. Alors,

$$\texttt{pts}(\mathcal{T}_t) = \sum_{\Delta \in \Sigma(\mathcal{T}_t)} \prod_{x \in \mathcal{X} - \texttt{vars}(\Delta)} \left|\texttt{dom}^{\texttt{init}}(x)\right|$$

Sur la base de cette métrique, la récompense associée au run t est donnée par la taille normalisée des arbres élagués :

$$\mathcal{R}_t^{\texttt{npts}} = \frac{\log(\texttt{pts}(\mathcal{T}_t))}{\log\left(\prod_{x \in \mathcal{X}} |\texttt{dom}^{\texttt{init}}(x)|\right)}$$

Une mise à l'échelle logarithmique est nécessaire pour obtenir une récompense plus équilibrée entre 0 et 1. Plus grande est la taille cumulée des arbres élagués, meilleure est la récompense.

`auvr` [CHERIF *et al.* 2020b] propose un moyen de mesurer la performance d'un run afin d'améliorer l'heuristique $\texttt{dom/wdeg}^{\texttt{chs}}$ en utilisant la *moyenne des ratios de variables non-assignées* (`auvr` — *Average of Unassigned Variable Ratios*). Comme pour `npts`, cette récompense estime la capacité d'une heuristique à atteindre rapidement des échecs en calculant la profondeur des conflits qu'elle a provoqués. En d'autres termes, cette mesure donne la moyenne de la profondeur des conflits au cours d'une exécution. Plus la profondeur est faible (c'est-à-dire plus le nombre de variables non-assignées est grand), plus le principe du *fail-first* est respecté et meilleure est la récompense. Soit $\texttt{cft}(\mathcal{T})$ l'ensemble des nœuds conflictuels de l'arbre $\mathcal{T}$:

$$\mathcal{R}_t^{\texttt{auvr}}(i_t) = \frac{1}{|\texttt{cft}(\mathcal{T}_t)|} \sum_{\nu \in \texttt{cft}(\mathcal{T}_t)} \frac{|\texttt{fut}(\nu)|}{|\mathcal{X}|}$$

esb Dans l'étude [PAPARRIZOU & WATTEZ 2020] présentée dans le chapitre 6, le *sous-arbre exploré* (esb — *Explored Sub-Tree*), est donné par le nombre de nœuds visités lors d'un run, divisé par la taille du sous-arbre complet défini sur les variables sélectionnées lors du run. Cette dernière est simplement la taille du produit cartésien des domaines des variables sélectionnées pendant le run. Sont considérées comme variables sélectionnées, celles qui ont été choisies au moins une fois par l'heuristique courante. esb représente l'espace de recherche couvert par le solveur par rapport à l'espace total possible sur les variables sélectionnées. L'intuition est qu'une exploration qui découvre des conflits profondément dans l'arbre de recherche (c'est-à-dire que de nombreuses variables sont assignées) sera pénalisée (en raison du grand dénominateur) par rapport à une exploration qui découvre des conflits dans les branches supérieures.

En termes formels, étant donné un arbre de recherche $\mathcal{T}$ généré lors d'un run, soit $\texttt{vars}(\mathcal{T})$ l'ensemble des variables qui ont été sélectionnées au moins une fois pendant l'exploration de $\mathcal{T}$ et $\texttt{size}(\mathcal{T})$ le nombre de nœuds visités. Alors,

$$\mathcal{R}_t^{\texttt{esb}}(i_t) = \frac{\log(\texttt{size}(\mathcal{T}_t))}{\log\left(\prod_{x \in \texttt{vars}(\mathcal{T}_t)} |\texttt{dom}(x)|\right)}$$

Comme pour npts, une mise à l'échelle logarithmique est nécessaire pour obtenir une meilleure discrimination entre les bras. Plus le ratio/récompense est élevé, plus la performance du run est bonne. Cette récompense a montré les meilleures performances dans le contexte de l'étude [PAPARRIZOU & WATTEZ 2020].

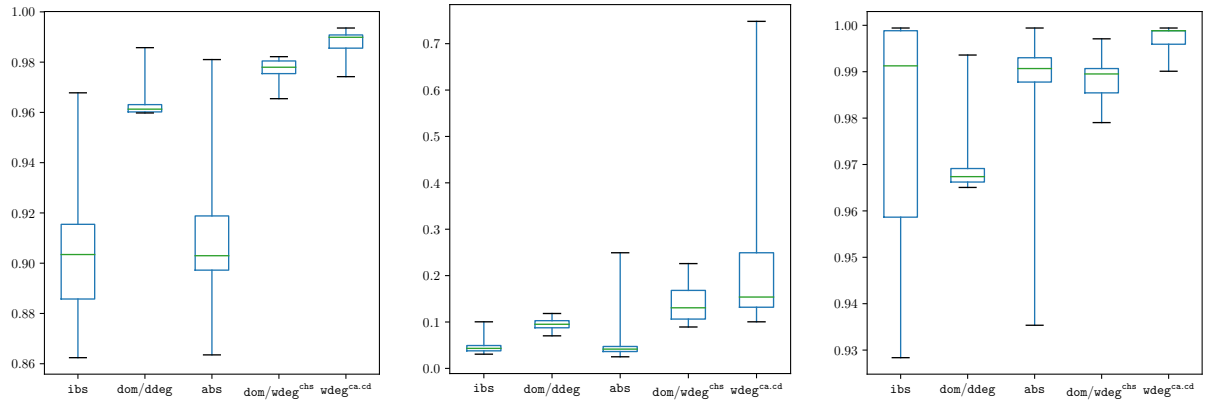
3.5.2 Analyse des récompenses

Afin d'analyser la qualité des fonctions de récompense précédemment déclarées à bien récompenser les heuristiques, nous reprenons la campagne Ψ_3 précédemment déclarée avec uniquement le tie-breaker lex. En plus d'y avoir extrait des données tel que le temps de résolution de chaque paire instance/solveur, nous y avons aussi extrait les récompenses calculées par les trois fonctions de récompense auvr, esb et npts. Ci-après, nous décrivons ces résultats sur une sélection d'instances dont la différence de performance (en terme de temps de résolution) est suffisamment intéressante. À travers les graphiques en boites à moustaches qui suivent, et pour chacune des instances de cet échantillon, nous montrons les r premières récompenses (correspondant aux r premiers runs) de chaque solveur pour chacune des fonctions de récompense. r correspond au nombre de runs nécessaires au meilleur solveur pour résoudre une instance donnée.

Ainsi, à travers les boites à moustaches qui suivent, nous essayons d'observer une corrélation entre la distribution de récompenses (en ligne) et le résultat en temps de résolution (connaissance *a posteriori*). Chaque boite à moustaches décrit la valeur minimale et maximale d'une distribution, le premier et troisième quartiles et la médiane.

Bibd-sc-12-06-05_c18

Une première instance, *Bibd-sc-12-06-05_c18*, est proposée avec des performances très hétérogènes. Nous voyons que l'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$ est la plus rapide à trouver une solution (6.0s et 14 runs),



(a) auvr (b) esb (c) npts

FIGURE 3.2 – Distributions des récompenses pour les heuristiques ibs (*timeout*), dom/ddeg (*timeout*), abs (*timeout*), dom/wdeg^{chs} (50.4s), wdeg$^{ca.cd}$ (6.0s) sur l'instance *Bibd-sc-12-06-05_c18* (14 runs) [Ψ_3, $\mathcal{I}_{CSP}$, ↻]

suivie de dom/wdeg^{chs} (50.4s) prenant presque dix fois plus de temps, et enfin les autres heuristiques ont produit un timeout.

Il n'est pas étonnant d'observer, à travers la représentation des distributions (figure 3.2), que l'heuristique wdeg$^{ca.cd}$ possède une médiane de récompenses plus élevée que pour les autres heuristiques et ce, pour l'ensemble des fonctions de récompense. En seconde position, nous pouvons remarquer, pour auvr et esb, que la distribution de récompenses semblent en accord avec les résultats de dom/wdeg^{chs}. npts semble moins informative pour cette seconde heuristique et n'a peut-être pas encore suffisamment recueilli d'information pour trancher avec les autres heuristiques en timeout.

CarSequencing-85-02_c18

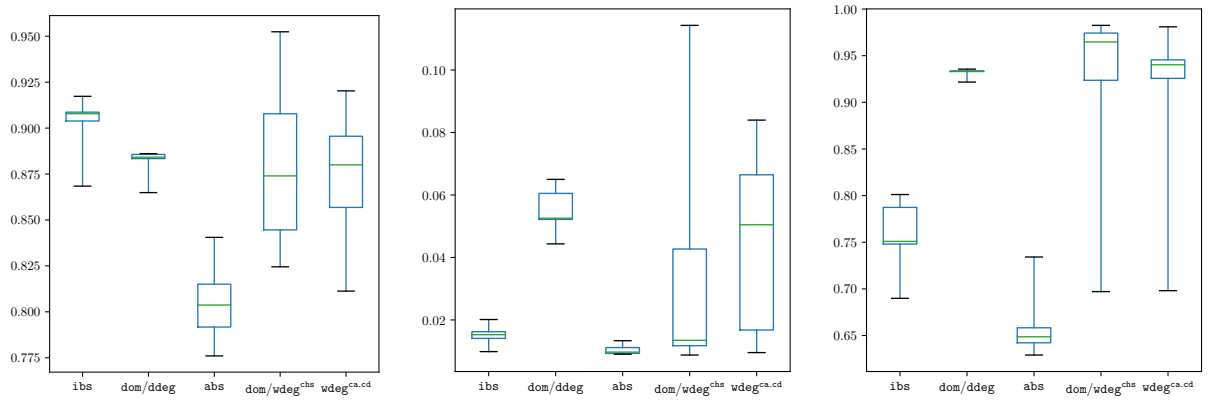
Cette seconde instance, *CarSequencing-85-02_c18*, montre toujours les hautes performances de dom/wdeg^{chs} (11.9s en 17 runs) et wdeg$^{ca.cd}$ (2 116.3s). Cette fois-ci, dom/wdeg^{chs} domine avec un temps de résolution presque 200 fois plus faible que pour wdeg$^{ca.cd}$. Les autres heuristiques sont en timeout.

Si nous nous concentrons sur la médiane, la figure 3.3 montre que seul npts est suffisamment informative pour discriminer la qualité de l'heuristique dom/wdeg^{chs}, puis en second lieu wdeg$^{ca.cd}$. Pour auvr et esb, il semble plus correct de se fier aux valeurs maximales de la distribution, plutôt que la médiane, pour discriminer la qualité de ces deux heuristiques.

CostasArray-16

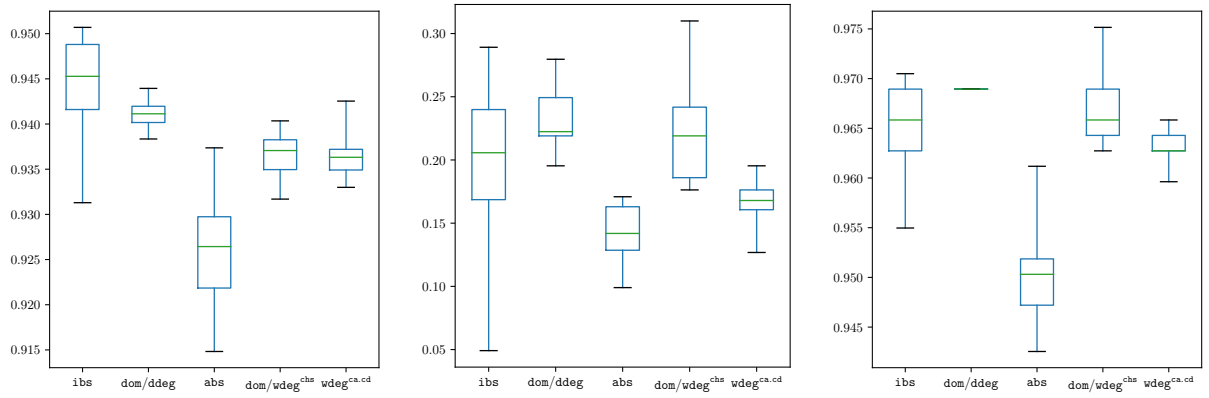
Pour *CostasArray-16*, dom/ddeg est l'heuristique domininant les autres heuristiques en résolvant cette instance en 7.2s et 19 runs, suivie par dom/wdeg^{chs} (9.9s), wdeg$^{ca.cd}$ (26.9s) et plus loin abs (219.6s). ibs a produit un timeout.

Du côté des résultats présentés dans la figure 3.4, les distributions semblent plus controversées. Seules esb et npts montrent une médiane mieux positionnée pour l'heuristique dom/ddeg, puis dom/wdeg^{chs}. Dans le reste des cas, il semblerait que ibs, l'heuristique la moins efficace, ait de meilleurs résultats selon les distributions.



(a) `auvr` (b) `esb` (c) `npts`

FIGURE 3.3 – Distributions des récompenses pour les heuristiques `ibs` (*timeout*), `dom/ddeg` (*timeout*), `abs` (*timeout*), $\texttt{dom/wdeg}^{\texttt{chs}}$ (11.9s), $\texttt{wdeg}^{\texttt{ca.cd}}$ (2 116.3s) sur l'instance *CarSequencing-85-02_c18* (17 runs) [Ψ_3, $\mathcal{I}_{\texttt{CSP}}$, ↻]



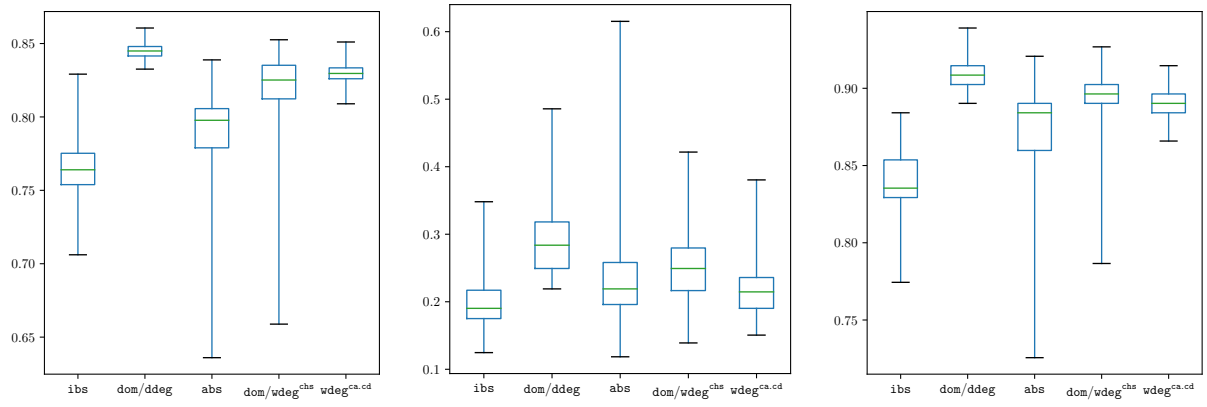
(a) `auvr` (b) `esb` (c) `npts`

FIGURE 3.4 – Distributions des récompenses pour les heuristiques `ibs` (*timeout*), `dom/ddeg` (7.2s), `abs` (219.6s), $\texttt{dom/wdeg}^{\texttt{chs}}$ (9.9s), $\texttt{wdeg}^{\texttt{ca.cd}}$ (26.9s) sur l'instance *CostasArray-16* (19 runs) [Ψ_3, $\mathcal{I}_{\texttt{CSP}}$, ↻]

GracefulGraph-K05-P02_c18

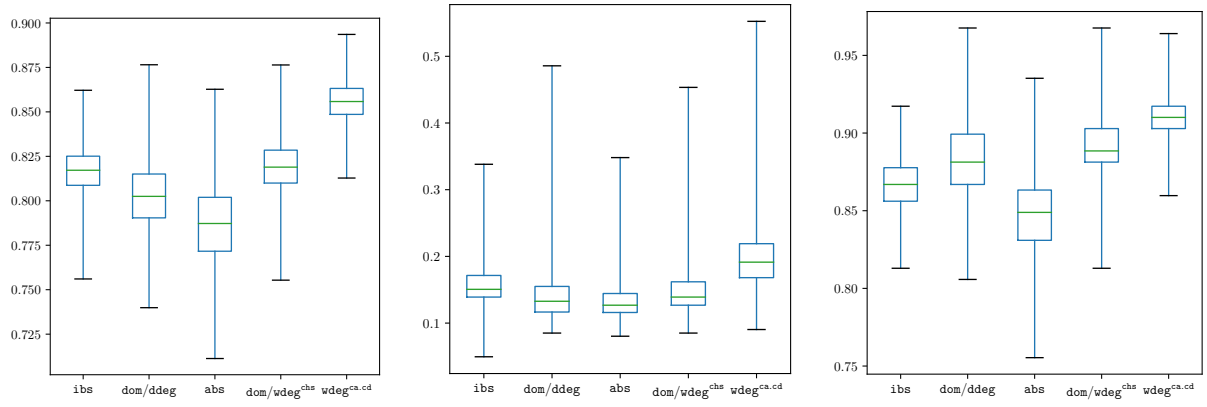
L'instance *GracefulGraph-K05-P02_c18* est rapidement résolue par `dom/ddeg` (9.4s) en 156 runs. Celle-ci est suivie par $\texttt{dom/wdeg}^{\texttt{chs}}$ (23.4s) et `abs` (27.5s) ayant mis deux à trois fois plus de temps. Enfin $\texttt{wdeg}^{\texttt{ca.cd}}$ (99.0s) et `ibs` (333.3s), plus loins, résolvent elles aussi cette instance.

À l'unanimité, les trois fonctions de récompenses (figure 3.5) montrent que `dom/ddeg` est l'heuristique se positionnant la mieux au niveau des distributions suivie par $\texttt{dom/wdeg}^{\texttt{chs}}$. `ibs` trouve la pire place dans ces trois cas.



(a) auvr (b) esb (c) npts

FIGURE 3.5 – Distributions des récompenses pour les heuristiques ibs (333.3s), dom/ddeg (9.4s), abs (27.5s), dom/wdeg^{chs} (23.4s), wdeg$^{ca.cd}$ (99.0s) sur l'instance *GracefulGraph-K05-P02_c18* (156 runs) [Ψ_3, $\mathcal{I}_{CSP}$, ↻]



(a) auvr (b) esb (c) npts

FIGURE 3.6 – Distributions des récompenses pour les heuristiques ibs (*timeout*), dom/ddeg (1 982.1s), abs (*timeout*), dom/wdeg^{chs} (*timeout*), wdeg$^{ca.cd}$ (*timeout*) sur l'instance *frb-59-26-3-mgd_c18* (1 2 923 runs) [Ψ_3, $\mathcal{I}_{CSP}$, ↻]

frb-59-26-3-mgd_c18

Une dernière instance est présentée, *frb-59-26-3-mgd_c18*, où seule l'heuristique dom/ddeg trouve une solution en 1 982.1 secondes et 12 923 runs.

Il est difficile à travers les distributions, présentées dans la figure 3.6, de discriminer cette heuristique. La récompense médiane de dom/ddeg ne semble jamais la mieux positionnée. Les valeurs maximales de cette distribution semblent, tout de même, montrer les bonnes performances de cette heuristique (notamment pour la fonction npts).

D'un point de vue global, il semblerait que ces fonctions de récompense proposent des distributions de récompenses assez fidèles à la réalité et aux résultats *a posteriori* des expériences que nous avons

menées. Bien que ces fonctions semblent informatives, nous remarquons aussi que certaines irrégularités apparaissent quant aux distributions proposées. Au même titre qu'une heuristique, une fonction de récompense n'est pas continuellement la meilleure à utiliser. Par la suite, nous voyons empiriquement si l'usage de ces fonctions de récompense permettent de nous diriger favorablement à travers l'espace de recherche en sélectionnant judicieusement l'heuristique à utiliser.

3.6 Conclusion

À travers ce premier chapitre expérimental, nous évaluons une base d'heuristiques ($\mathcal{H}^{\texttt{base}}$) dont certaines ont été affinées ou adaptées pour les expériences qui suivent. Un protocole expérimental est proposé et va être suivi durant l'ensemble des chapitres suivants. À travers l'analyse approfondie de ces heuristiques, il apparaît qu'une certaine disparité d'efficacité existe, où certaines sont bien meilleures que d'autres, mais aussi que les heuristiques les plus faibles peuvent tout de même contribuer à de meilleurs résultats. L'une des contributions de ce chapitre a été de mettre en avant des fonctions de récompense qui, au cours de l'exécution (en ligne), essayent d'évaluer le plus justement possible les heuristiques afin de se rendre compte de leur efficacité ; cette efficacité qui n'est, habituellement, que décelable *a posteriori* d'expériences. À l'aide de ce nouvel outil capable de discriminer les heuristiques en cours d'exécution, le but va être maintenant de l'associer convenablement avec une politique de sélection d'heuristiques étant capable de converger vers l'heuristique apportant, au solveur, les meilleurs performances.

Chapitre 4

Apprentissage de la Meilleure Heuristique

Sommaire

4.1 Introduction

Dans les applications de résolution sous contraintes, il est souvent demandé à l'utilisateur d'être un expert, en réglant manuellement divers paramètres (ou options de contrôle) du solveur pour améliorer son efficacité sur une instance de problème donnée. Il est clair que la nécessité de ces connaissances de base nuisent à la diffusion de la technologie de programmation par contraintes vers un large public. Afin d'atténuer ce problème, l'idée de la résolution *autonome* est d'ajuster les paramètres du solveur et de traiter efficacement toute instance de problème sans réglage manuel. Notamment, la sélection de l'heuristique de choix de variable peut conduire à des performances radicalement différentes. Une question clé se pose alors : comment trouver la meilleure heuristique de choix de variable pour une instance de problème, étant donné un ensemble d'heuristiques fournies par le système de résolution ? Pour répondre à cette question, nous proposons un framework algorithmique qui combine les bandits multi-bras et les redémarrages de la recherche. Chaque heuristique candidate est considérée comme un bras/action, et le framework apprend à estimer la meilleure heuristique en utilisant un algorithme de bandits multi-bras.

Le mécanisme commun des redémarrages est utilisé pour fournir des retours d'information afin de renforcer l'algorithme de bandit. Sur la base d'une évaluation expérimentale approfondie, nous montrons que ce framework est capable de trouver la meilleure heuristique pour la plupart des problèmes étudiés ; notamment, il surpasse l'état de l'art en termes de temps et d'instances résolues.

4.2 Travaux connexes

La résolution autonome sous contraintes a reçu une attention croissante dans la communauté de la programmation par contraintes, comme en témoigne la diversité des travaux utilisant des techniques d'apprentissage automatique pour régler automatiquement certains composants du solveur. Ces approches peuvent être divisées en deux catégories, en fonction du paradigme d'apprentissage sur lequel elles reposent.

4.2.1 Apprentissage supervisé

La première catégorie utilise l'apprentissage supervisé : étant donné un espace S de configurations pour un composant du solveur, nous partons d'un échantillon d'instances CSP, chacune étant étiquetée avec la meilleure configuration dans S, puis nous apprenons une hypothèse de mise en correspondance des instances CSP dans S. Dans cette catégorie, Balcan et al. [BALCAN *et al.* 2018] ont récemment développé un framework d'apprentissage supervisé pour estimer la meilleure heuristique de choix de variable. [EPSTEIN & PETROVIC 2007] ont synthétisé une *nouvelle* stratégie (par exemple, une combinaison d'algorithme de recherche et d'heuristique) par l'utilisation de l'apprentissage supervisé. Les approches d'apprentissage supervisé pour la résolution de contraintes sont étroitement liées aux *techniques de portfolio*, où S est un ensemble de solveurs ou d'algorithmes candidats. Des portfolios ont été proposés pour les solveurs SAT [XU *et al.* 2008] et CP [O'MAHONY *et al.* 2008, HURLEY *et al.* 2014, AMADINI *et al.* 2016].

4.2.2 Apprentissage par renforcement

Généralités La deuxième catégorie repose sur l'apprentissage par renforcement : en utilisant à nouveau un espace S de configurations, la meilleure configuration dans S est estimée pendant la recherche, en observant un retour sur chaque configuration sélectionnée. Notre framework, ainsi que d'autres approches utilisant un bandit multi-bras pour la résolution de contraintes [BALAFREJ *et al.* 2015, GAGLIOLO & SCHMIDHUBER 2007, LOTH *et al.* 2013, XIA & YAP 2018], appartiennent à cette catégorie. Gagliolo et Schmidhuber [GAGLIOLO & SCHMIDHUBER 2007] appliquent l'algorithme du bandit `EXP3` pour apprendre des stratégies de redémarrage, tandis que Balafrej et al., auteurs de [BALAFREJ *et al.* 2015], utilisent `UCB1` pour sélectionner différents niveaux de propagation pendant la recherche. Dans [LOTH *et al.* 2013], les bandits multi-bras sont exploités pour sélectionner le nœud d'un arbre de recherche *Monte Carlo* à étendre.

Bandit multi-bras et heuristiques Le travail le plus proche de notre approche est sans doute celui de Xia et Yap [XIA & YAP 2018], qui appliquent des algorithmes de bandits multi-bras existants (`UCB1` et `TS`) pour estimer la meilleure heuristique de choix de variable, étant donné un ensemble prédéfini d'heuristiques candidates (bras ou actions). Dans leur framework, un seul arbre de recherche est exploré (il n'y a pas de redémarrage), et l'algorithme du bandit multi-bras est appelé à chaque nœud de l'arbre. Plus précisément, pour chaque nœud non exploré, l'algorithme sélectionne une action et utilise l'heuristique correspondante pour assigner une variable à ce nœud.

Une récompense pour le bras lié à ce nœud est calculée après le parcours complet du sous-arbre enraciné à ce nœud :

$$\mathcal{R}_d^{\texttt{size}} = 1 - \frac{\texttt{size}(\mathcal{T}_{\nu_d})}{\max_{d' \in [d]} \texttt{size}(\mathcal{T}_{\nu_{d'}})}$$

où la fonction `size` retourne le nombre de nœuds visités dans un arbre $\mathcal{T}$. $\mathcal{T}_{\nu_d}$ correspond au sous-arbre enraciné au $d$ème nœud ν_d — d correspond aussi au $d$ème choix du bandit. Cette récompense tente de représenter la qualité du choix d en fonction de la taille de l'arbre produit par ce choix : plus cette valeur est importante, moins bonne est la récompense. Ainsi, la taille de l'arbre enraciné au nœud d est normalisé par la plus grande taille d'arbre connue jusqu'ici (au sein d'un même run). Ce ratio est retranché à 1 afin de correspondre aux futurs bandits qui tenteront de maximiser la valeur des récompenses.

D'un point de vue conceptuel, la principale différence entre cette approche et la nôtre réside dans la caractérisation des essais au cours du processus d'apprentissage séquentiel. Dans cette approche, les essais sont associés aux nœuds inexplorés d'un arbre de recherche, alors que dans notre approche, les essais sont associés aux runs en utilisant un mécanisme de redémarrage.

Afin d'être plus précis sur l'implantation effectuée dans ACE, nous reprenons le principe de présentation d'une heuristique afin d'introduire ce framework. Nous considérons cet ensemble d'heuristiques, manipulées par un bandit en association avec une fonction de récompense, comme une méta-heuristique.

L'heuristique VII présente le framework NOD (*NODe-based framework*) prenant en paramètre un ensemble d'heuristiques $\mathcal{H}$ et une politique de sélection $\mathcal{B}$. Une première méthode, INIT, permet d'initialiser le bandit et chacun de ses bras/heuristiques : un compteur de sélection d est initialisé à 0 ; K est initialisé au nombre d'heuristiques disponibles ; puis le bandit, comme chacune des heuristiques, sont classiquement initialisés. Les méthodes AVANTRUN, APRÈSASSIGNATION et APRÈSCONFLIT ne font que parcourir l'ensemble des bras afin d'appeler ces mêmes méthodes des heuristiques sous-jacentes — en effet, bien qu'une seule heuristique $\mathcal{H}_{i_t}$ ne puisse donner sa décision à chaque essai d (SCORE), le framework NOD met à jour l'ensemble des heuristiques. La méthode AVANTASSIGNATION, appelée avant chaque assignation, est l'occasion pour le bandit de choisir quelle est la prochaine heuristique à utiliser. À cette occasion, le compteur d est incrémenté et le bandit choisit le bras i_d suivant. Qui dit sélection, dit aussi par la suite récompense : la méthode BACKTRACK(d') est appelée à chaque fois que le solveur produit un retour en arrière jusqu'au nœud $\nu_{d'}$ indiquant un parcours exhaustif du sous-arbre enraciné au nœud $\nu_{d'}$. Lors de l'appel à cette méthode, l'heuristique sélectionnée pour ce nœud est récompensée par la fonction de récompense $\mathcal{R}^{\texttt{height}}$.

4.3 RES : un framework basé sur les redémarrages

4.3.1 Contexte et implantation du bandit

Étant donnés un réseau de contraintes $\mathcal{P}$, un ensemble d'heuristiques de choix de variable $\mathcal{H}_1, \cdots, \mathcal{H}_K$ et un mécanisme de redémarrage, quelle est la meilleure heuristique pour résoudre le CSP associé à $\mathcal{P}$? Cette tâche est présentée comme un problème de bandit multi-bras à K bras, où chaque bras est une heuristique candidate. À ce stade, rappelons qu'un problème de bandit est un processus de décision séquentiel dans lequel l'algorithme du bandit interagit avec son environnement. Au cours de chaque tour/essai t, l'algorithme sélectionne un bras i_t dans $[K] = \{1, \cdots, K\}$ et reçoit une récompense $\mathcal{R}_t(i_t)$ pour ce bras. L'objectif est de minimiser le regret cumulé sur T essais, qui est défini comme l'espérance de la différence entre la récompense totale obtenue par le meilleur bras et la récompense totale obtenue par l'algorithme du bandit. Il est important de noter que l'algorithme observe la récompense pour le bras choisi après chaque essai, mais pas pour les autres bras qui auraient pu être

Heuristique VII : NOD($\mathcal{H}$: heuristiques, $\mathcal{B}$: politique de sélection)

Méthode INIT() **:**
  $d \leftarrow 0$
  $K \leftarrow |\mathcal{H}|$
  INIT$_{\mathcal{B}}(K)$
  pour chaque $i \in [K]$ **faire**
    INIT$_{\mathcal{H}_i}()$
  fin

Méthode AVANTRUN() **:**
  pour chaque $i \in [K]$ **faire**
    AVANTRUN$_{\mathcal{H}_i}()$
  fin

Méthode AVANTASSIGNATION() **:**
  $d \leftarrow d + 1$
  $i_d \leftarrow$ SELECT$_{\mathcal{B}}(d)$

Méthode APRÈSASSIGNATION($\mathcal{P}_{av}$, $\mathcal{P}_{ap}$) **:**
  pour chaque $i \in [K]$ **faire**
    APRÈSASSIGNATION$_{\mathcal{H}_i}(\mathcal{P}_{av}, \mathcal{P}_{ap})$
  fin

Méthode APRÈSCONFLIT(c) **:**
  pour chaque $i \in [K]$ **faire**
    APRÈSCONFLIT$_{\mathcal{H}_i}()$
  fin

Méthode BACKTRACK(d') **:**
  MAJ$_{\mathcal{B}}(\mathcal{R}_{d'}^{\texttt{height}})$

Méthode SCORE(x) **:**
  retourner SCORE$_{\mathcal{H}_{i_d}}(x)$

choisis. Par conséquent, la minimisation du regret est obtenue en équilibrant *exploration* (acquisition de nouvelles informations) et l'*exploitation* (utilisation des informations acquises).

Afin d'avancer dans la spécification de notre framework basé sur un bandit, nous devons clarifier les notions d'*essais* et de *récompenses*, et bien sûr, nous devons choisir des politiques de bandit appropriées pour la tâche de sélection de la meilleure heuristique pendant la résolution des contraintes. L'idée clé de ce travail est d'exploiter le mécanisme de redémarrage et ainsi, considérer les essais comme des *runs*. Ainsi, au cours de chaque essai t, l'algorithme du bandit sélectionne un bras i_t, et l'algorithme de résolution est exécuté le temps d'un run en utilisant l'heuristique correspondante $\mathcal{H}_{i_t}$. Lorsque le cutoff `restart`(t) est atteint, l'algorithme du bandit reçoit une récompense $\mathcal{R}_t(i_t)$ qui reflète la performance du solveur. Notamment, ce feedback peut exploiter certaines informations sur l'arbre de recherche binaire $\mathcal{T}_t$ exploré lors du run t (comme proposé dans la section 3.5).

Avec ces notions en main, notre framework RES (*REStart-based framework*) est donné par l'heu-

Heuristique VIII : RES($\mathcal{H}$: heuristiques, $\mathcal{B}$: politique de sélection, $\mathcal{R}$: fonction de récompense)

Méthode INIT() **:**
  $t \leftarrow 0$
  $K \leftarrow |\mathcal{H}|$
  INIT$_{\mathcal{B}}(K)$
  pour chaque $i \in [K]$ **faire**
    INIT$_{\mathcal{H}_i}()$
  fin

Méthode AVANTRUN() **:**
  $t \leftarrow t + 1$
  $i_t \leftarrow$ SELECT$_{\mathcal{B}}(t)$
  AVANTRUN$_{\mathcal{H}_{i_t}}()$

Méthode APRÈSRUN() **:**
  MAJ$_{\mathcal{B}}(\mathcal{R}_t)$

Méthode APRÈSCONFLIT(c) **:**
  APRÈSCONFLIT$_{\mathcal{H}_{i_t}}()$

Méthode APRÈSASSIGNATION($\mathcal{P}_{av}$, $\mathcal{P}_{ap}$) **:**
  APRÈSASSIGNATION$_{\mathcal{H}_{i_t}}(\mathcal{P}_{av}, \mathcal{P}_{ap})$

Méthode SCORE(x) **:**
  retourner SCORE$_{\mathcal{H}_{i_t}}(x)$

ristique VIII. Le framework prend en entrée un ensemble d'heuristiques de choix de variable $\mathcal{H}$, une politique de sélection $\mathcal{B}$ et une fonction de récompense $\mathcal{R}$. La méthode INIT est similaire à celle présentée pour NOD : le bandit et les heuristiques sont initialisées. Ensuite, à chaque début de run (AVANTRUN) un bras i_t est sélectionné, puis la méthode AVANTRUN$_{\mathcal{H}_{i_t}}$ de l'heuristique $\mathcal{H}_{i_t}$ est appelée. Contrairement au précédent framework, chaque heuristique est isolée, c'est-à-dire que seule l'heuristique $\mathcal{H}_{i_t}$, sélectionnée au run t, est mise à jour (il en est de même pour les méthodes APRÈSCONFLIT, APRÈSASSIGNATION et SCORE). Après chaque run (APRÈSRUN) l'heuristique $\mathcal{H}_{i_t}$ est récompensée par la fonction de récompense $\mathcal{R}$.

La partie délicate de ce framework est de définir une fonction de récompense appropriée qui mette en correspondance, à chaque exécution, la performance d'un run à un retour numérique. Là où $\mathcal{R}^{\texttt{height}}$ est adaptée au framework NOD, nous expérimentons par la suite les fonctions de récompenses $\mathcal{R}^{\texttt{auvr}}$, $\mathcal{R}^{\texttt{esb}}$ et $\mathcal{R}^{\texttt{npts}}$, décrites dans la section 3.5, en association avec RES.

4.3.2 Choix de la suite de redémarrages

Comme il a déjà été mentionnée auparavant, la fonction de redémarrage utilisée lors des expérimentations correspond à la suite de Luby. Le problème des suites `exp` et `rexp`, présentées dans la section 1.3.3, est qu'elles affaiblissent la quantité de runs produits lors d'une résolution. En effet, la progression exponentielle est telle que, à partir de $O(n)$ runs (où n est le nombre de variable), la suite propose une

limite de taille d'arbre, $\texttt{exp}(n)$, sensiblement égale au nombre de nœuds nécessaires pour parcourir l'ensemble des combinaisons d'assignations de variables du problème. La suite `rexp` propose à peine plus de possibilités : $O(n^2)$ runs. Des tests préliminaires, non-présentés ici, appuient ces faits et nous montrent que les bandits n'arrivent pas à dépasser les capacités d'une politique naïve aux choix aléatoires ; ce qui est cohérent avec le fait que le bandit n'a pas encore entamé pleinement sa phase d'exploitation avec si peu de runs et que celui-ci explore au même titre que ces politiques aléatoires. C'est pourquoi, en accord avec le besoin considérable de feedbacks dont nécessite une politique de bandit pour apprendre, la suite de Luby semble la plus appropriée pour remplir cette tâche avec sa progression linéaire, proposant ainsi un nombre exponentiel de runs en fonction du nombre de variables : $O(2^n)$. De plus, quand aucune connaissance n'est connue à l'avance pour le choix d'une taille de run fixe et optimale pour une instance donnée, un bon compromis, avec appui théorique, est proposé par la diversité des tailles de runs que propose la suite de Luby [LUBY *et al.* 1993].

4.4 Expérimentations

Afin d'expérimenter les frameworks, nous allons procéder dans un premier temps aux analyses séparées de ceux-ci puis les comparer sur la base des heuristiques de $\mathcal{H}^{\texttt{base}}$. Il est à préciser que les temps indiqués incluent chacun l'ensemble du temps de résolution des solveurs : par exemple, le temps de redémarrage avec l'apprentissage du bandit est inclus dans le temps total.

4.4.1 Analyse du framework NOD

L'étude de `NOD` [XIA & YAP 2018] précise que ce framework a été implémenté dans un solveur sans mécanisme de redémarrage. Dans l'étude [WATTEZ *et al.* 2020], il est montré que ce framework est évidemment plus robuste associé à un mécanisme de redémarrage. Au même titre qu'une résolution classique, avec une seule heuristique, dont les premiers choix sont cruciaux quant à l'efficacité de la recherche, le bandit (tout comme l'ensemble d'heuristiques) n'a encore rien appris de son environnement et ses premiers choix sont donc de pauvres qualités et nécessitent un mécanisme de redémarrage de la recherche.

Ici, nous nous concentrons exclusivement sur la version la plus robuste de `NOD`, étant ainsi dans les mêmes conditions que le framework `RES`.

Pour cette première analyse, nous définissons l'ensemble Ψ_4 de solveurs :

$$\left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc}=0}(\texttt{NOD}(\mathcal{H}^{\texttt{base}}, \mathcal{B})) \;\middle|\; \mathcal{B} \in \{\varepsilon\texttt{-Greedy}, \texttt{UCB1}, \texttt{MOSS}, \texttt{TS}, \texttt{UNI}\} \right\} \qquad (\Psi_4)$$

Au-delà des bandits testés dans l'étude originale du framework `NOD` ([XIA & YAP 2018]), ε`-Greedy`, `MOSS` et `UNI` sont aussi testés. La politique de ces bandits, ne correspondant plus à leur usage initial, a dû être adaptée au framework : `NOD` peut demander au bandit plusieurs bras d'affilés sans les récompenser entre temps. Dans le cas de ε`-Greedy`, `UCB1` et `MOSS`, ces appels consécutifs et sans mise-à-jour provoquent la sélection du même bras consécutivement. `TS`, grâce à son choix aléatoire selon la distribution `beta`, outrepasse cet effet non-désiré. Ce comportement est problématique avec le bandit `EXP3` — convergeant bien trop vite vers le premier bras venu, c'est pourquoi celui-ci n'apparaît pas dans Ψ_4.

La table 4.1 présente les résultats de la campagne sur Ψ_4. Nous remarquons que la politique de choix uniforme est celle détériorant le plus les résultats, permettant ainsi de considérer les bandits comme efficaces par rapport à cette politique témoin. Pas très loin, le solveur correspondant à la politique $\texttt{NOD}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{TS}})$ produit de légères meilleures performances. Au-delà, nous retrouvons les politiques

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
$\texttt{NOD}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}})$	**516**	**92 217**	459	20 173	538
$\texttt{NOD}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{MOSS}})$	510	105 313	459	**15 955**	538
$\texttt{NOD}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}})$	509	104 162	459	18 391	538
$\texttt{NOD}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{TS}})$	494	142 585	459	19 130	538
$\texttt{NOD}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UNI}})$	481	182 766	459	27 615	538

TABLE 4.1 – Comparaison des politiques basées sur les nœuds de recherche [Ψ_4, $\mathcal{I}_{\text{CSP}}$, ↻]

composées des bandits UCB1, MOSS et ε−Greedy. Malgré ces chiffres, aucune politique n'égale les performances de la meilleure des heuristiques de $\mathcal{H}^{\texttt{base}}$ (table 4.5).

4.4.2 Analyse du framework RES

Pour cette deuxième analyse, nous définissons l'ensemble Ψ_5 de solveurs :

$$\left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc=0}}(\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}, \mathcal{R})) \;\middle|\; \begin{array}{l} \mathcal{B} \in \{\varepsilon\texttt{-Greedy}, \texttt{UCB1}, \texttt{MOSS}, \texttt{TS}, \texttt{EXP3}\} \\ \mathcal{R} \in \{\texttt{auvr}, \texttt{esb}, \texttt{npts}\} \end{array} \right\} \qquad (\Psi_5)$$

Pour le framework RES, il est possible de tester l'intégralité des bandits et des fonctions de récompense associées sans modifier leur comportement d'origine. Afin de simplifier la description des résultats, ceux-ci sont divisés en trois tableaux décrivant séparément chacune des fonctions de récompense.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{auvr}})$	**551**	**67 780**	464	9 738	562
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{MOSS}}, \mathcal{R}^{\texttt{auvr}})$	548	73 781	464	9 420	562
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{TS}}, \mathcal{R}^{\texttt{auvr}})$	542	85 716	464	**8 338**	562
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{auvr}})$	535	95 061	464	10 584	562
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{EXP3}}, \mathcal{R}^{\texttt{auvr}})$	476	256 879	464	41 224	562

TABLE 4.2 – Comparaison des politiques basées sur les redémarrages et la fonction de récompense auvr [Ψ_5, $\mathcal{I}_{\text{CSP}}$, ↻]

Dans la table 4.2, excepté la politique utilisant EXP3, toutes les politiques sont proches ou dépassent la meilleure heuristique de l'ensemble $\mathcal{H}^{\texttt{base}}$. $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{auvr}})$ semble être la politique la plus robuste en association avec la fonction de récompense auvr.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{EXP3}}, \mathcal{R}^{\texttt{esb}})$	**551**	**77 437**	527	**20 076**	567
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{TS}}, \mathcal{R}^{\texttt{esb}})$	550	81 688	527	21 446	567
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{esb}})$	550	86 134	527	28 341	567
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{MOSS}}, \mathcal{R}^{\texttt{esb}})$	547	86 687	527	23 498	567
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{esb}})$	541	106 624	527	32 052	567

TABLE 4.3 – Comparaison des politiques basées sur les redémarrages et la fonction de récompense esb [Ψ_5, $\mathcal{I}_{\text{CSP}}$, ↻]

Dans la table 4.3, sans exception cette fois-ci, toutes les politiques sont proches ou dépassent la meilleure heuristique de l'ensemble $\mathcal{H}^{\texttt{base}}$. $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{EXP3}}, \mathcal{R}^{\texttt{esb}})$ semble être la politique la plus robuste en association avec la fonction de récompense `esb`.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{npts}})$	**551**	**69 991**	459	12 708	563
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{TS}}, \mathcal{R}^{\texttt{npts}})$	550	70 998	459	11 804	563
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{MOSS}}, \mathcal{R}^{\texttt{npts}})$	549	77 372	459	14 045	563
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{npts}})$	543	86 781	459	**11 529**	563
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{EXP3}}, \mathcal{R}^{\texttt{npts}})$	464	280 580	459	40 243	563

TABLE 4.4 – Comparaison des politiques basées sur les redémarrages et la fonction de récompense `npts` [Ψ_5, $\mathcal{I}_{\texttt{CSP}}$, ↻]

Enfin, la table 4.4, excepté à nouveau la politique utilisant `EXP3`, l'ensemble des politiques sont proches ou dépassent la meilleure heuristique de l'ensemble $\mathcal{H}^{\texttt{base}}$. $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{npts}})$ semble être la politique la plus robuste en association avec la fonction de récompense `npts`.

Presque à l'unanimité, il semblerait que le bandit `UCB1` soit celui qui s'en sorte le mieux, excepté pour le cas de la fonction de récompense `esb` où `EXP3` est légèrement plus performant. Dans les autres cas, `EXP3` ne réussit pas à converger vers les bonnes heuristiques proposant ainsi des résultats bien plus faibles que la meilleure des heuristiques seule.

Le comportement d'`EXP3` peut s'expliquer par sa capacité à très vite converger. Si la fonction de récompense n'est pas suffisamment informative dès les premiers runs et que celle-ci donne de larges récompenses aux mauvais bras, il est possible qu'`EXP3` ait convergé. Ainsi, nous pourrions remarquer la capacité de la fonction de récompense `esb` à être suffisamment informative dès les premiers runs, contrairement aux autres fonctions. Néanmoins, nous remarquons que chacune de ces fonctions donne la capacité aux autres bandits, et au framework, de proposer de meilleures performances que les heuristiques seules.

4.4.3 Comparaison

Dans cette section, nous souhaitons approfondir notre perception sur les meilleures stratégies soulignées précédemment et en comparaison directe avec les heuristiques seules, leur `VBS` et la stratégie de choix uniforme liée au framework `RES`. Pour cette dernière analyse, nous proposons la comparaison de l'ensemble de solveurs Ψ_6 :

$$\left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc}=0}(h), h \in \left| \begin{array}{c} \mathcal{H}^{\texttt{base}} \cup \\ \{\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UNI}}), \texttt{VBS}(\mathcal{H}^{\texttt{base}})\} \end{array} \right. \right\} \cup \texttt{best}(\Psi_4) \cup \texttt{best}(\Psi_5) \qquad (\Psi_6)$$

La table 4.5 présente la campagne composée des solveurs de l'ensemble Ψ_6 expérimentés sur les instances $\mathcal{I}_{\texttt{CSP}}$. Des moins bons aux meilleurs résultats, nous observons le positionnement des heuristiques `ibs`, `dom/ddeg` et `abs` avec moins de 425 résolutions. Au-delà de 500 résolutions, nous voyons apparaître la politique basée sur les nœuds de recherche, puis les plus récentes heuristiques $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca.cd}}$ englobant la stratégie $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UNI}})$. Cette dernière stratégie, pourtant naïve, arrive à égaler les meilleures heuristiques même s'il reste compliqué de dépasser les capacités de la meilleure d'entre elles. Nous pouvons aussi remarquer une certaine robustesse en terme de temps pour $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UNI}})$. Au-delà de $\texttt{wdeg}^{\texttt{ca.cd}}$, nous remarquons le positionnement des meilleures stratégies de `RES`, sur

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
VBS($\mathcal{H}^{\texttt{base}}$)	**569**	**52 355**	314	**2 830**	575
RES($\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{EXP3}}, \mathcal{R}^{\texttt{esb}}$)	551	96 637	314	4 074	575
RES($\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{npts}}$)	551	98 791	314	4 690	575
RES($\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{auvr}}$)	551	98 980	314	4 350	575
wdeg$^{\texttt{ca.cd}}$	543	123 546	314	5 908	575
RES($\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UNI}}$)	541	113 903	314	4 239	575
dom/wdeg$^{\texttt{chs}}$	529	140 219	314	4 029	575
NOD($\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\varepsilon\text{-Greedy}}$)	516	181 017	314	9 280	575
abs	422	408 484	314	18 831	575
dom/ddeg	407	427 455	314	5 472	575
ibs	384	482 069	314	14 795	575

TABLE 4.5 – Comparaison globale des meilleures stratégies NOD et RES [Ψ_6, $\mathcal{I}_{\text{CSP}}$, ↻]

les trois fonctions de récompense, toutes trois avec 551 résolutions et des temps de résolutions semblables. En première place, nous retrouvons évidemment le solveur virtuel (considéré comme celui ayant les choix optimaux) avec 569 résolutions en un temps de résolution divisé par deux par rapport aux meilleures stratégies réelles.

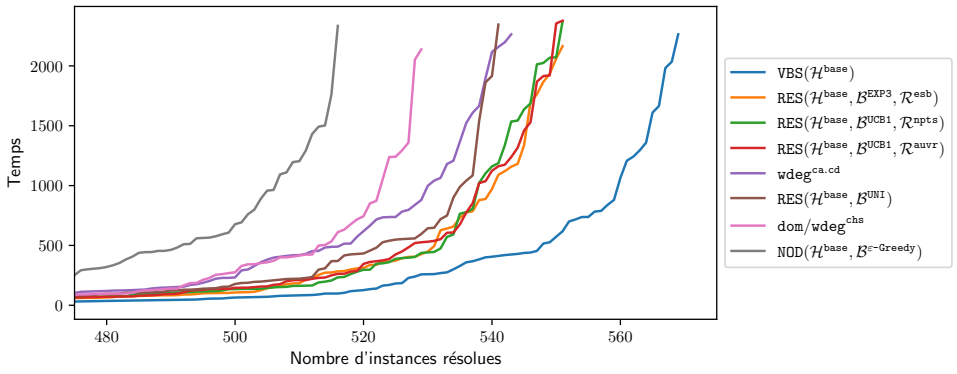


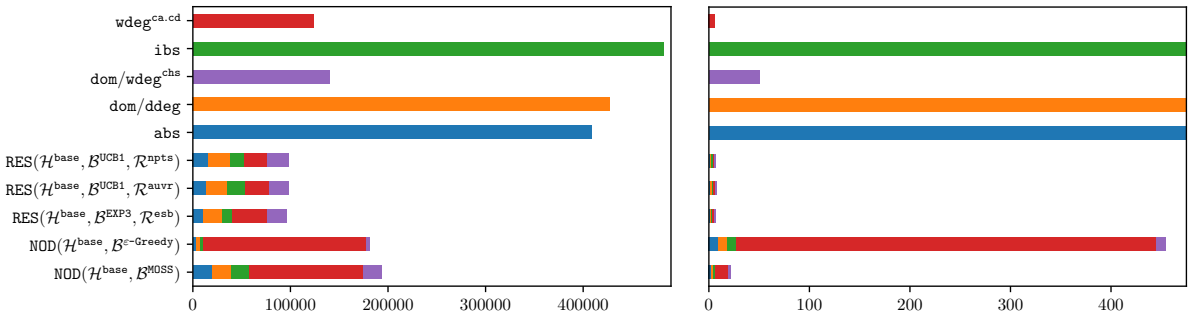
FIGURE 4.1 – Comparaison des meilleures politiques liées aux frameworks RES et NOD, des heuristiques $\mathcal{H}^{\texttt{base}}$ et de leur VBS [Ψ_6, $\mathcal{I}_{\text{CSP}}$, ↻]

Un dernier graphique (figure 4.1) permet d'observer plus précisément les huit meilleures solveurs de la table 4.5. Nous classons ces solveurs en trois groupes distincts. Un premier groupe est composé des deux meilleures heuristiques et de la stratégie uniforme. Il s'agit du minimum requis pour qu'une stratégie soit considérée efficace dans cette analyse (dont la meilleure stratégie basée sur les nœuds de recherche ne fait pas partie). Toujours dans ce groupe, nous pouvons remarquer à travers ce graphique que, sous 2000 secondes de résolution, la stratégie uniforme est celle possédant les meilleures performances. Le dernier groupe contient uniquement VBS ($\mathcal{H}^{\texttt{base}}$). En acheminement vers cette stratégie virtuelle servant de témoin optimal, l'ensemble des trois stratégies basées sur le framework RES sont proches les une des autres.

4.5 Discussions

L'étude proposée dans [XIA & YAP 2018] aborde la même problématique que celle proposée dans ce travail mais utilise un modèle différent. `NOD` appelle un algorithme de bandit multi-bras à chaque nœud de la recherche. La récompense pour le bras utilisé, pour un nœud de la recherche donné, est basée sur la taille du sous-arbre enraciné à ce nœud. `NOD` est, initialement, utilisé avec deux politiques d'exploration : `UCB1` et `TS`. Dans leur étude, Xia et Yap ont montré que `NOD` améliore la recherche et la rend plus robuste que les heuristiques à l'unité. Dans cette section et à travers notre analyse, nous soulevons quelques-uns de leurs biais et, par la même occasion, soulevons aussi des biais de notre modèle.

4.5.1 Critiques communes



(a) Campagne entière

(b) Instance *Bibd-sc-12-06-05_c18*

FIGURE 4.2 – Distribution des sélections d'heuristiques [Ψ_6, $\mathcal{I}_{\texttt{CSP}}$, ↻]

Le modèle Dans la figure 4.2, nous présentons, pour chaque stratégie (heuristiques seules ou stratégies mixtes), la proportion d'appels à chaque heuristique pour l'ensemble de la campagne (figure 4.2a) et pour l'instance *Bibd-sc-12-06-05_c18* (figure 4.2b). Évidemment, une heuristique seule ne s'appelle qu'elle-même. La couleur des heuristiques seules reste la même dans les barres empilées des stratégies à plusieurs heuristiques. La longueur d'une barre correspond au temps nécessaire pour résoudre la ou les instance(s).

En premier lieu, nous remarquons la forte convergence des stratégies `NOD` vers l'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$ négligeant rapidement les autres heuristiques (notamment pour la stratégie basée sur ε`-Greedy`). Bien que ce soit le meilleur choix à produire, cette rapide convergence n'a pas l'air de convenir à la stratégie de sélection quant à son efficacité globale. Il semblerait donc que la fonction de récompense $\mathcal{R}^{\texttt{height}}$ permette une convergence précisément vers $\texttt{wdeg}^{\texttt{ca.cd}}$ (un bon choix) mais que la stratégie globale contraigne cette heuristique à être perturbée par d'autres heuristiques interférant dans son développement. De plus, rien ne certifie qu'il faille constamment converger vers $\texttt{wdeg}^{\texttt{ca.cd}}$. Cela est aussi remarquable à l'échelle de l'instance *Bibd-sc-12-06-05_c18*. Une fois de plus, le fait d'entrelacer les appels au sein d'un même run semble porter préjudice à cette stratégie.

Les stratégies `RES` sont, elles, beaucoup plus prudentes en ce qui concerne leur convergence. Nous observons que les heuristiques $\texttt{wdeg}^{\texttt{ca.cd}}$, $\texttt{dom/wdeg}^{\texttt{chs}}$ et `dom/ddeg` semblent être les trois plus appelées. Les deux premiers choix semblent cohérents quant à l'efficacité de ces heuristiques. Contrairement aux autres heuristiques dont le score des variables est initialisé à 0 en début de recherche et déléguant principale au tie-breaker `lexico`, `dom/ddeg` propose un premier ordonnancement suffisamment efficace pour obtenir de meilleurs récompenses dès les premiers runs. C'est pourquoi, il n'est

pas étonnant de remarquer un important nombre d'appels à cette heuristique.

Néanmoins, cette analyse globale nécessiterait une analyse plus détaillée. Dans le chapitre 5, nous proposons une analyse par famille, reprenant les meilleurs résultats de cette campagne.

Interaction entre bandit et fonction de récompense À travers les expérimentations, nous avons aussi pu remarquer que l'association d'une politique de bandit et d'une fonction de récompense est importante. Un bandit $\mathcal{B}^1$ peut démontrer de bonnes performances avec une fonction de récompense $\mathcal{R}^1$, et tout au contraire, perdre en performance avec une fonction de récompense $\mathcal{R}^2$. A contrario, un bandit $\mathcal{B}^2$ pourrait tout à fait se retrouver avec de meilleures performances avec la fonction de récompense $\mathcal{R}^2$. Ces interactions sont fortement liées à la distribution de récompenses que proposent ces fonctions et l'interprétation qu'en ont les bandits. Certaines caractéristiques telles que l'écart-type (plus ou moins important), la stationnarité ou non de la distribution, *etc.*, font qu'un bandit peut être désorienté alors qu'un autre non.

4.5.2 Critique de NOD

Modèle Un premier biais provient du fait que le bandit sélectionne une nouvelle heuristique à chaque nœud de la recherche. Ainsi, la philosophie derrière une heuristique est brisée car aucune heuristique n'est utilisée pour déterminer un ordre de variables consécutives, mais au contraire, à chaque niveau, une variable est donnée par une heuristique potentiellement différente. Comme au début le bandit explore principalement, NOD ne permet pas qu'une seule et même heuristique, même la meilleure, soit exécutée sur une séquence de nœuds. Il faut attendre que NOD converge vers une heuristique pour que de réelles séquences de choix de variable avec une seule heuristique apparaissent. A contrario, RES laisse une seule et même heuristique se développer pendant tout un run et conserve le comportement complet de l'heuristique le temps de ce run.

Récompense Un autre inconvénient de NOD est lié à la façon dont les récompenses sont calculées. Chaque fois qu'une action est récompensée pour sa performance à un nœud donné, cette récompense dépend du choix des heuristiques à l'intérieur du sous-arbre enraciné à ce nœud et non seulement de la performance pure de l'action sélectionnée.

Un autre biais est lié à la fonction de récompense et sa dépendance à la profondeur actuelle de l'arbre : les nœuds aux niveaux les plus hauts de l'arbre auront toujours un plus grand sous-arbre, et ainsi, de moins bonnes récompenses. Par conséquent, la fonction de récompense ne sera pas en mesure de discriminer les bonnes actions. Considérons le cas où des heuristiques efficaces ont fait de bons choix pendant les premières décisions et une mauvaise heuristique prend les dernières décisions, alors ce sera la mauvaise heuristique qui recevra une bonne récompense bien que non-méritée. En général, toute heuristique en bas de l'arbre recevra une bonne récompense. Si la profondeur était prise en compte, ce biais pourrait certainement être éliminé (par exemple, le modèle de [Balafrej *et al.* 2015] utilise un bandit par niveau au lieu d'un seul pour surmonter cet inconvénient).

Choix du benchmark et actions Le benchmark dans [Xia & Yap 2018] provient de la compétition XCSP'09 et date de neuf ans au moment de l'étude. Si l'on observe leurs résultats expérimentaux, on constate que chaque heuristique, utilisée comme bras, a résolu plus ou moins la même quantité d'instances. Par exemple, l'heuristique dom/ddeg se montre aussi efficace qu'une heuristique de la famille dom/wdeg, alors que la différence est démontrée dans ce manuscrit avec la généricité du benchmark utilisé. À moins que le VBS de cette sélection d'heuristiques montre une grande performance, cela montre sinon que la difficulté des instances est telle que, chaque heuristique offre à peu près les mêmes capacités et donc l'utilité d'un bandit pour les discriminer est faible.

Perturbation des heuristiques Dans l'implantation de NOD, nous remarquons que toute décision prise par une heuristique affecte l'ensemble des heuristiques. Cette pratique n'apporte pas forcément des résultats positifs. En effet, comme une heuristique possède une philosophie propre à elle, si celle-ci se retrouve perturbée par la philosophie d'une autre, l'ordonnancement des variables va s'en retrouver perturbé. Dans le chapitre 6, nous proposons un usage de cette interaction montrant, dans un cadre précis, des effets positifs de cette perturbation.

Actuellement, RES est aussi soumis indirectement à ce genre de perturbation : les heuristiques ne sont pas entièrement indépendantes. En effet, à chaque heuristique sélectionnée, le solveur continue d'apprendre des nogoods et supprime des valeurs incohérentes en fin de run. Cet apprentissage peut aussi perturber le déroulement classique de chacune des heuristiques. Néanmoins, nous remarquons dans cette étude [WATTEZ *et al.* 2020], où la configuration du solveur omettait l'apprentissage des nogoods, que les résultats sont semblables à ceux présentés dans ce manuscrit. Ainsi, une heuristique peut emprunter l'apprentissage (par exemple, la base commune de nogoods) d'une autre sans être pénalisée, et respectant ainsi la configuration par défaut des solveurs apprenants actuels.

4.5.3 Critique de RES

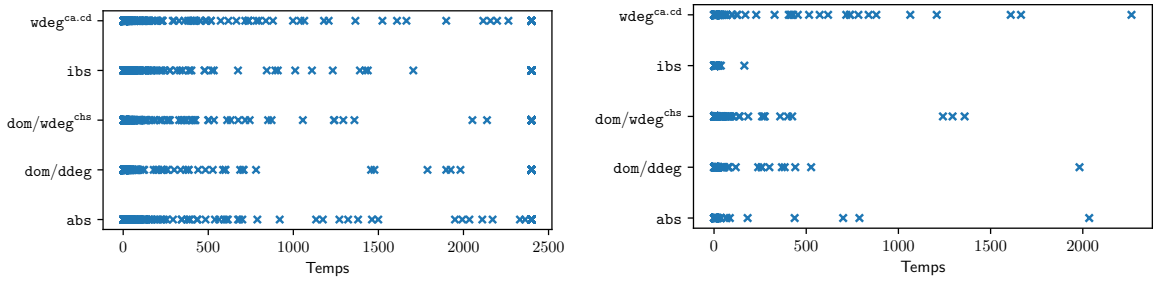
Agrégation des récompenses Une critique que nous pouvons faire au modèle actuel de RES est qu'il ne différencie pas les récompenses attribuées à un bras lors des runs de différentes longueurs. En effet, suivant la suite de Luby, tous les runs ne sont pas égaux. Ainsi, il ne serait pas surprenant de voir que les runs de plus grandes tailles ont, en moyenne, de meilleures récompenses. Or, dans le modèle actuel, aucun mécanisme ne permet de prendre en compte ces différences. Nous voyons, dans le chapitre 5, comment hiérarchiser ces différentes tailles de run.

Stratégie de choix uniforme Dans la figure 4.3, nous visualisons la chronologie de résolution par heuristique. Le graphique de gauche (figure 4.3a) montre exhaustivement l'ensemble des instances résolues par heuristique. Celui de droite (figure 4.3b) montre cette même distribution en supprimant les instances d'une heuristique qui ont été résolues plus rapidement par une autre heuristique. Nous pouvons constater, à travers ces deux graphiques, que la résolution des instances par un solveur suit une distribution logarithmique à travers le temps. Cela explique que même une stratégie à choix uniforme d'heuristiques (c'est-à-dire sans apprentissage par le biais d'une fonction de récompense) résout autant d'instances que la meilleure des heuristiques. De plus, si les instances faciles ne sont pas les mêmes pour chaque heuristique, le bandit uniforme aura de meilleures performances que les heuristiques seules. La raison est que, lorsqu'une instance est facile pour l'heuristique h_1 et difficile pour l'heuristique h_2, la stratégie uniforme la résoudra grâce aux premiers appels à h_1. Si une autre instance est facile pour h_2 et pas pour h_1, il la résoudra avec h_2. C'est pourquoi dans certaines classes de problèmes, une stratégie de choix uniforme peut être plus efficace que n'importe quelle heuristique seule.

Malgré la naïveté de cette stratégie, il est intéressant de retenir que la forte quantité d'exploration semble porter ses fruits quant à l'efficacité proposée. Il est aussi à noter que les heuristiques adaptatives s'améliorent avec le temps et qu'il est important de pratiquer de nombreux essais avec elles afin d'en retirer leurs réels potentiels.

4.6 Conclusion

Dans ce chapitre, nous avons proposé un nouveau framework pour configurer automatiquement l'option de sélection de l'heuristique de choix de variable d'un solveur CSP. Ce framework utilise un algorithme de bandit multi-bras pour apprendre le meilleur choix d'heuristique pour une instance donnée.



(a) Ensemble des résolutions

(b) Résolutions filtrées

FIGURE 4.3 – Distributions des temps de résolution [Ψ_3, $\mathcal{I}_{\texttt{CSP}}$, $\circlearrowleft$]

Nous proposons un modèle de bandit original et facile à intégrer, qui exploite le mécanisme de redémarrage du solveur CSP afin de fournir un feedback à l'algorithme d'apprentissage. Une vaste étude expérimentale a été menée et montre que le framework proposé est plus efficace que l'état de l'art des heuristiques et framework existants. Au chapitre 5, nous essayons de pallier certains phénomènes pouvant détériorer la qualité de la stratégie actuelle en proposant une nouvelle politique de bandit. Entre temps, au chapitre 6, nous tentons de reprendre et revisiter le framework actuel afin d'y incorporer de la perturbation bénéfique pour l'efficacité de la recherche.

Chapitre 5

Organisation d'un Tournoi d'Heuristiques

Sommaire

5.1 Introduction

Dans ce nouveau chapitre, nous prenons en compte les précédentes remarques afin de produire une nouvelle politique de bandit mieux adaptée aux solveurs de contraintes, à leur mécanisme de redémarrage et plus précisément à la suite de Luby. Partant de cet environnement et des nombreuses remarques précédemment extraites, nous proposons une nouvelle politique essayant de pallier les problèmes de sensibilité aux distributions de récompenses numériques, d'agrégation de celles-ci selon l'importance du run et d'exploitation plus importante sur les runs longs. Aussi, nous essayons de prendre en compte les qualités de la stratégie uniforme, notamment dans le fait d'échantillonner les heuristiques plus longuement et uniformément (par exemple, une heuristique adaptative s'améliore avec le temps ; les récompenses initiales ne sont pas forcément représentatives). Enfin, une remarque faite lors de l'analyse des récompenses du chapitre 3 (section 3.5.2) montre que les valeurs maximales des distributions de récompenses sont aussi à prendre en compte.

Partant de ces constats, nous nous proposons d'employer d'autres paradigmes de bandits multi-bras, comme proposés en section 2.4, dans le but de produire un *tournoi* entre les heuristiques de choix de variable. Dans un premier temps, nous décrivons une nouvelle manière d'interpréter la suite de Luby, sans modifier son comportement, afin d'y appliquer cette nouvelle politique de bandit pour laquelle nous fournissons des limites probabilistes de convergence (sous certaines hypothèses stochastiques des heuristiques de choix de variable). L'évaluation expérimentale menée sur les instances $\mathcal{I}_{\texttt{CSP}}$ démontre les avantages de cette nouvelle approche en comparaison des performances du chapitre précédent.

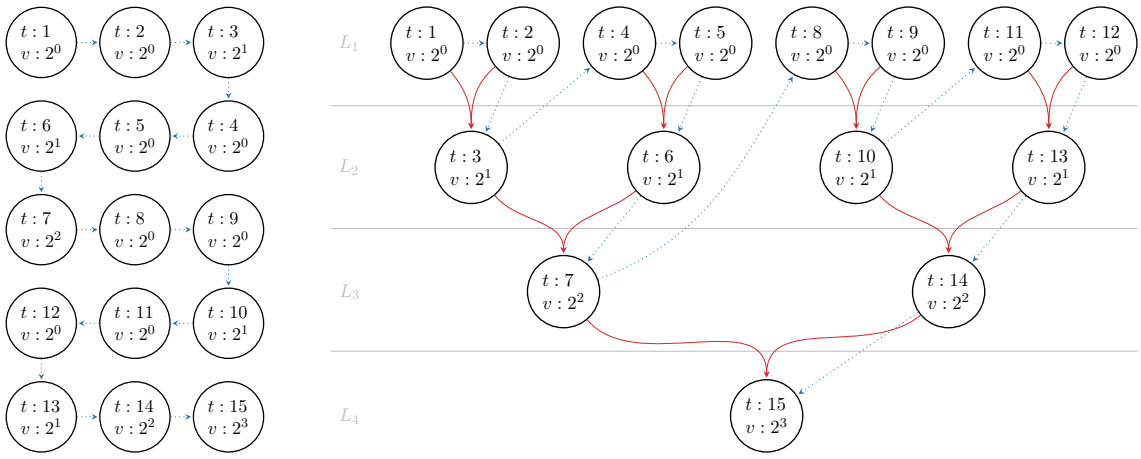
5.2 `ST` : proposition d'une nouvelle politique de bandit

Dans la quête d'une nouvelle politique pour améliorer l'apprentissage de la meilleure heuristique, une première approche pourrait être de partir de la politique uniforme, déjà très performante, et lui ajouter une touche d'apprentissage. Dans cette optique, les bandits à pure exploration (introduits dans la section 2.4.3) proposent une première phase où l'exploration est pure et uniforme, après laquelle une seconde phase s'en suit et exploite les connaissances précédemment acquises (les phases d'exploration et d'exploitation ne se chevauchent pas). Dérivé de ce paradigme, nous pouvons imaginer que le bandit explore sur des runs de petites tailles et exploite sur de plus gros runs : on fixe k' tel que $1 \leq 2^{k'} \leq 2^k$ et $2^{k'}$ est la taille du run à partir duquel le bandit exploite ce qu'il a appris. Le problème reste d'identifier k' étant donné la diversité d'instances proposées.

Afin d'éviter ce problème d'identification et de répondre à un plus large panel de problèmes que nous avons rencontrés, nous nous dirigeons vers une solution utilisant le paradigme duelliste du bandit multi-bras. Dans un premier temps, nous proposons une nouvelle manière d'exploiter la suite de Luby en lui donnant une forme arborescente. Suite à cette première constatation, nous exploitons cet arbre afin de produire des duels entre heuristiques et proposer à chaque gagnant des runs plus importants. Nous finissons par formaliser théoriquement ce nouveau bandit et en proposons une implantation simple.

5.2.1 Une autre vision de la suite de Luby

Une première observation est que, pour toute sous-séquence de la suite de Luby partant de 1 jusqu'à un indice quelconque, s'il existe m occurrences de la valeur 2^k (où $m > 0$ et $k > 0$) dans cette sous-séquence, il existe également $2 \times m$ occurrences de 2^{k-1} apparaissant avant la $m$ième occurrence de 2^k. Une vue séquentielle des premières itérations de la séquence est donnée par la figure 5.1a : t y représente l'indice de la suite de Luby et v la valeur associée. Ainsi, pour $T = 2^4 - 1$ où T représente les T premières valeurs de la suite de Luby, il existe deux fois plus de valeurs 1 que de 2, deux fois plus de 2 que de 4 et deux fois plus de 4 que de 8.



(a) Vue séquentielle

(b) Vue arborescente

FIGURE 5.1 – Différentes visions de la suite de Luby

Partant de cette observation, la suite de Luby peut être observée comme un tournoi sous la forme d'un arbre binaire qui, pour passer d'un niveau à un autre, nécessite le duel de deux occurrences de runs de même valeur 2^k amenant le « gagnant » à profiter du prochain run de valeur 2^{k+1}. Le cheminement

de la suite est toujours le même, mais une sémantique supplémentaire est donnée à celui-ci grâce à cette vision arborescente (figure 5.1).

Plus formellement, la séquence de Luby peut être considérée comme un arbre binaire complet, dont les nœuds ayant les mêmes valeurs de cutoff sont organisés par niveaux. Plus précisément, une séquence de Luby de longueur $T = 2^d - 1$ est un arbre binaire complet de profondeur d, où chaque niveau (*level*) L_i (pour $1 \leq i \leq d$) est formé par les 2^{d-i} nœuds/candidats $\langle C_i^1, C_i^2, \ldots, C_i^{2^{d-i}} \rangle$ ayant la même valeur de cutoff $u \times 2^{i-1}$ (où u est une constante fixée par le solveur). Ainsi, chaque niveau possède la même quantité de budget : d'un niveau L_i à un niveau L_{i+1} le nombre de runs est divisé par deux mais le cutoff associé à chaque run est multiplié par deux. En lien avec cette formalisation, des travaux menés dans le cadre des bandits duellistes sous forme de tournoi montrent des résultats de convergence [CHEN & FRAZIER 2017].

5.2.2 Implantation

À partir de l'interface avec laquelle nous avons décrit l'implantation des précédents bandits, nous réutilisons les trois méthodes INIT, SELECT et MAJ pour décrire cette nouvelle instance de bandit duelliste.

Bandit G : ST(K : actions, d : profondeur de l'arbre, M : échantillonnage, `duel` : opérateur gagnant)

Méthode INIT() **:**
  Initialisation des candidats aux duels du premier niveau (L_1)
  pour chaque $m \in 1..2^{d-2}$ **faire**
    Les candidats d'un même duel sont différents et tirés aléatoirement
    $(C_1^{2m-1}, C_1^{2m}) \leftarrow$ choix aléatoire de $(i, j) \in [K] \times [K] | i \neq j$
  fin

Méthode SELECT(t) **:**
  $l, c \leftarrow$ `index_ST`$(\lceil t/M \rceil)$
  $i_t \leftarrow C_l^c$
  retourner i_t

Méthode MAJ($\mathcal{R}_t$) **:**
  $l, c \leftarrow$ `index_ST`$(\lceil t/M \rceil)$
  ajout de $\mathcal{R}_t(i_t)$ dans R_l^c
  À la fin d'un duel, initialisation du gagnant au niveau supérieur
  si $|R_l^c| = M$ *&* c *est pair* **alors**
    $C_{l+1}^{(c-1)/2} \leftarrow$ `duel`(C_l^{c-1}, C_l^c)
  fin

Le bandit G, nommé ST (*Single Tournament*), prend en paramètres le nombre d'actions K, la profondeur de l'arbre d, la valeur de l'échantillonnage M (c'est-à-dire, le nombre de répétitions d'une même valeur $\texttt{luby}_i$ de la suite) et enfin l'opérateur `duel` désignant le gagnant d'un duel. La méthode INIT génère le premier niveau L_1 de 2^{d-1} candidats appariés en 2^{d-2} duels. Les deux membres d'un duel sont aléatoirement et uniformément tirés depuis $[K] \times [K]$ tout en évitant la sélection de deux mêmes heuristiques. La méthode SELECT sélectionne le candidat suivant en fonction du run t actuel et

de l'échantillonnage appliqué. Une fonction `index_ST` permet de convertir n'importe quel indice t de la suite de Luby en un couple de valeurs (l, c) désignant le niveau et le candidat correspondant dans le structure C. Par cette simple conversion, l'action i_t est assignée au candidat actuel puis retournée. Enfin, la méthode MAJ récupère la récompense $\mathcal{R}_t(i_t)$ et l'ajoute au tableau de récompenses R_l^c du candidat C_l^c. Une dernière condition vérifie s'il s'agit du candidat droit du duel et si celui-ci a complété son échantillonnage ; si tel est le cas, un nouveau candidat $C_{l+1}^{(c-1)/2}$ est initialisé en confrontant les deux membres du duel grâce à l'opérateur `duel`.

La profondeur de l'arbre peut être surestimée en prenant un d suffisamment grand pour être sûr de proposer une valeur de cutoff permettant de parcourir exhaustivement l'espace de recherche. En pratique, d n'a pas besoin d'être connu à l'avance et le premier niveau de l'arbre de recherche peut être construit dynamiquement en fonction de la progression de la suite de Luby.

En ce qui concerne l'opérateur discriminant deux candidats en duel, nous proposons deux instanciations — $\texttt{duel}^{\text{all}}$ et $\texttt{duel}^{\text{max}}$, définies par les deux fonctions suivantes :

$$f^{\text{all}}(C_l^c, C_l^{c'}) = \sum_{(r,r') \in (R_l^c, R_l^{c'})} p(r, r')$$

$$f^{\text{max}}(C_l^c, C_l^{c'}) = p(\max_{r \in R_l^c}(r), \max_{r' \in R_l^{c'}}(r'))$$

où p compare deux récompenses r et r' et retourne un 1 si r est préféré, -1 si r' est préféré, 0 sinon. La fonction f^{all} compare chaque échantillon des deux candidats deux à deux, compte le nombre de victoires de chacun et retourne un entier interprétable par l'algorithme 3. La seconde fonction f^{max} récupère le meilleur échantillon de chacun des deux candidats du duel et retourne un entier discriminant les deux candidats du duel.

Algorithme 3 : $\texttt{duel}^f(C_l^c, C_l^{c'})$

$s \leftarrow f(R_l^c, R_l^{c'})$
si $s > 0$ **alors retourner** C_l^c ;
sinon si $s < 0$ **alors retourner** $C_l^{c'}$;
sinon retourner C_l^c ou $C_l^{c'}$ aléatoirement ;

La fonction f est donc interprétée par l'algorithme 3. Le candidat C_l^c est retourné si le résultat de la fonction est positif, ou le candidat $C_l^{c'}$ si ce dernier est négatif, ou alors en cas d'égalité le choix est aléatoire.

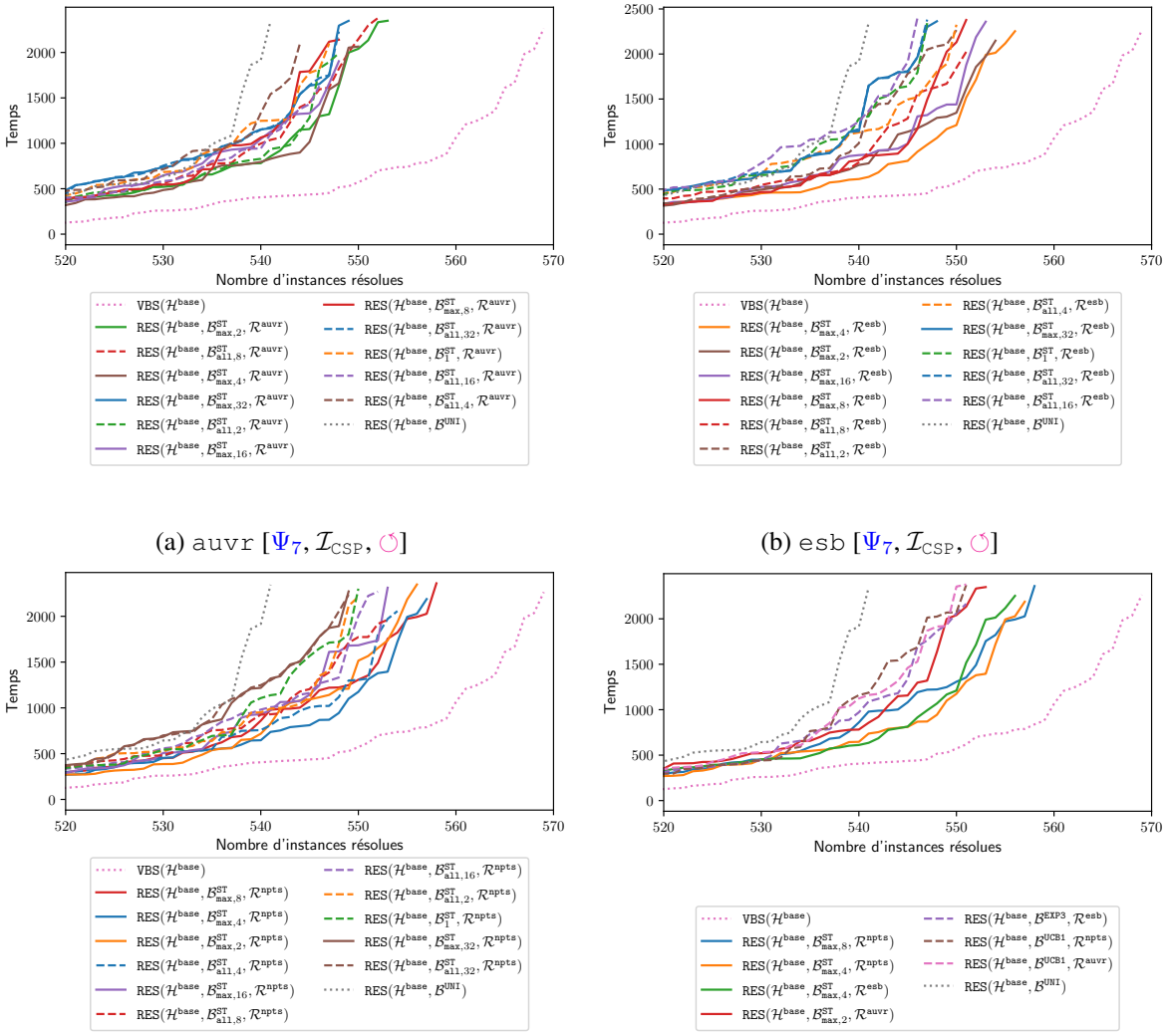
5.3 Expérimentations

Pour les campagnes d'expérimentation qui suivent, nous analysons dans un premier temps le nouveau bandit `ST` avec l'ensemble des fonctions de récompense. Dans un second temps, nous comparons cette campagne aux meilleurs résultats obtenus jusqu'ici. Enfin, nous essayons d'observer plus en détail le comportement de l'ensemble des meilleures politiques.

5.3.1 Analyse et comparaison du bandit `ST`

Pour cette première campagne, nous analysons le bandit `ST` avec différents échantillonnages et les deux opérateurs $\texttt{duel}^{\text{all}}$ et $\texttt{duel}^{\text{max}}$. Soit l'ensemble de solveurs Ψ_7 :

$$\left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc=0}}(\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{duel},M}, \mathcal{R})) \;\middle|\; \begin{array}{l} \texttt{duel} \in \{\texttt{duel}^{\texttt{all}}, \texttt{duel}^{\texttt{max}}\} \\ M \in \{1,2,4,8,16,32\} \\ \mathcal{R} \in \{\texttt{auvr}, \texttt{esb}, \texttt{npts}\} \end{array} \right\} \qquad (\Psi_7)$$



(a) `auvr` [$\Psi_7, \mathcal{I}_{\texttt{CSP}}, \circlearrowleft$]

(b) `esb` [$\Psi_7, \mathcal{I}_{\texttt{CSP}}, \circlearrowleft$]

(c) `npts` [$\Psi_7, \mathcal{I}_{\texttt{CSP}}, \circlearrowleft$]

(d) Comparaison des meilleures politiques [$\Psi_8, \mathcal{I}_{\texttt{CSP}}, \circlearrowleft$]

FIGURE 5.2 – Variation de l'échantillonnage de `ST` pour chaque fonction de récompense et comparaison avec les meilleures politiques [$\Psi_7 \cup \Psi_8, \mathcal{I}_{\texttt{CSP}}, \circlearrowleft$]

Cette première campagne est représentée à travers la figure 5.2. Les trois premiers cactus plots composant cette figure représentent, dans l'ordre, les fonctions `auvr`, `esb` et `npts`. Chaque cactus plot représente les solveurs témoins en pointillés — $\texttt{UNI}(\mathcal{H}^{\texttt{base}})$ et $\texttt{VBS}(\mathcal{H}^{\texttt{base}})$, les politiques `ST` avec la fonction $\texttt{duel}^{\texttt{all}}$ en tirets et avec la fonction $\texttt{duel}^{\texttt{max}}$ en ligne continue. Un même échantillonnage M est représenté par la même couleur. D'un point de vue global sur ces trois graphiques, nous pouvons directement observer que l'ensemble des solveurs implantant la politique `ST` dépasse largement le témoin basé sur le choix uniforme des heuristiques. Sur l'ensemble de ces figures, nous remarquons que l'opérateur $\texttt{duel}^{\texttt{max}}$ a l'air de mieux s'en sortir et aussi, l'échantillonnage basé sur $M = 4$ possède globalement une

plus grande régularité et robustesse.

Afin d'agrémenter la campagne actuelle par rapport à ces nouvelles performances, nous comparons ces résultats avec les meilleurs résultats obtenus lors de la campagne avec les solveurs Ψ_6. Soit Ψ_8 :

$$\texttt{best}(\Psi_6) \cup \texttt{best}(\Psi_7) \tag{Ψ_8}$$

La figure 5.2d représente un cactus plot avec les solveurs témoins (courbes en pointillés) et l'ensemble des plus robustes politiques d'apprentissage de la meilleure heuristique, basées d'une part sur les classiques bandits UCB1 et EXP3 (courbes en tirets), et d'autre part sur le nouveau bandit duelliste ST (courbes en ligne continue). Incontestablement, nous observons que l'ensemble des stratégies basés sur ST dépassent les bandits classiques, et ce, peu importe le temps d'exécution. Les trois stratégies ST utilisant les fonctions de récompense esb et npts sont proches les unes des autres ; néanmoins, nous remarquons que $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$, bien qu'elle soit à une instance de la stratégie ayant résolu le plus d'instances au moment du timeout, est celle qui domine les autres stratégies pendant les 2 000 premières secondes d'exécution. À l'échelle de la course entre les deux solveurs témoins, nous remarquons que, là où les bandits classiques sont à $1/3$ de l'objectif VBS, le solveur implantant $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$ se retrouve au-delà de la mi-chemin vers la stratégie optimale (peu importe le seuil de temps). La figure 5.1 vient appuyer les précédents dires en montrant numériquement les résultats au moment du timeout. La politique favorite ($\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$) réduit le temps de résolution de 15% par rapport à la précédente meilleure politique tout en conservant sensiblement le même temps de résolution commune.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.
$\texttt{VBS}(\mathcal{H}^{\texttt{base}})$	**569**	**49 955**	531	**10 049**	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},8}, \mathcal{R}^{\texttt{npts}})$	558	83 255	531	19 415	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$	557	80 117	531	22 330	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},2}, \mathcal{R}^{\texttt{esb}})$	554	88 631	531	21 595	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},2}, \mathcal{R}^{\texttt{auvr}})$	553	93 291	531	24 514	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{EXP3}}, \mathcal{R}^{\texttt{esb}})$	551	94 237	531	21 353	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{npts}})$	551	96 391	531	19 320	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{auvr}})$	551	96 580	531	22 382	574
$\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{UNI}})$	541	111 503	531	24 404	574

TABLE 5.1 – Comparaison des meilleures politiques de bandit et la politique ST [Ψ_8, $\mathcal{I}_{\texttt{CSP}}$, ↻]

5.3.2 Analyse détaillée

Afin d'y voir plus clair sur le comportement qu'ont la meilleure politique ST, la meilleure politique des bandits classiques, la politique de choix uniforme, les deux meilleures heuristiques et la politique virtuelle et optimale (le VBS), nous proposons la table 5.2 décrivant les familles intéressantes de $\mathcal{I}_{\texttt{CSP}}$.

Chaque colonne de la table 5.2 représente un solveur, chaque ligne représente une famille d'instances mis-à-part les deux dernières lignes correspondant à des statistiques plus globales sur la dominance en terme du nombre d'instances résolus et de temps de résolution. Afin de simplifier l'affichage, certaines familles non-pertinentes, quant aux résultats des solveurs proches les uns des autres, sont retirées : lorsque le nombre de résolutions est similaire entre tous les solveurs et, que le meilleur et le pire solveur ont moins de 50% de différence en temps de résolution. Une cellule représente le nombre de résolutions et le temps de résolution entre parenthèses.

À travers cet ensemble de familles, nous décrivons quelques cas intéressants. Un premier cas, *Bibd*, permet de nous rendre compte que les stratégies multi-heuristiques réussissent à égaler le VBS (le cas le

	VBS($\mathcal{H}^{\texttt{base}}$)	RES($\mathcal{H}^{\texttt{base}}$, $\mathcal{B}^{\texttt{ST}}_{\texttt{max},4}$, $\mathcal{R}^{\texttt{npts}}$)	RES($\mathcal{H}^{\texttt{base}}$, $\mathcal{B}^{\texttt{EXP3}}$, $\mathcal{R}^{\texttt{esb}}$)	RES($\mathcal{H}^{\texttt{base}}$, $\mathcal{B}^{\texttt{UNI}}$)	wdeg$^{\texttt{ca.cd}}$	dom/wdeg$^{\texttt{cbs}}$
Bibd	**#14** (**24 604s**)	**#14** (24 691s)	**#14** (25 294s)	**#14** (27 740s)	#10 (33 818s)	#10 (34 121s)
Cabinet	**#04** (**11s**)	**#04** (16s)	**#04** (16s)	**#04** (15s)	**#04** (16s)	**#04** (22s)
CarSequencing	**#19** (**12 589s**)	#18 (14 457s)	#18 (14 833s)	#17 (17 572s)	**#19** (14 704s)	#15 (21 013s)
ConsecutiveSquarePacking	**#02** (30 415s)	**#02** (29 079s)	#01 (31 230s)	**#02** (**29 030s**)	**#02** (30 418s)	**#02** (30 949s)
CostasArray	**#06** (**96s**)	**#06** (140s)	**#06** (299s)	**#06** (334s)	**#06** (152s)	**#06** (526s)
CoveringArray	**#06** (**101s**)	**#06** (344s)	**#06** (228s)	**#06** (283s)	#03 (7 206s)	**#06** (102s)
cril	**#04** (**20s**)	**#04** (24s)	**#04** (28s)	**#04** (45s)	#03 (2 640s)	**#04** (202s)
Crossword	**#10** (**22 704s**)	**#10** (23 459s)	**#10** (24 759s)	#09 (24 879s)	**#10** (23 724s)	**#10** (23 104s)
Dubois	**#14** (**12 806s**)	#12 (16 623s)	#11 (17 625s)	#11 (19 551s)	#10 (20 544s)	**#14** (**12 806s**)
Eternity	**#07** (19 982s)	**#07** (**19 537s**)	**#07** (21 703s)	#06 (21 667s)	**#07** (19 994s)	#06 (21 691s)
frb	**#06** (**26 781s**)	#05 (28 597s)	#03 (31 219s)	#05 (28 708s)	#03 (31 214s)	#05 (27 507s)
geometric	**#04** (**22s**)	**#04** (29s)	**#04** (29s)	**#04** (45s)	**#04** (64s)	**#04** (34s)
GracefulGraph	**#08** (21 894s)	#07 (24 065s)	**#08** (**21 796s**)	**#08** (23 631s)	**#08** (23 883s)	**#08** (21 933s)
graph	**#07** (**12 120s**)	#04 (16 458s)	#03 (18 694s)	#02 (19 329s)	#06 (15 069s)	#02 (19 331s)
Haystacks	**#03** (31 274s)	**#03** (**31 215s**)	**#03** (31 307s)	**#03** (31 440s)	#02 (33 604s)	**#03** (31 275s)
Kakuro	**#12** (106s)	**#12** (**90s**)	**#12** (114s)	**#12** (207s)	#11 (2 428s)	**#12** (108s)
KnightTour	**#11** (**3 560s**)	#09 (7 384s)	#10 (5 748s)	#09 (7 309s)	**#11** (3 564s)	#05 (16 814s)
LangfordBin	**#11** (**14 609s**)	**#11** (14 970s)	#09 (19 220s)	#10 (16 904s)	#10 (17 271s)	#04 (31 741s)
MagicHexagon	**#07** (**24 413s**)	#06 (26 887s)	#06 (26 532s)	#06 (26 514s)	**#07** (26 098s)	**#07** (24 577s)
MagicSquare	**#22** (**32 145s**)	#21 (35 736s)	#21 (34 156s)	#21 (33 898s)	#20 (36 637s)	#20 (36 162s)
MarketSplit	**#06** (**38s**)	**#06** (67s)	**#06** (108s)	**#06** (112s)	**#06** (419s)	**#06** (130s)
mdd	**#04** (**264s**)	**#04** (378s)	**#04** (445s)	**#04** (818s)	**#04** (1 381s)	**#04** (279s)
MultiKnapsack	**#08** (**23s**)	**#08** (31s)	**#08** (40s)	**#08** (35s)	**#08** (32s)	**#08** (34s)
NumberPartitioning	**#06** (**266s**)	**#06** (1 268s)	**#06** (1 866s)	#05 (3 117s)	**#06** (545s)	#04 (5 237s)
Ortholatin	#01 (12 003s)	**#02** (**9 760s**)	#01 (12 003s)	#01 (12 003s)	#01 (12 003s)	#01 (12 003s)
Pb	**#07** (**30 275s**)	#06 (31 310s)	#06 (31 408s)	#06 (31 792s)	#06 (31 323s)	**#07** (30 317s)
pigeonsPlus	**#06** (**93s**)	**#06** (163s)	**#06** (154s)	**#06** (210s)	**#06** (155s)	**#06** (218s)
Primes	**#06** (39s)	**#06** (**35s**)	**#06** (185s)	**#06** (88s)	**#06** (62s)	**#06** (39s)
QuasiGroup	**#16** (**53 117s**)	**#16** (55 268s)	**#16** (54 075s)	**#16** (53 716s)	#15 (56 077s)	**#16** (53 896s)
QueensKnights	**#06** (**421s**)	**#06** (477s)	**#06** (1 213s)	**#06** (1 072s)	**#06** (537s)	**#06** (422s)
qwh	**#07** (**513s**)	**#07** (**513s**)	**#07** (1 265s)	**#07** (1 603s)	**#07** (810s)	**#07** (2 493s)
rand	**#16** (**622s**)	**#16** (1 285s)	**#16** (1 299s)	**#16** (736s)	**#16** (1 395s)	**#16** (706s)
Rlfap	**#29** (**1 760s**)	**#29** (3 198s)	**#29** (2 061s)	#27 (6 579s)	**#29** (1 765s)	**#29** (3 145s)
SchurrLemma	**#09** (**2 888s**)	**#09** (3 157s)	**#09** (3 241s)	**#09** (3 260s)	#08 (5 449s)	#08 (5 613s)
SocialGolfers	**#15** (**21 739s**)	**#15** (21 866s)	**#15** (21 806s)	**#15** (22 084s)	**#15** (22 524s)	#14 (24 188s)
SportsScheduling	**#06** (26 471s)	**#06** (25 469s)	**#06** (**24 498s**)	#05 (26 625s)	**#06** (26 476s)	#05 (27 097s)
StripPacking	**#05** (**33 854s**)	#04 (36 542s)	#04 (35 545s)	#03 (36 143s)	**#05** (34 005s)	#03 (36 110s)
Subisomorphism	**#15** (**6 521s**)	**#15** (8 059s)	**#15** (8 329s)	#14 (8 968s)	#14 (10 082s)	#14 (8 736s)
SuperSadeh	**#06** (**62s**)	**#06** (232s)	**#06** (733s)	#05 (2 469s)	#05 (2 612s)	#05 (2 459s)
SuperTaillard	#02 (10 666s)	**#03** (**8 611s**)	**#03** (10 151s)	#01 (12 002s)	#02 (10 666s)	#01 (12 002s)
DOMINANCES (#RÉS.)	95.0%	75.0%	70.0%	52.5%	57.5%	57.5%
DOMINANCES (TPS)	77.5%	17.5%	5.0%	2.5%	0.0%	2.5%

TABLE 5.2 – Comparaison par famille [Ψ_8, $\mathcal{I}_{\texttt{CSP}}$, ↻]

plus classique), là où les deux meilleures heuristiques de l'ensemble $\mathcal{H}^{\texttt{base}}$ ont 4 résolutions en moins. De plus, la stratégie ST égale de peu le VBS en terme de temps. Dans certains cas, notamment la meilleure politique ST, le VBS est battu en terme de temps et d'instances résolus : *Ortholatin* et *SuperTaillard*. D'autres cas, peu fréquents, montrent que la stratégie apprenante est moins bonne que la stratégie uni-

forme : *ConsecutiveSquarePacking* et *rand*. Dans cette même fréquence, il existe des cas où les stratégies multi-heuristiques s'en sortent moins bien que les meilleures heuristiques : *CarSequencing* et *Dubois*.

Les deux dernières lignes de ce tableau nous permettent de comprendre le comportement global des différents solveurs de cette étude. L'avant-dernière ligne présente la proportion de dominance (égalités comprises) des différents solveurs en terme du nombre de résolutions. En première place, nous rencontrons inévitablement le `VBS` avec 95% de dominance, suivi des stratégies apprenantes (au-delà de 70%), puis la stratégie uniforme et les deux meilleures heuristiques avec moins de 58% de dominance ; la stratégie uniforme est moins dominante que ces heuristiques unitaires. La dernière ligne de ce tableau représente la dominance en terme de temps de résolution. Le `VBS` domine à 77.5%, là où la stratégie `ST` récupère 17.5% de dominance et où les stratégies restantes oscillent entre 0% et 5%.

5.4 Discussions

5.4.1 Améliorations apportées

Les bandits utilisés avant ce dernier chapitre sont sensibles à l'écart entre les récompenses moyennes. Or, certaines fonctions possèdent des fonctions de récompense dont les moyennes sont proches les une des autres. Une différence dans la moyenne de chacune, même minime, peut représenter un élagage d'arbre de recherche important ; nous pensons notamment aux fonctions de récompense `esb` et `npts` subissant une mise à l'échelle logarithmique afin d'équilibrer la récompense pour la nature exponentielle des tailles d'arbres explorés/élagués. Les bandits basés sur les duels, ne prenant pas en compte cette récompense absolue, ne font que comparer relativement les bras entre eux. Ainsi, cette différence minime entre bras est compensée par cette comparaison relative donnant la priorité à la meilleure récompense même avec un écart minime.

Dans la section 3.5.2, montrant la distribution des récompenses à travers les heuristiques d'origines, nous avions remarqué que les valeurs maximales avaient aussi un impact bénéfique sur le fait qu'une heuristique s'en sorte mieux qu'une autre. Dans cette optique, l'opérateur $\texttt{duel}^{\texttt{max}}$ vient sélectionner l'échantillon, parmi les M échantillons, ayant produit la meilleure récompense. De cette manière, cet opérateur prélève ponctuellement (non pas à l'échelle de toute la durée d'exécution) les pics de grande efficacité d'une heuristique permettant d'améliorer la progression de la résolution.

Un autre comportement des heuristiques adaptatives était à prendre en compte : celles-ci apprennent et s'améliorent au fur et à mesure que la résolution progresse. Ainsi, les récompenses associés à ces heuristiques ont une tendance non-stationnaire en début de recherche. La politique uniforme, n'ayant pas de biais quant à la qualité d'une heuristique, permet d'utiliser ces heuristiques semblant moins bonnes en début de recherche et risquant d'être écartées dans le cas d'une politique de bandit. Ce problème est en partie pris en compte par la nouvelle politique `ST` proposant, au premier niveau de l'arbre correspondant à la suite de Luby, le choix aléatoire et uniforme des heuristiques. Ainsi, aucun biais ne vient entraver une possible amélioration d'heuristique en cours de résolution et aucune n'est écartée lors de ces runs de premier niveau.

Un dernier point concernant l'amélioration des précédents comportement : la taille des runs est maintenant prise en compte. En effet, en hiérarchisant la suite de Luby, les duels ne se font qu'entre deux runs comparables de part leurs limites avant redémarrage. Seules les heuristiques gagnantes peuvent obtenir un run avec un plus grand budget et être, à nouveau, comparées entre elles de façon équitables.

5.4.2 Améliorations à apporter et perspectives

Le modèle actuel, de part sa flexibilité, offre des possibilités d'améliorations et perspectives intéressantes. Dans l'étude actuelle, nous ne testons qu'une faible partie de ce que `ST` peut apporter et nous

pensons que certains points, cités par la suite, pourraient encore améliorer la qualité de la résolution.

Un premier point concerne la constance de l'échantillonnage M. Une idée serait de décrémenter progressivement cette valeur en fonction de la profondeur du tournoi. Théoriquement, si une heuristique est bien meilleure que les autres et pour $K = 2^x$ heuristiques, il suffirait donc de x niveaux pour découvrir cette meilleure heuristique. L'échantillonnage au-delà de ces x niveaux n'a donc plus aucun intérêt car les duels seront entre une même heuristique optimale. Des tests préliminaires tentent de diviser par 2 la valeur de M entre niveaux (partant de différents M initiaux allant de 2 à 32) mais ne montrent pas encore un bénéfice significatif.

Une autre étude préliminaire s'intéresse aux états d'heuristique. Simplement, l'*état d'une heuristique* correspond à une heuristique dont les scores, la caractérisant et permettant l'ordonnancement des variables de l'instance, ont une valeur particulière à un temps donné de l'exécution du solveur. Partant du principe qu'une récompense est associée à la qualité d'une heuristique lors d'un run donné, l'idée est de retenir l'état de l'heuristique au début du run et si celle-ci produit une bonne récompense à la fin du run, l'état de l'heuristique est conservé pour être exploité à nouveau par la suite. Cette idée a d'autant plus d'intérêt que, nous remarquons que les heuristiques récentes, telles que $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca}\cdot\texttt{cd}}$, reformulent très rapidement leur ordonnancement de variable pouvant ainsi produire de très bons résultats comme de moins bons. Ainsi, retenir des états historiques permet de replacer certains de leurs meilleurs ordonnancement sur des runs à plus grand budget.

Dans cette étude, deux opérateurs de duel sont proposés, mais bien d'autres pourraient être intéressants. Par exemple, les opérateurs que nous utilisons jusqu'ici sélectionne la meilleure heuristique. Hors du cadre de la politique de bandit, il serait intéressant de tirer partie de l'apprentissage indépendant de ces deux heuristiques (h_{1a} et h_{1b}) et produire une fusion prenant en compte les performances de chacune. Une proposition de fusion pourrait être de produire un ordonnancement (heuristique statique) h_2 ordonnant les variables du problème ayant produit les meilleurs nogoods construits indépendamment par les deux heuristiques h_{1a} et h_{1b}. L'idée serait alors de sélectionner et réinitialiser la meilleure heuristique h_1 que l'on associerait avec le tie-breaker h_2 composée des précédentes performances agrégées.

En extension de cette précédente idée, la structure arborescente du tournoi d'heuristiques `ST` pourrait accueillir un algorithme génétique. De part sa nature, une heuristique (adaptative, plus précisément) subit des mutations au cours de la résolution (le score associé aux variables change continuellement). Du point de vue de la politique `ST`, l'échantillonnage est une belle occasion pour tester plusieurs mutations d'une même heuristique et sélectionner celle produisant les meilleurs résultats. Comme dit précédemment, l'opérateur de duel pourrait se transformer en un opérateur de fusion permettant de tirer partie de la meilleure mutation de chacun des deux candidats. Le choix de l'opérateur de fusion de deux heuristiques demandent une nouvelle et attentive étude, car comme nous l'avons vu avec le modèle `NOD` du chapitre 4, cela peut être néfaste d'associer plusieurs heuristiques aux philosophies différentes.

Enfin, avec les connaissances expertes accumulées sur les heuristiques proposées (hors du cadre de l'autonomie souhaitée dans ce manuscrit), il peut être intéressant de sélectionner un ensemble d'heuristiques proposant des performances complémentaires afin de proposer un ensemble d'heuristiques plus finement choisi au bandit. Une méthode pour choisir efficacement un ensemble de x heuristiques est d'étudier ce que le `VBS` de ces heuristiques proposent comme performances tout en le bridant en divisant le timeout par le nombre d'heuristiques inclus dans le sous-ensemble. L'idée derrière est que, un grand sous-ensemble permettra une moins grande exploration de chaque bras, là où un trop petit ensemble propose un moins large panel de performances possibles. Cet ensemble peut ainsi être estimé *hors ligne* en étudiant les performances des heuristiques à l'unité et leur `VBS` bridé.

5.5 Conclusion

À travers ce nouveau chapitre, tentant d'améliorer l'usage que fait un solveur de contraintes d'une politique de bandit multi-bras, nous explorons un nouveau paradigme qu'est la politique des bandits duellistes. Accompagnée d'une réinterprétation de la suite de Luby, nous obtenons une suite de redémarrages sous la forme d'un arbre binaire. Cette formulation de la suite de Luby rend favorable l'usage d'un bandit duelliste en produisant un tournoi d'heuristiques nommé `ST` (*Single Tournament*).

La mise en place de cette nouvelle politique permet de nous rendre compte que plusieurs des problèmes que nous nous étions posés auparavant se sont résolus : l'importance de la hiérarchie du budget des runs et de l'équité quant à la comparaison des heuristiques, l'interprétation relative des récompenses, la prise en compte des valeurs maximales dans la distribution de récompenses et la sélection uniforme des heuristiques. Tout en essayant de résoudre ces problèmes, les résultats empiriques montrent un intéressant gain de performance pour cette nouvelle politique, en accord avec la théorie proposée.

Enfin, cette nouvelle interprétation de la suite de redémarrage sous forme de tournoi offre de nombreuses perspectives. Certaines d'entre elles tentent de conserver l'aspect bandit, où certains paramètres peuvent être améliorées (fonction de récompense, échantillonnage, opérateur de duel, *etc.*). Selon d'autres perspectives, nous pourrions y observer un algorithme génétique capable de sélectionner les meilleures mutations d'heuristique et ainsi, de produire une nouvelle heuristique, par principe de fusion, à chaque rencontre (appelée *duel* dans le cadre du bandit).

Chapitre 6

Perturbation d'Heuristique

Sommaire

6.1 Introduction

Ce chapitre entre en extension de la discussion entamée dans la section 4.5.2. Partant des principes de perturbation que le framework NOD impose aux heuristiques, nous essayons dans ce chapitre, de proposer un framework où cette perturbation est constructive et évite les vulnérabilités extraites du framework NOD. Il est à ajouter que la recherche avec retour en arrière est également vulnérable à un comportement instable en raison de la nature combinatoire des problèmes soumis. Au début des années 2000, les différences de comportement ont été étudiées dans le cadre des phénomènes de transition de phase [HOGG *et al.* 1996] et des distributions à queue longue du temps de résolution [GOMES *et al.* 2000]. Ces effets sont réduits par l'introduction de politiques de redémarrage, la randomisation [VAN BEEK 2006] et l'enregistrement de no-goods pendant la recherche. Bien que les méthodes de redémarrage soient maîtrisées, choisir *quand* et *où* ajouter de l'aléatoire au sein d'une heuristique donnée reste un problème ouvert dans la conception des solveurs modernes. Dans ce chapitre, nous présentons plusieurs stratégies de perturbation conceptuellement simples pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrage. La quantité de perturbation est contrôlée et apprise par un framework dirigé par un bandit sous les différentes politiques d'exploration stochastiques et adversarials présentées jusqu'ici, pendant des redémarrages successifs.

6.2 Travaux connexes

6.2.1 Généralités

L'introduction d'une touche de randomisation pour mieux diversifier la recherche locale et complète s'est avérée assez efficace à la fois en SAT et en CP. Une recherche locale stochastique nécessite le bon réglage du paramètre produisant le *bruit* afin d'optimiser ses performances. Ce paramètre détermine la probabilité d'échapper aux minima locaux en effectuant des mouvements non-optimaux. Dans GSAT [SELMAN *et al.* 1992], il est appelé le paramètre de *marche aléatoire* et dans walkSAT [SELMAN *et al.* 1994] comme étant simplement du *bruit*. La *recherche par voisinage* (*Large Neighborhood Search*) utilise la randomisation pour effectuer des sauts dans l'espace de recherche tout en figeant un fragment de la meilleure solution obtenue à un moment donné [PISINGER & ROPKE 2010].

6.2.2 $\texttt{EQ}^{\alpha}$: ajout d'un paramètre d'équivalence

En résolution complète, la première preuve que la diversification peut améliorer la recherche remonte à la recherche de Harvey et Ginsberg [HARVEY & GINSBERG 1995]. Harvey et Ginsberg ont proposé un algorithme de recherche déterministe avec retour en arrière qui diffère du chemin heuristique par un petit nombre de décisions divergeant de celle-ci. Ensuite, Gomes et al. [GOMES *et al.* 1998, GOMES *et al.* 2000] ont montré qu'une forme contrôlée de randomisation affaiblit le phénomène de la *distribution de coûts à queue longue* (c'est-à-dire une probabilité non-négligeable qu'une instance du problème nécessite un temps de résolution exponentiellement plus long que toutes les exécutions précédentes). La randomisation a été appliquée en tant que mécanisme de tie-breaking : si plusieurs choix sont classés de manière égale, on choisit entre eux de manière aléatoire. Cependant, si l'heuristique est suffisamment robuste, elle attribue rarement le score le plus élevé à plus d'un choix. Les auteurs ont donc introduit un paramètre d'*équivalence heuristique* afin d'élargir l'ensemble de choix pour le tie-breaking aléatoire. Dans leur étude, le paramètre d'équivalence est empiriquement fixé à 30%. Par la suite, cette politique est notée $\texttt{EQ}^{\alpha}$ — où α représente le paramètre d'équivalence, et est testée sur l'ensemble des heuristique $\mathcal{H}^{\texttt{base}}$.

6.2.3 `prob` : une stratégie d'échantillonnage

Plus récemment, Grimes et Wallace [GRIMES & WALLACE 2007] ont proposé une façon d'améliorer l'heuristique classique `dom/wdeg` en utilisant une procédure d'échantillonnage avant la procédure de résolution. L'idée principale est de générer les poids des variables avec de nombreuses mais très courtes exécutions (c'est-à-dire des redémarrages) avant la recherche, afin de prendre de meilleures décisions de branchement au début de la recherche. Une fois les poids initialisés, une recherche complète est effectuée pendant laquelle les poids restent gelés ou continuent à se mettre à jour. Cette politique est appelée $\texttt{prob}(h)$, où h est l'heuristique à échantillonner avant d'utiliser entièrement l'heuristique h elle-même. Dans l'idée de l'article original, la politique permettant d'initialiser la pondération d'une heuristique h donnée est fixée à 40 runs de longueur n nœuds de recherche, où n est le nombre de variables, pendant lesquels l'heuristique aléatoire est appelée et modifie les résultats de l'heuristique sélectionnée par l'utilisateur.

6.3 Perturbation et diversification d'heuristique

Dans ce travail, nous utilisons la randomisation pour perturber le processus de sélection des variables. Ces perturbations sont conçues pour maintenir un équilibre contrôlé entre l'exploitation et l'exploration.

Nous présentons des *stratégies de perturbation* conceptuellement simples pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrage. La plupart des stratégies que nous présentons sont adaptatives, ce qui signifie que la quantité de perturbation est apprise au cours des redémarrages successifs. Nous exploitons le mécanisme de redémarrage qui existe dans tous les solveurs modernes pour contrôler la politique de choix aléatoires. À nouveau, nous employons les précédentes politiques de bandit multi-bras pour déterminer à chaque run, si elle appliquera l'heuristique standard, intégrée dans le solveur de contraintes, ou une procédure proposant des choix de branchement aléatoires. Il s'agit d'un problème de décision séquentiel et, en tant que tel, il peut être modélisé comme un problème de bandit multi-bras ; plus précisément, comme un problème à double bras. Dans un même temps, il est aussi proposé une version statique de cette stratégie qui perturbe une heuristique donnée avec une probabilité fixe, dans le but de déterminer si la version adaptative retrouve cette probabilité fixée empiriquement. Enfin, nous proposons une dernière stratégie reprenant le framework `RES` et diversifiant une heuristique h grâce à son tie-breaker.

6.3.1 Perturbation dirigée par un bandit

Comme précédemment indiqué, le processus de résolution de contraintes avec une politique de redémarrage peut être considéré comme une séquence $\langle 1, 2, \cdots, T \rangle$ de *runs*. Au cours de chaque run t, le solveur appelle l'algorithme MAC pour construire un arbre de recherche $\mathcal{T}_t$, dont la taille est déterminée par le cutoff de la politique de redémarrage. Si le solveur n'a accès qu'à une seule heuristique d'ordonnancement des variables, disons h, il exécutera MAC avec h après chaque redémarrage. Cependant, si le solveur est également autorisé à *randomiser* l'ordre de ses variables, il est confronté à un choix fondamental à chaque exécution t : soit appeler MAC avec l'heuristique h afin d'*exploiter* les calculs précédents effectués avec cette heuristique, soit appeler MAC avec un ordre de variables aléatoire $r = \texttt{rand}$ afin d'*explorer* de nouveaux arbres de recherche, et potentiellement de meilleurs ordres de variables. Ici, r est un ordre de variables tiré au hasard selon une distribution uniforme sur l'ensemble des permutations de $\mathcal{X}$. Nous devons souligner que les exécutions aléatoires intermédiaires de r perturbent l'heuristique classique concernée h en mettant à jour ses paramètres, ce qui affecte finalement le comportement/les performances de h. En d'autres termes, les exécutions heuristiques ultérieures ne produiront pas les mêmes ordres que dans le processus de résolution traditionnel, permettant ainsi au solveur de s'attaquer (potentiellement) à des instances que ni h ni r ne résoudraient seuls.

La tâche consistant à incorporer des perturbations dans la résolution de contraintes avec redémarrage peut être considérée comme un problème de bandit *double bras* : au cours de chaque exécution t, nous devons décider si l'algorithme MAC doit être appelé en utilisant h (bras d'exploitation) ou r (bras d'exploration). Une fois que MAC a construit un arbre de recherche $\mathcal{T}_t$, la performance du bras choisi peut être évaluée à l'aide d'une fonction de récompense définie en fonction de $\mathcal{T}_t$. L'objectif global est de trouver une politique faisant correspondre chaque exécution t à une distribution de probabilité sur $\mathcal{H}^{h,r} = \{h, r\}$ de manière à maximiser les récompenses cumulées.

L'heuristique IX présente, de façon générique, le framework accueillant le principe sus-mentionné. Le framework `PER` (*PERturbation policy*) reprend le principe même du framework `RES` (heuristique VIII) à la différence que la décision de n'importe quelle heuristique influence la mise-à-jour de l'ensemble des heuristiques. Ainsi, seules les méthodes AVANTRUN, APRÈSCONFLIT et APRÈSASSIGNATION ont été modifiée afin de parcourir l'ensemble des heuristique $\mathcal{H}$ plutôt que seule l'heuristique $\mathcal{H}_{i_t}$. En accord avec le principe de perturbation d'une simple heuristique h avec l'heuristique de choix aléatoire r, l'appel à cette heuristique peut être décrit comme suit : `PER`($\mathcal{H}^{h,r}$, $\mathcal{B}$, $\mathcal{R}$) où r représente l'heuristique de choix aléatoire `rand`.

Heuristique IX : PER($\mathcal{H}$: heuristiques, $\mathcal{B}$: politique de sélection, $\mathcal{R}$: fonction de récompense)

Méthode INIT() **:**
  $t \leftarrow 0$
  $K \leftarrow |\mathcal{H}|$
  INIT$_{\mathcal{B}}(K)$
  pour chaque $i \in [K]$ **faire**
    INIT$_{\mathcal{H}_i}()$
  fin

Méthode AVANTRUN() **:**
  $t \leftarrow t+1$
  $i_t \leftarrow$ SELECT$_{\mathcal{B}}(t)$
  pour chaque $i \in [K]$ **faire**
    AVANTRUN$_{\mathcal{H}_i}()$
  fin

Méthode APRÈSRUN() **:**
  MAJ$_{\mathcal{B}}(\mathcal{R}_t)$

Méthode APRÈSCONFLIT(c) **:**
  pour chaque $i \in [K]$ **faire**
    APRÈSCONFLIT$_{\mathcal{H}_i}()$
  fin

Méthode APRÈSASSIGNATION($\mathcal{P}_{av}, \mathcal{P}_{ap}$) **:**
  pour chaque $i \in [K]$ **faire**
    APRÈSASSIGNATION$_{\mathcal{H}_i}(\mathcal{P}_{av}, \mathcal{P}_{ap})$
  fin

Méthode SCORE(x) **:**
  retourner SCORE$_{\mathcal{H}_{i_t}}(x)$

6.3.2 Politique statique

Enfin, en plus des politiques de bandit adaptatives, apprenant une distribution sur $\mathcal{H}$ en fonction des récompenses observées, nous considérerons la politique statique (SP — *Static Policy*) suivante : à chaque tour t, SELECT$_{\mathcal{B}^{\text{SP}}}$ choisit h avec une probabilité $(1-\epsilon)$, et r avec une probabilité ϵ. Bien que cette politique présente certaines similitudes avec l'algorithme du bandit ε-Greedy, il existe une différence importante : la distribution sur $\mathcal{H}^{h,r}$ est fixée à l'avance, et par conséquent, SP ne tient pas compte de la moyenne empirique des récompenses observées. En d'autres termes, MAJ$_{\mathcal{B}^{\text{SP}}}$ est une procédure fictive qui renvoie toujours $(1-\epsilon, \epsilon)$. Cette politique stationnaire servira de référence pour les politiques adaptatives dans les expériences.

6.3.3 Diversification de l'heuristique

Une autre manière de penser la perturbation d'une heuristique est de diversifier la configuration de celle-ci. À travers la table 3.3 de la section 3.4.2, nous avions remarqué que la simple diversification du

tie-breaker h_2 associé à une heuristique h_1 — c'est-à-dire $h_1 > h_2$ — rendait le VBS correspondant plus performant. L'efficacité qu'apporte ces tie-breakers réside notamment dans le premier ordonnancement de variables qu'ils proposent et dont l'heuristique h_1 s'inspire afin d'apprendre du réseau de contraintes : il s'agit d'un premier biais très important quant au développement de l'heuristique h_1.

Pour extraire l'efficacité que proposent différents tie-breakers, il suffit d'emprunter le précédent framework, RES, auquel nous allons confier l'heuristique h_1 diversifiée par différent tie-breakers que seront : lex, deg, $\texttt{rand}_1$, $\texttt{rand}_2$ et $\texttt{rand}_3$. $K = 5$ bras sont ici aussi proposés afin de correspondre au nombre de bras des précédents chapitres. Cet ensemble d'heuristiques créé à partir d'une heuristique h est notée $s_5(h)$. $\texttt{rand}_i$ correspond à l'heuristique de choix aléatoire fixe dont la graine de la fonction aléatoire est initialisée avec i.

Une autre manière de diversifier l'heuristique $\texttt{wdeg}^{\texttt{chs}}$ est proposée dans l'étude [CHERIF *et al.* 2020b]. À l'origine, la méthode AVANTRUN$_{\texttt{wdeg}^{\texttt{chs}}}$ initialise à chaque début de run une variable α à $1/10$. L'heuristique est ainsi diversifiée en l'instanciant neuf fois avec différentes valeurs de α allant de $1/10$ à $9/10$ avec un pas de $1/10$. Afin de rester dans les termes de notre étude et le choix de $K = 5$ bras, nous reprenons cette même idée avec un pas de $2/10$ entre chaque instanciation de $\texttt{dom/wdeg}^{\texttt{chs}}$, noté $\alpha_5(h)$.

6.4 Expérimentations

6.4.1 Politiques de perturbation statique et provenant de la littérature

Dans cette première section expérimentale, l'ensemble des témoins sont expérimentés. Afin de ne pas surcharger cette section, le premier ensemble de solveurs Ψ_9 est présenté dans l'annexe B.1.1 :

$$\left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc}=0}(\texttt{PER}(\mathcal{H}^{r,h}, \mathcal{B}^{\texttt{SP}_{x\%}})) \;\middle|\; \begin{array}{l} h \in \mathcal{H}^{\texttt{base}} \\ x \in \{10, 20, \ldots, 70\} \end{array} \right\} \qquad (\Psi_9)$$

Cette campagne teste la perturbation statique sur l'ensemble des heuristiques de $\mathcal{H}^{\texttt{base}}$. Cette perturbation est trouvée linéairement en testant l'ensemble des ratios de perturbation entre 10% et 70% avec un pas de 10%. Au-delà de 70% de perturbation (c'est-à-dire, d'appel à l'heuristique rand venant perturber l'heuristique courante h), nous remarquons une décroissance généralisée de la qualité de résolution pour toutes les heuristiques.

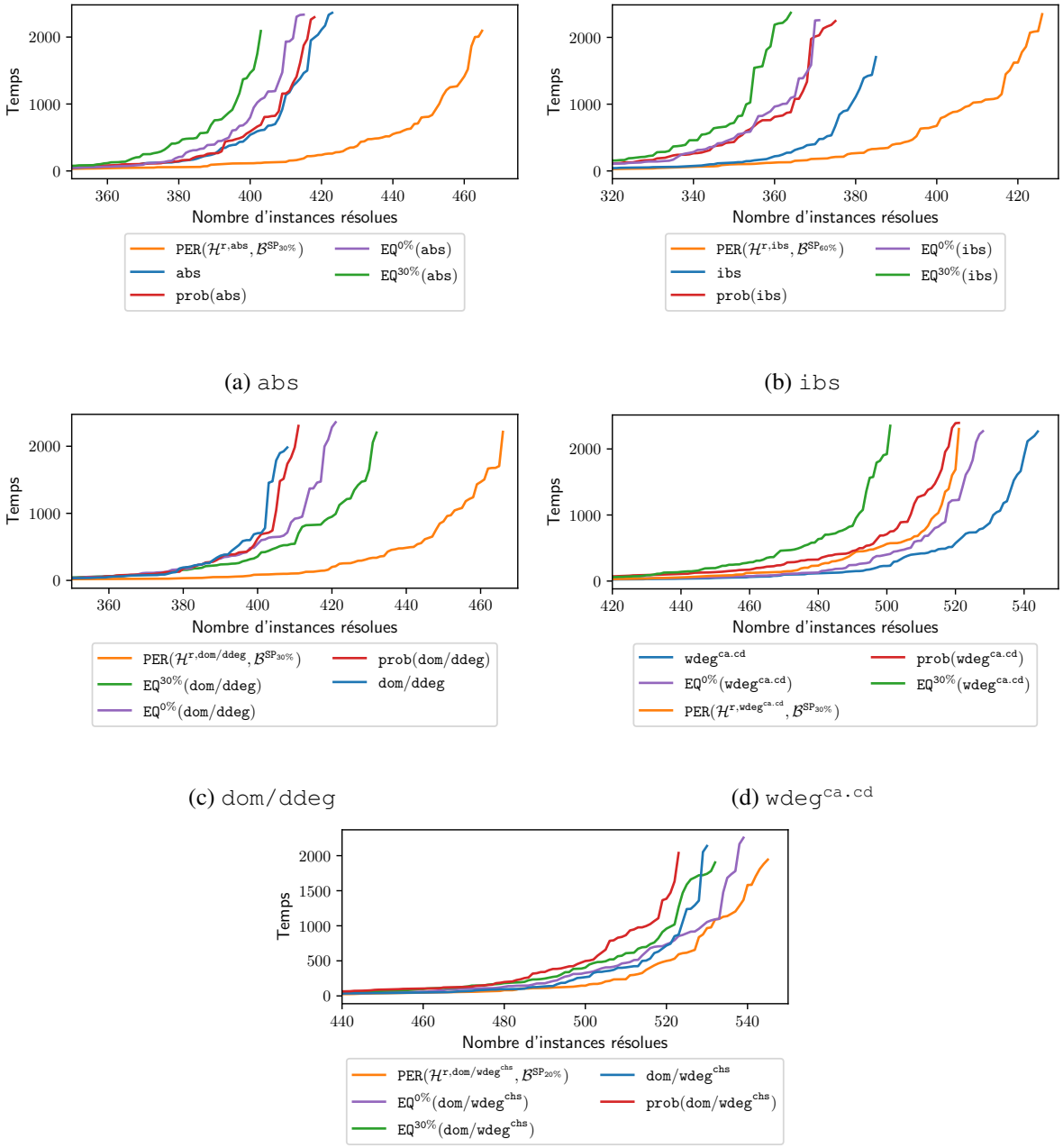
La campagne Ψ_{10} reçoit les meilleurs résultats de la précédente campagne et ajoute par la même occasion les résultats que donnent les politiques d'échantillonnage $\texttt{prob}(h)$ venant s'additionner à l'heuristique h et les politiques implantant le paramètre d'équivalence des scores $\texttt{EQ}^{x\%}(h)$ et délégant à l'heuristique rand en cas d'égalité :

$$\texttt{best}(\Psi_9) \cup \left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc}=0}(\texttt{prob}_h + h) \;\middle|\; h \in \mathcal{H}^{\texttt{base}} \right\} \cup \left\{ \texttt{ACE}^{\texttt{luby}}_{\texttt{lc}=0}(\texttt{equiv}^{x\%}(h)) \;\middle|\; \begin{array}{l} h \in \mathcal{H}^{\texttt{base}} \\ x \in \{0, 30\} \end{array} \right\} \qquad (\Psi_{10})$$

Deux paramètres d'équivalence sont testés : l'un correspond à celui trouvé empiriquement dans l'étude [GOMES *et al.* 1998] — soit 30% d'équivalence, et l'autre est fixé à 0% correspondant ainsi au classique tie-breaking en cas d'égalité stricte.

La figure 6.1 montre les résultats de la campagne sus-mentionnée. Chacune des heuristiques est représentée par un cactus-plot indépendant. Pour un cactus plot donné, l'heuristique originale y est présentée, accompagné des deux versions utilisant le paramètre d'équivalence, de la politique produisant un échantillonnage en premier lieu et enfin la politique dont la quantité statique de randomisation a été trouvée empiriquement.

Dans la globalité des cactus plots de la figure 6.1, nous remarquons que la politique `SP` se démarque largement. Plus en détail, nous observons que `SP` est la seule politique améliorant `abs` et `ibs`. Concernant l'heuristique `dom/ddeg`, toutes les techniques associées à cette heuristique dynamique l'améliorent. L'heuristique $\texttt{dom/wdeg}^{\texttt{chs}}$ est aussi améliorée par les politiques avec le paramètre d'équivalence et notamment la politique `SP`. Enfin, un cas particulier survient avec $\texttt{wdeg}^{\texttt{ca.cd}}$ pour laquelle aucune stratégie n'améliore l'heuristique de base.



(a) `abs` (b) `ibs`

(c) `dom/ddeg` (d) $\texttt{wdeg}^{\texttt{ca.cd}}$

(e) $\texttt{dom/wdeg}^{\texttt{chs}}$

FIGURE 6.1 – Comparaison des politiques témoins pour chaque heuristique de $\mathcal{H}^{\texttt{base}}$ [Ψ_{10}, $\mathcal{I}_{\texttt{CSP}}$, ⟳]

6.4.2 Comparaison avec les frameworks basés sur un bandit

À nouveau pour éviter d'encombrer la section des nombreux résultats demandés par les campagnes suivantes, le résultat des campagnes liées aux ensembles de solveurs Ψ_{11}, Ψ_{12} et Ψ_{13} se retrouvent dans les annexes B.1.2 et B.1.3.

Ψ_{11} représente l'ensemble des solveurs composés du framework de perturbation PER instanciant le couple d'heuristiques $\mathcal{H}^{r,h}$ accompagné de l'un des bandits $\mathcal{B}$ (ε-Greedy, UCB1, MOSS, TS, EXP3, ST) et de l'une des fonctions de récompense $\mathcal{R}$ (auvr, esb, npts) :

$$\left\{ \mathtt{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(\mathtt{PER}(\mathcal{H}^{r,h}, \mathcal{B}, \mathcal{R})) \;\middle|\; \begin{array}{l} h \in \mathcal{H}^{\mathtt{base}} \\ \mathcal{B} \in \{\varepsilon\text{-}\mathtt{Greedy}, \mathtt{UCB1}, \mathtt{MOSS}, \mathtt{TS}, \mathtt{EXP3}, \mathtt{ST}\} \\ \mathcal{R} \in \{\mathtt{auvr}, \mathtt{esb}, \mathtt{npts}\} \end{array} \right\} \qquad (\Psi_{11})$$

Concernant la nouvelle politique de bandit introduite précédemment, seule celle correspondant aux meilleurs résultats précédemment recueillis est prise en compte : $\mathtt{ST}_{\mathtt{max},4}$ simplifiée ici en ST. Le résultat de cette campagne est donné dans l'annexe B.1.2 dont les meilleurs résultats sont reportés et détaillés lors de la campagne Ψ_{14}.

De la même manière, la perturbation d'une heuristique par la diversification de son tie-breaker ou de paramètre interne à l'heuristique (dans le cas de $\mathtt{dom/wdeg}^{\mathtt{chs}}$) est donnée par les ensembles Ψ_{12} et Ψ_{13} :

$$\left\{ \mathtt{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(\mathtt{RES}(\mathcal{H}^{s_5(h)}, \mathcal{B}, \mathcal{R})) \;\middle|\; \begin{array}{l} h \in \mathcal{H}^{\mathtt{base}} \\ \mathcal{B} \in \{\varepsilon\text{-}\mathtt{Greedy}, \mathtt{UCB1}, \mathtt{MOSS}, \mathtt{TS}, \mathtt{EXP3}, \mathtt{ST}\} \\ \mathcal{R} \in \{\mathtt{auvr}, \mathtt{esb}, \mathtt{npts}\} \end{array} \right\} \qquad (\Psi_{12})$$

$$\left\{ \mathtt{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(\mathtt{RES}(\mathcal{H}^{\alpha_5(\mathtt{dom/wdeg}^{\mathtt{chs}})}, \mathcal{B}, \mathcal{R})) \;\middle|\; \begin{array}{l} \mathcal{B} \in \{\varepsilon\text{-}\mathtt{Greedy}, \mathtt{UCB1}, \mathtt{MOSS}, \mathtt{TS}, \mathtt{EXP3}, \mathtt{ST}\} \\ \mathcal{R} \in \{\mathtt{auvr}, \mathtt{esb}, \mathtt{npts}\} \end{array} \right\} \qquad (\Psi_{13})$$

Le résultat des campagnes correspondantes sont affichés dans l'annexe B.1.3 et les meilleurs résultats sont aussi reportés dans la campagne avec les solveurs de l'ensemble Ψ_{14} :

$$\mathtt{best}(\Psi_{10}) \cup \mathtt{best}(\Psi_{11}) \cup \mathtt{best}(\Psi_{12}) \cup \mathtt{best}(\Psi_{13}) \qquad (\Psi_{14})$$

L'un des défis de la campagne qui suit, pour les politiques PER utilisant un bandit pour estimer la quantité nécessaire de perturbation, est d'apprendre et retrouver la quantité optimale de perturbation calculée empiriquement, voire d'améliorer ces performances.

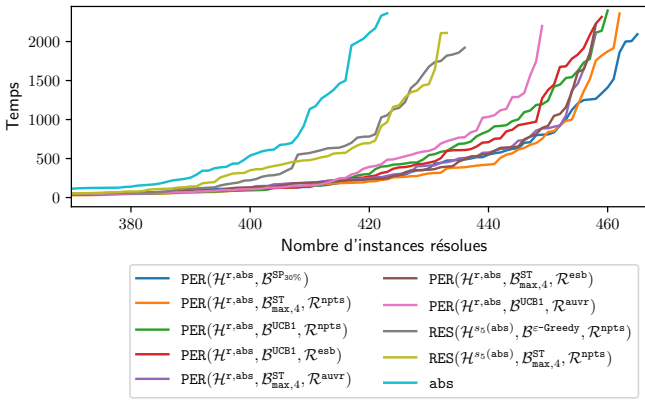
Les figures qui suivent sont une composition de tableau de statistiques, de cactus plot et de scatter plot. Chacune de ces compositions décrit la campagne Ψ_{14} pour une famille d'heuristiques donnée. Le tableau et le cactus plot donnent des résultats généraux, là où le scatter plot va s'intéresser à la comparaison de deux solveurs : l'heuristique d'origine et la politique l'étendant la plus intéressante. De manière globale, nous remarquons que les politiques basées sur ST et la fonction de récompense npts ont une tendance à dominer les autres. C'est pourquoi, même si elles ne sont pas toujours optimales localement pour une heuristique, nous présentons celle-ci comme politique optimale face à l'heuristique d'origine dans le scatter plot. La comparaison au sein d'un scatter plot se fait entre deux solveurs pour lesquels nous récupérons l'ensemble des temps de résolution de chacune des instances et les plaçons sur ce graphique : un point de coordonnées (x, y) correspond à une instance dont le solveur s_1 a mis x secondes pour la résoudre et le solveur s_2 y secondes. Ainsi, la comparaison se fait en fonction de

la diagonale du graphique : plus un point est écarté de cette diagonale, plus l'un des deux solveurs est efficace à la résoudre. Afin d'ajouter une information pouvant être intéressante au sein des scatter plots, les points sont colorés en fonction de la réponse du solveur : *SAT* ou *UNSAT*.

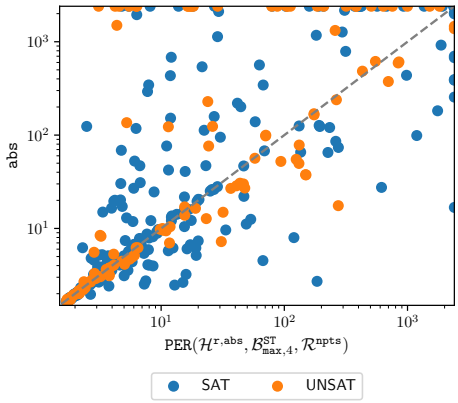
abs Nous présentons tout d'abord la figure 6.2 représentant les performances de l'heuristique `abs` et ses variantes perturbées. Le cactus plot présente les améliorations de l'heuristique `abs`. Nous voyons que l'ensemble des politiques améliore l'heuristique d'origine. Concernant les politiques utilisant un bandit, il semblerait qu'`abs` soit plus sensible à la perturbation provoquée par l'heuristique `rand` plutôt que par la diversification de son tie-breaker. La politique de perturbation basée sur un bandit ayant les meilleures performances est $\texttt{PER}(\mathcal{H}^{r,\texttt{abs}}, \mathcal{B}^{\texttt{ST}}, \mathcal{R}^{\texttt{npts}})$ et talonne la meilleure politique de perturbation statique fixée à 30%.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.	%PERT.
$\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{SP}_{30\%}})$	**465**	**121 084**	392	15 208	497	30.0
$\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$	462	121 281	392	**9 877**	497	32.6
$\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{npts}})$	460	136 198	392	14 348	497	33.4
$\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{esb}})$	459	135 794	392	17 157	497	29.0
$\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{auvr}})$	458	128 990	392	12 701	497	25.0
$\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{esb}})$	458	129 076	392	9 987	497	21.1
$\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{auvr}})$	449	149 974	392	11 942	497	26.5
$\texttt{RES}(\mathcal{H}^{s_5(\texttt{abs})}, \mathcal{B}^{\varepsilon\text{-Greedy}}, \mathcal{R}^{\texttt{npts}})$	436	186 206	392	12 148	497	–
$\texttt{RES}(\mathcal{H}^{s_5(\texttt{abs})}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$	433	188 523	392	12 891	497	–
`abs`	423	218 888	392	14 606	497	–

(a) Tableau de statistiques [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]



(b) Cactus plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]



(c) Scatter plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

FIGURE 6.2 – Analyse et comparaisons de l'heuristique `abs` et ses versions perturbées [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

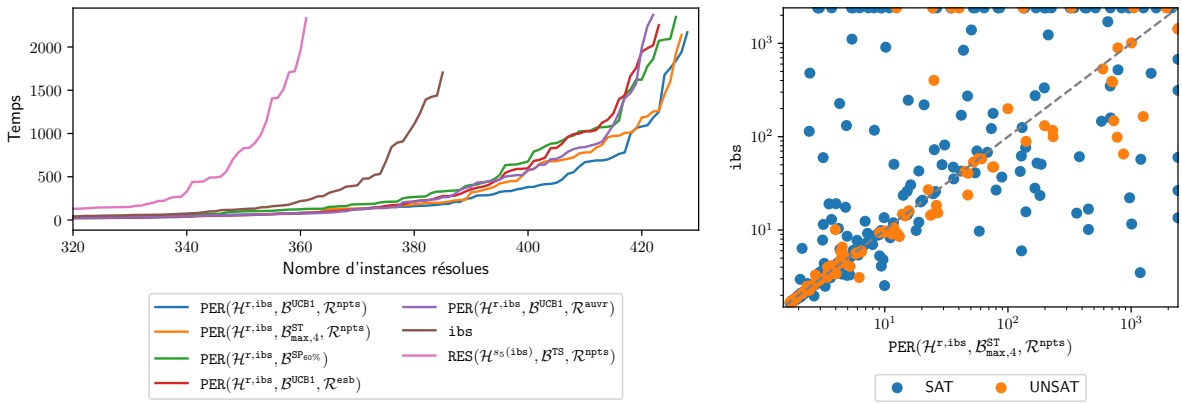
Le tableau de statistiques nous en dit plus par rapport à ces résultats. Nous apprenons que la politique fixe optimale, bien qu'ayant trois instances résolues supplémentaires par rapport à la politique `ST`, celle-ci possède un temps de résolution similaire et cela se remarque d'autant plus sur le temps de résolution des instances communément résolues : `ST` prend 35% moins temps à les résoudre que la politique statique optimale. Outre ces statistiques concernant le temps, la dernière colonne renseigne la quantité de perturbation sur l'ensemble de la campagne. L'ensemble des bandits perturbant avec l'heuristique aléatoire semble trouver, approximativement, la même valeur que celle calculée empiriquement.

Enfin, le scatter plot nous renseigne sur la distribution du temps de résolution de l'heuristique originale `abs` et de sa version perturbée par le bandit `ST` et la fonction de récompense `npts`. À travers ce graphique, nous remarquons rapidement que la version perturbée est celle récupérant la majeure partie des instances, en terme de temps tout comme en terme de résolutions (là où l'autre heuristique a produit un timeout, visible par la position extrême du point). Quelques instances satisfaisables n'ont pas pu être récupérées depuis l'heuristique originale, mais les instances *SAT* sont connues pour être trouvées, en partie, grâce à l'aléa de la recherche. Cette perte est compensée par le fait de trouver d'autres instances satisfaisables grâce à ce même aléa. Les instances insatisfaisables demandent, elles, une recherche complète de l'espace de recherche et donc une efficacité accrue de l'heuristique. Ici, nous remarquons que seules deux instances n'ont pas pu être récupérées car celles-ci prenaient plus de mille secondes pour être résolues par l'heuristique originale. En contre-partie, la version perturbée a résolu une grande quantité d'instances insatisfaisables supplémentaires dont l'heuristique d'origine n'a pas réussi à explorer entièrement l'espace de recherche.

`ibs` Le cactus plot de la figure 6.3 montre lui aussi un grand gap dans l'amélioration de l'heuristique d'origine. Mis-à-part la politique basée sur la diversification de l'heuristique, qui ne semble absolument pas porter ses fruits, les autres stratégies ont permis de résoudre une quarantaine d'instances supplémentaires. Dans ce top, nous remarquons que les deux politiques, basées sur les bandits `UCB1` et `ST` s'entremêlent pour la première place. Ces deux bandits semblent légèrement plus performants que la politique dont la valeur optimale a été fixée à 60% d'exploration.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.	%PERT.
$\mathrm{PER}(\mathcal{H}^{\mathtt{r,ibs}}, \mathcal{B}^{\mathtt{UCB1}}, \mathcal{R}^{\mathtt{npts}})$	**428**	**82 711**	337	**12 983**	447	38.8
$\mathrm{PER}(\mathcal{H}^{\mathtt{r,ibs}}, \mathcal{B}^{\mathtt{ST}}_{\mathtt{max},4}, \mathcal{R}^{\mathtt{npts}})$	427	87 971	337	16 755	447	40.0
$\mathrm{PER}(\mathcal{H}^{\mathtt{r,ibs}}, \mathcal{B}^{\mathrm{SP}_{60\%}})$	426	101 638	337	18 360	447	60.0
$\mathrm{PER}(\mathcal{H}^{\mathtt{r,ibs}}, \mathcal{B}^{\mathtt{UCB1}}, \mathcal{R}^{\mathtt{esb}})$	423	99 986	337	16 742	447	29.1
$\mathrm{PER}(\mathcal{H}^{\mathtt{r,ibs}}, \mathcal{B}^{\mathtt{UCB1}}, \mathcal{R}^{\mathtt{auvr}})$	422	98 044	337	19 883	447	26.1
`ibs`	385	172 473	337	15 221	447	–
$\mathrm{RES}(\mathcal{H}^{s_5(\mathtt{ibs})}, \mathcal{B}^{\mathtt{TS}}, \mathcal{R}^{\mathtt{npts}})$	361	235 429	337	15 269	447	–

(a) Tableau de statistiques [Ψ_{14}, $\mathcal{I}_{\mathrm{CSP}}$, ⟳]



(b) Cactus plot [Ψ_{14}, $\mathcal{I}_{\mathrm{CSP}}$, ⟳]

(c) Scatter plot [Ψ_{14}, $\mathcal{I}_{\mathrm{CSP}}$, ⟳]

FIGURE 6.3 – Analyse et comparaisons de l'heuristique `ibs` et ses versions perturbées [Ψ_{14}, $\mathcal{I}_{\mathrm{CSP}}$, ⟳]

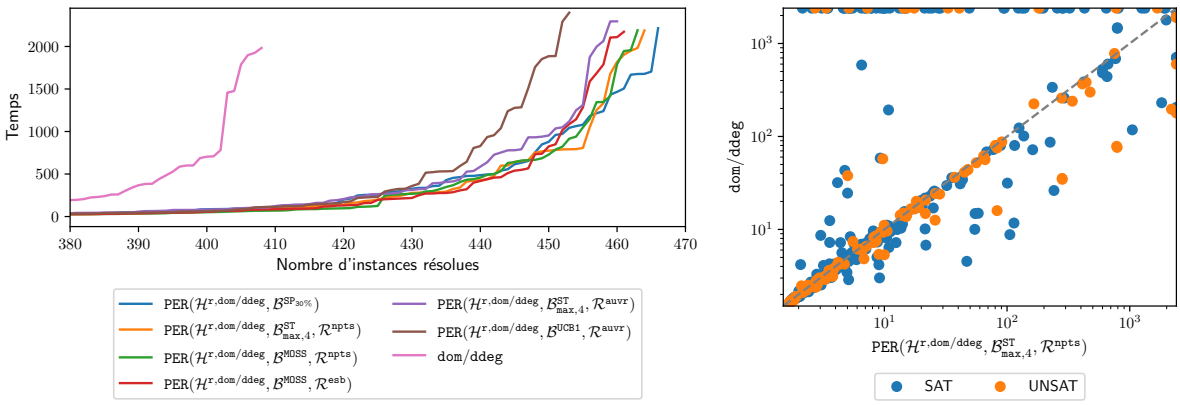
Le tableau de statistiques nous indique que ces deux meilleures stratégies semblent avoir trouvé un taux plus favorable de perturbation aux environs de 40%. Il n'est pas si étonnant de voir ce chiffre, car dans la figure B.1b nous remarquons aussi que la politique fixe à ce taux semble aussi efficace. Nous remarquons aussi qu'`UCB1` a l'air légèrement plus performant d'une instance, mais aussi en terme de temps de résolution.

Enfin, le scatter plot montre la politique basée sur `ST` face à l'heuristique d'origine `ibs`. À nouveau, nous remarquons au vue de la distribution des points que, presque toutes les instances insatisfaisables sont récupérées et que quelques instances satisfaisables ne le sont pas. En contrepartie, beaucoup de nouvelles instances de ces deux familles ont été récupérés par la stratégie de perturbation.

`dom/ddeg` Le cactus plot de la figure 6.4 montre de larges et nouvelles performances pour l'heuristique dynamique `dom/ddeg` en résolvant près de soixante instances supplémentaires. De nouveau, la politique basée sur `ST` et la fonction de récompense `npts` suit de peu la politique à perturbation statique fixée à 30%. Le tableau de statistiques nous montre que ces deux stratégies sont séparées par deux résolutions et un temps de résolution assez semblable. Le taux de 30% de perturbation semble avoir été retrouvé par la politique adaptative.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.	%PERT.
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/ddeg}}, \mathcal{B}^{\texttt{SP}_{30\%}})$	**466**	74 887	396	17 300	480	30.0
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/ddeg}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$	464	**74 366**	396	18 369	480	35.5
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/ddeg}}, \mathcal{B}^{\texttt{MOSS}}, \mathcal{R}^{\texttt{npts}})$	463	74 594	396	15 892	480	33.4
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/ddeg}}, \mathcal{B}^{\texttt{MOSS}}, \mathcal{R}^{\texttt{esb}})$	461	78 031	396	**14 655**	480	17.6
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/ddeg}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{auvr}})$	460	85 696	396	17 799	480	28.2
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/ddeg}}, \mathcal{B}^{\texttt{UCB1}}, \mathcal{R}^{\texttt{auvr}})$	453	99 382	396	16 242	480	26.1
`dom/ddeg`	408	197 059	396	12 589	480	–

(a) Tableau de statistiques [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]



(b) Cactus plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

(c) Scatter plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

FIGURE 6.4 – Analyse et comparaisons de l'heuristique `dom/ddeg` et ses versions perturbées [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

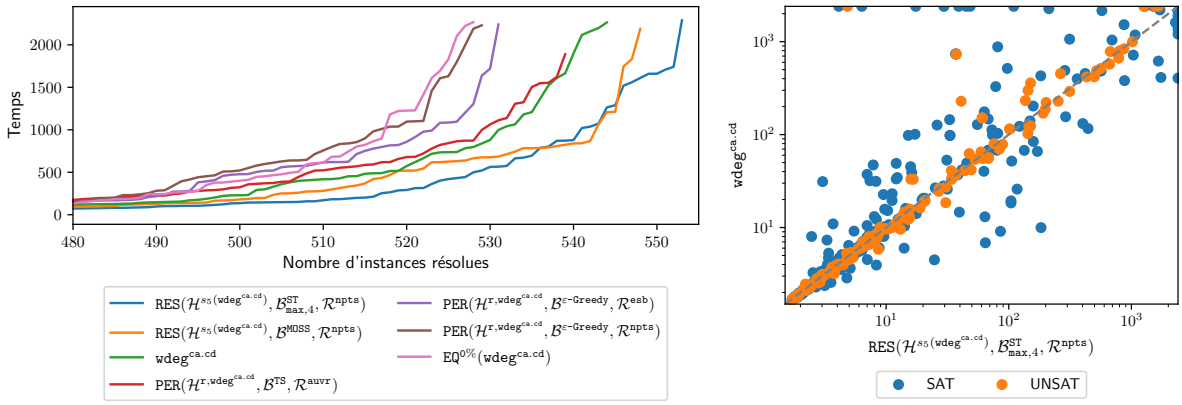
Enfin, le scatter plot nous montre que presque toutes les instances de l'heuristique d'origine `dom/ddeg` sont récupérées et que, surtout, plus d'une soixantaine sont ajoutées grâce à la politique de perturbation basée sur `ST`.

wdeg$^{\texttt{ca.cd}}$ L'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$ est une exception quant à l'efficacité de la politique basée sur la perturbation par l'heuristique aléatoire (figure 6.5). Ni la politique dont la perturbation statique est fixée empiriquement, ni celle utilisant des méthodes adaptatives n'arrivent à améliorer cette heuristique et la dégrade même. Heureusement, cette fois-ci, c'est la perturbation du tie-breaker de cette heuristique qui permet de légères améliorations de cette heuristique, déjà très performante de base, en récupérant neuf instances : à nouveau, il s'agit de la politique de bandit duelliste, ST, qui permet de récupérer les meilleures performances.

Le tableau de statistiques nous montre aussi l'écart que créent ces nouvelles performances, avec un temps de résolution réduit de plus de 25%. Nous remarquons aussi que les politiques cherchant dynamiquement la bonne perturbation à apporter semblent, elles-mêmes, totalement perturbées par cette heuristique en pratiquant plus de 50% de perturbation.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.	%PERT.
$\texttt{RES}(\mathcal{H}^{s_5(\texttt{wdeg}^{\texttt{ca.cd}})}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$	**553**	**74 798**	501	**19 166**	567	–
$\texttt{RES}(\mathcal{H}^{s_5(\texttt{wdeg}^{\texttt{ca.cd}})}, \mathcal{B}^{\texttt{MOSS}}, \mathcal{R}^{\texttt{npts}})$	548	84 269	501	24 981	567	–
$\texttt{wdeg}^{\texttt{ca.cd}}$	544	101 950	501	22 762	567	–
$\texttt{PER}(\mathcal{H}^{\texttt{r},\texttt{wdeg}^{\texttt{ca.cd}}}, \mathcal{B}^{\texttt{TS}}, \mathcal{R}^{\texttt{auvr}})$	539	110 739	501	24 819	567	56.0
$\texttt{PER}(\mathcal{H}^{\texttt{r},\texttt{wdeg}^{\texttt{ca.cd}}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{esb}})$	531	125 080	501	25 432	567	61.5
$\texttt{PER}(\mathcal{H}^{\texttt{r},\texttt{wdeg}^{\texttt{ca.cd}}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{npts}})$	529	135 137	501	29 337	567	48.6
$\texttt{EQ}^{0\%}(\texttt{wdeg}^{\texttt{ca.cd}})$	528	133 914	501	23 913	567	–

(a) Tableau de statistiques [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]



(b) Cactus plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

(c) Scatter plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

FIGURE 6.5 – Analyse et comparaisons de l'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$ et ses versions perturbées [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

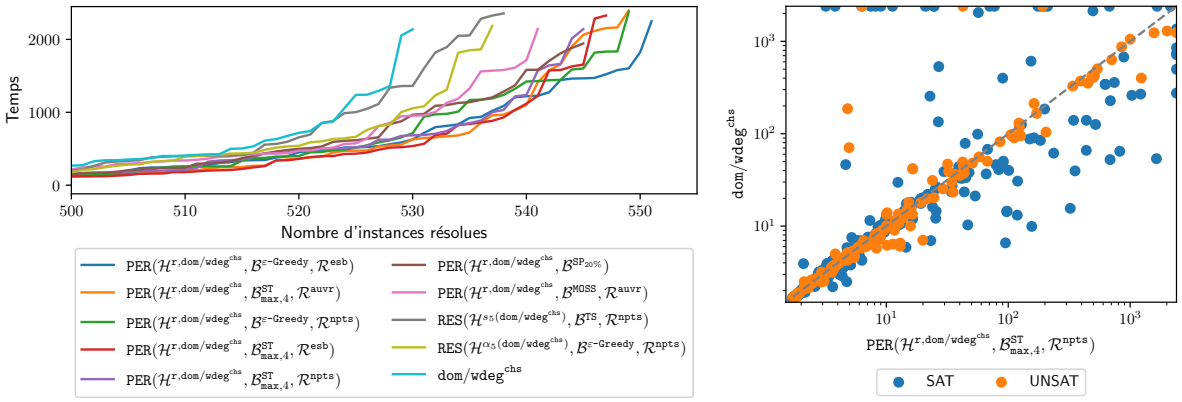
Enfin, le scatter plot montre que toutes les instances insatisfaisables sont récupérées et que quelques nouvelles ont aussi été trouvées. Seules quelques instances satisfaisables, que l'heuristique d'origine a mis plus de 500 secondes à résoudre, n'ont pas été récupérées.

dom/wdeg$^{\texttt{chs}}$ La dernière heuristique étudiée, à travers la figure 6.6, montre aussi de larges performances acquises grâce aux politiques de perturbation par le choix aléatoire. La meilleure politique, basée sur le bandit ε-Greedy et la récompense esb permet de récupérer 21 instances supplémentaires par rapport à l'heuristique d'origine. Il semblerait aussi, à travers le tableau de statistiques, que cette heu-

ristique apprécie les hauts taux de perturbation (plus de 60% pour cette meilleure politique), bien que la politique statique ait trouvé un taux bien plus faible et fixée à 20%.

	#RÉS.	TPS (S)	#RÉS. COMM.	TPS COMM. (S)	#TOT.	%PERT.
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/wdeg}^{\texttt{chs}}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{esb}})$	**551**	**76 818**	512	20 991	565	60.7
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/wdeg}^{\texttt{chs}}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{auvr}})$	549	79 783	512	23 987	565	39.7
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/wdeg}^{\texttt{chs}}}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{npts}})$	549	82 437	512	22 785	565	46.0
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/wdeg}^{\texttt{chs}}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{esb}})$	547	78 404	512	**19 822**	565	42.5
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/wdeg}^{\texttt{chs}}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},4}, \mathcal{R}^{\texttt{npts}})$	545	84 510	512	22 614	565	35.7
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/wdeg}^{\texttt{chs}}}, \mathcal{B}^{\texttt{SP}_{20\%}})$	545	88 422	512	25 565	565	20.0
$\texttt{PER}(\mathcal{H}^{\texttt{r,dom/wdeg}^{\texttt{chs}}}, \mathcal{B}^{\texttt{MOSS}}, \mathcal{R}^{\texttt{auvr}})$	541	94 875	512	24 826	565	51.2
$\texttt{RES}(\mathcal{H}^{s_5(\texttt{dom/wdeg}^{\texttt{chs}})}, \mathcal{B}^{\texttt{TS}}, \mathcal{R}^{\texttt{npts}})$	538	107 659	512	20 777	565	–
$\texttt{RES}(\mathcal{H}^{\alpha_5(\texttt{dom/wdeg}^{\texttt{chs}})}, \mathcal{B}^{\varepsilon\texttt{-Greedy}}, \mathcal{R}^{\texttt{npts}})$	537	101 111	512	21 390	565	–
$\texttt{dom/wdeg}^{\texttt{chs}}$	530	113 823	512	18 575	565	–

(a) Tableau de statistiques [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]



(b) Cactus plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

(c) Scatter plot [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

FIGURE 6.6 – Analyse et comparaisons de l'heuristique $\texttt{dom/wdeg}^{\texttt{chs}}$ et ses versions perturbées [Ψ_{14}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

Enfin, le scatter plot montrant les performances de la politique sur laquelle nous nous focalisons depuis le début de cette analyse — $\texttt{PER}(\mathcal{H}^{\texttt{r,abs}}, \mathcal{B}^{\texttt{ST}}, \mathcal{R}^{\texttt{npts}})$, montre que toutes les instances insatisfaisantes sont récupérées et que d'autres sont récupérées grâce à la perturbation.

6.5 Discussions

Perturbation et diversification De nombreuses observations utiles sont tirées de cette étude. Plus une heuristique est inefficace, plus la perturbation est efficace. Un solveur perturbé peut compenser un choix heuristique potentiellement mauvais fait par l'utilisateur, car il permet d'améliorer automatiquement ses performances en visitant des parties inconnues de l'espace de recherche. Ceci est dû aux exécutions aléatoires, pendant lesquelles l'heuristique acquiert des connaissances supplémentaires, autres que celles obtenues lors d'une exécution seule. Il est montré que les stratégies adaptatives sont, la majeure partie du temps, plus performantes ou équivalentes aux stratégies statiques dont la quantité de perturbation est fixée empiriquement, car elles peuvent adapter leur comportement à l'instance à résoudre et au paramètre

heuristique. Dans l'ensemble, les résultats montrent les avantages de la mise en place de la perturbation dans les solveurs CP pour améliorer leur performance globale, quel que soit leur paramètre par défaut.

Néanmoins, l'heuristique la plus performance, $\texttt{wdeg}^{\texttt{ca·cd}}$, n'est pas sensible à cette perturbation. Une autre méthode perturbant l'heuristique est le choix de son tie-breaker. Ainsi, la diversification de ce dernier et l'usage du framework RES permet de nouvelles performances pour chaque heuristique. Cette technique est notamment la seule à permettre à $\texttt{wdeg}^{\texttt{ca·cd}}$ de nouvelles performances.

Modèle ST et fonction de récompense npts Si un choix nous est demandé quant à la sélection d'une politique de bandit suffisamment générique pour toutes les heuristiques, ce dernier s'orienterait vers le bandit ST accompagné de la fonction de récompense npts. Cette politique a montré de larges performances dans le précédent chapitre, tout comme dans celui-ci. Nous remarquons que les autres politiques de bandit permettent aussi de bonnes performances, mais aucune de ces classiques politiques ne permet une certaine constance ; a contrario de la politique ST. Dans ce chapitre, nous avons emprunté la politique ST ayant rencontré les meilleures performances dans le précédent chapitre. Peut-être qu'une étude plus fine sur les paramètres à assigner à ST permettrait à nouveau une marge de progression.

Amélioration des heuristiques abs et ibs Les heuristiques abs et ibs possèdent, à l'origine, un mécanisme d'échantillonnage permettant d'initialiser leurs scores. Pour ce manuscrit, ce mécanisme est ignoré afin de ne pas compliquer le déroulé des stratégies multi-heuristiques. C'est pourquoi, il n'est pas si étonnant d'observer une progression, de part l'usage de bandit pour la perturbation, chez ces heuristiques demandant un échantillonnage au préalable. Dans une étude complémentaire, il pourrait être intéressant de vérifier la véritable marge gagnée sur ces heuristiques accompagnées de leurs échantillonnages de base.

D'autres perspectives Actuellement, l'apprentissage de la perturbation et de la meilleure heuristique est dissocié. Nous savons, grâce à de précédentes expériences (NOD et d'autres, non-présentées dans ce manuscrit, testant RES), que la perturbation globale où toute heuristique peut perturber les autres, est néfaste pour une résolution efficace. Néanmoins, il pourrait être intéressant, de façon contrôlée, de reprendre l'ensemble $\mathcal{H}^{\texttt{base}}$ d'heuristiques et d'y inclure l'heuristique de choix uniforme et aléatoire. Ainsi, à chaque sélection de l'heuristique aléatoire (grâce à la politique ST, par exemple), chacune des autres heuristiques choisirait, oui ou non, si elle souhaite être pertubée par ce choix aléatoire (la quantité d'aléa peut, à ce même titre, être contrôlé par un bandit indépendant). De cette manière, chaque heuristique non-aléatoire se verrait perturbée de façon contrôlée.

Une autre perspective concerne la diversification et notamment l'heuristique $\texttt{wdeg}^{\texttt{ca·cd}}$. Actuellement, les tie-breakers sont choisis arbitrairement. La sélection de ces tie-breakers est assez cruciale : ils doivent être sélectionnés de manière à rendre le comportement des différentes heuristiques h_1 (dans le schéma $h_1 > h_2$ où h_2 est le tie-breaker) très différent. Une manière d'être sûr que deux mêmes heuristiques h_1 aient un comportement différent serait de choisir le tie-breaker lex et son opposé anti-lex de façon à ce que les deux instances de h_1 se concentrent sur des parties complémentaires de l'espace de recherche.

6.6 Conclusion

Dans ce chapitre, plusieurs stratégies ont été présentées et améliorent les performances et la robustesse du solveur en perturbant ou diversifiant l'heuristique de choix de variable par défaut. C'est la première fois qu'une approche tente d'apprendre *comment* et *quand* appliquer la randomisation en ligne et sans paramètres. La recherche aléatoire contrôlée aide l'heuristique d'ordonnancement des variables

à acquérir des connaissances supplémentaires sur des parties de l'espace de recherche qu'elle n'était pas destinée à explorer. Nous résumons les avantages de cette nouvelle approche :

- Les différentes techniques de perturbation améliorent constamment les performances du solveur, indépendamment de l'heuristique utilisée comme ligne de base dans le solveur. Une stratégie perturbée (ou diversifiée dans le cas précis de $\texttt{wdeg}^{\texttt{ca}\cdot\texttt{cd}}$) est toujours plus performante que son homologue de base, tant en terme de temps que d'instances résolues.
- Les heuristiques perturbées peuvent compenser un mauvais choix heuristique fait par l'utilisateur. Grâce à la perturbation, la performance du solveur avec une mauvaise heuristique initiale peut atteindre ou même dépasser la performance du solveur avec une meilleure heuristique de base. Il s'agit d'une étape vers la résolution autonome et adaptative, où le solveur apprend et ajuste son comportement à l'instance en cours de résolution.
- Cette approche est générique et facile à intégrer dans tout solveur qui exploite les redémarrages.

Chapitre 7

Extension aux Problèmes d'Optimisation

Sommaire

7.1 Introduction

En programmation par contraintes (CP), même si de nombreuses extensions ont été proposées depuis les années 70, il est habituel de traiter soit des problèmes de satisfaction de contraintes (CSP), soit des problèmes d'optimisation sous contraintes (COP). En plus d'un ensemble de variables à assigner tout en satisfaisant un ensemble de contraintes, une instance COP implique une fonction objectif à optimiser (c'est-à-dire une fonction de coût à minimiser ou une fonction de récompense à maximiser). La résolution d'une instance COP nécessite, non seulement, de prouver la satisfaisabilité (c'est-à-dire de trouver au moins une solution), mais aussi et idéalement, de prouver l'optimalité ou au moins de trouver des solutions d'assez bonne qualité (c'est-à-dire proches de l'optimalité).

La recherche avec retour en arrière est une approche complète classique pour explorer l'espace de recherche d'une instance COP. Cela revient à résoudre une série d'instances CSP. En supposant une fonction objectif `obj` à minimiser où initialement `obj` est traitée sous la forme d'une contrainte $\texttt{obj} < \infty$, et chaque fois qu'une nouvelle solution de coût B est trouvée, B est utilisée comme nouvelle borne pour la contrainte objectif, de façon à devenir $\texttt{obj} < B$ (formant ainsi une instance CSP plus contrainte).

De cette façon, toute nouvelle solution trouvée est garantie d'être de meilleure qualité (coût inférieur) que la précédente, et l'optimalité peut être prouvée lorsque l'instance du problème devient insatisfaisable.

Au même titre que les problèmes de satisfaisabilité, les solveurs de problèmes d'optimisation sont aussi soumis aux différences d'efficacité selon la sélection d'une heuristique pour une instance donnée. À nouveau, les défis de sélection et d'apprentissage de la meilleure heuristique, ainsi que de la perturbation d'heuristique, sont applicables dans ce cadre.

Avant d'appliquer les techniques proposées dans les précédents chapitres, nous décrivons plus précisément le principe de résolution d'un solveur de contraintes sous optimisation et entamons une première campagne montrant les capacités du solveur par défaut `ACE` et la configuration utilisée par la suite (utilisant la suite de Luby). Cette première campagne nous donne aussi l'occasion de présenter quelques méthodes d'évaluation des performances en optimalité. L'étude des solveurs d'optimisation nous a aussi donné l'occasion d'explorer une nouvelle méthode d'amélioration agressive de la borne au cours de la recherche. Une fois ces points décrits, nous exploitons les précédentes techniques de bandit afin d'améliorer l'autonomie et la qualité de résolution de ces solveurs de contraintes sous optimisation.

7.2 Présentation des problèmes d'optimisation

7.2.1 Principe de résolution

Un *réseau de contraintes sous optimisation* (CNO — *Constraint Network under Optimization*) est un réseau de contraintes associé à une fonction objectif $\texttt{obj}$ (définition 21) faisant correspondre toute solution à une valeur [8] dans $\mathbb{D}$. Sans aucune perte de généralité, nous considérerons que $\texttt{obj}$ doit être minimisée. Une solution S d'un CNO est une solution du CN sous-jacent ; S est *optimal* s'il n'existe aucune autre solution S' telle que $\texttt{obj}(S') < \texttt{obj}(S)$. La tâche habituelle du *problème d'optimisation sous contraintes* (COP — *Constraint Optimization Problem*) est de trouver une solution optimale pour un CNO donné.

Comme nous le savons, la recherche avec retour en arrière est une procédure complète classique pour résoudre les instances de CSP/COP. Elle alterne les affectations de variables (et les réfutations) et un mécanisme appelé propagation de contraintes afin de filtrer l'espace de recherche. Typiquement, comme pour MAC (*Maintaining Arc Consistency*) qui propage les contraintes en maintenant la propriété de cohérence d'arc, un arbre de recherche binaire $\mathcal{T}$ est construit : à chaque nœud interne de $\mathcal{T}$, (i) on sélectionne un couple (x, v) où x est une variable non-fixée et v est une valeur dans $\texttt{dom}(x)$, et (ii) deux cas (branches) sont considérés, correspondant à l'affectation $x = v$ et à la réfutation $x \neq v$.

L'ordre dans lequel les variables sont choisies pendant le parcours en profondeur de l'espace de recherche est décidé par une *heuristique de choix de variable*. L'ordre dans lequel les valeurs sont choisies lors de l'affectation des variables est déterminé par une *heuristique de choix de valeur* ; pour les COPs, il est fortement recommandé d'utiliser d'abord la valeur présente dans la dernière solution trouvée, ce qui est une technique connue sous le nom de *solution(-based phase) saving* [Vion & Piechowiak 2017, Demirovic *et al.* 2018].

La recherche avec retour en arrière pour COP repose sur la résolution de CSP : le principe consiste à ajouter au réseau de contraintes une *contrainte objectif* spéciale $\texttt{obj} < \infty$ (bien qu'elle soit initialement satisfaite), et de mettre à jour la borne de cette contrainte chaque fois qu'une nouvelle solution est trouvée. Cela signifie qu'à chaque nouvelle solution S trouvée avec un coût $B = \texttt{obj}(S)$, la contrainte objectif devient $\texttt{obj} < B$. Par conséquent, une séquence de solutions, dont la qualité est croissante, est générée — la SATisfaisabilité est systématiquement prouvée par rapport à la borne actuelle de la contrainte objectif, jusqu'à ce qu'il n'en existe plus aucune — l'insatisfaisabilité (*UNSATisfiability*) est

8. Pour simplifier la présentation, nous considérons que les coûts sont donnés par des valeurs entières.

finalement prouvée par rapport à la borne imposée par la dernière solution trouvée, ce qui garantit que la dernière solution trouvée est optimale.

Tout comme nous l'avons vu pour les CSPs, les politiques de redémarrage jouent un rôle important dans la résolution de COPs car elles permettent de pallier le phénomène de *queue longue*.

7.2.2 Proposition d'évaluations

Dans la mesure du possible, nous souhaitons des méthodes dites *absolues* afin d'évaluer les campagnes. Ces méthodes absolues permettent d'évaluer un solveur dans son entièreté sans que l'évaluation n'implique d'autres solveurs. C'est le cas, par exemple, dans nos précédentes évaluations de campagne sur les instances CSP : le nombre d'instances résolues et le temps de résolution sur l'entièreté des instances sont indépendants de la résolution d'autres solveurs. En revanche, considérer les instances communément résolues entre un ensemble de solveurs crée une dépendance entre eux. Ce dernier cas correspond à une évaluation *relative*. Ces évaluations relatives, bien que nécessaires, sont à éviter car, tout changement de l'ensemble de solveurs impliquent un changement pour l'ensemble entier des évaluations de solveurs. Possiblement, un solveur ayant le meilleur score dans un ensemble donné de solveurs, si cet ensemble reçoit un nouveau solveur, l'évaluation change pour l'ensemble des solveurs et l'ancien gagnant n'est pas forcément conservé ; le nouveau gagnant n'est pas nécessairement le nouvel arrivant.

En optimisation, les évaluations proposées sont d'autant plus sujets à la nécessité d'utiliser ces méthodes relatives. Une manière d'obtenir une meilleure interprétation d'évaluation relative est de diversifier ces méthodes relatives. Tout comme pour les campagnes de CSPs, il existe des méthodes absolues pour évaluer les campagnes en optimisation : nous pouvons compter le nombre d'instances pour lesquelles le solveur à prouver l'optimalité, tout comme le temps de résolution sur la campagne entière. L'optimalité est une part importante de l'évaluation d'une campagne COP, mais n'est pas une évaluation très exhaustive. D'autres méthodes, amenant à une comparaison relative, comptent le nombre de fois où un solveur obtient la dominance dans une campagne : un ou plusieurs solveurs possèdent la meilleure borne de l'ensemble de la campagne à un temps donné. Une autre méthode relative tente de normaliser les bornes de chaque solveur, à un temps donné, entre 0 et 1, dans le but de remarquer les solveurs étant, de manière globale, proche de la meilleure borne. Par la suite, nous exprimons plus formellement ces méthodes.

Tout d'abord, nous présentons différentes notations que nous utilisons pour évaluer les résultats expérimentaux. Pour des raisons de simplicité, nous continuons de considérer que nous n'avons que des problèmes de minimisation. Étant donné un ensemble $\mathcal{I}$ d'instances et un ensemble Ψ de solveurs,

$$b^i_{s,t}$$

correspond à la meilleure borne (c'est-à-dire, dont le coût est le plus faible) obtenue par le solveur $s \in \Psi$ sur l'instance $i \in \mathcal{I}$ en t seconde(s), où $t \in [0, ..., T]$ et T est le timeout. Nous utilisons localement les valeurs t et T qu'il ne faut pas confondre avec le nombre de runs et l'horizon. $b^i_{s,t}$ peut être égale à ∞ si le solveur n'a actuellement trouvé aucune solution. Nous avons également un booléen

$$c^i_{s,t}$$

dont la valeur est *vrai* lorsque le solveur a accompli une exploration complète de l'espace de recherche de l'instance i en moins de t secondes, *faux* sinon. Nous pouvons maintenant définir deux valeurs correspondant respectivement à la plus petite (meilleure) et à la plus grande (pire) bornes obtenues par un ensemble de solveurs Ψ sur une instance donnée i au temps t :

$$\mathtt{min}^i_t = \min_{s \in \Psi} b^i_{s,t}$$

$$\texttt{max}_t^i = \max_{s \in \Psi} b_{s,t}^i$$

Ces précédentes déclarations permettent de normaliser les bornes $b_{s,t}^i$ en empruntant l'opération de normalisation *min-max* afin de rendre possible l'agrégation des instances :

$$n_{s,t}^i = \begin{cases} 0, & \text{si } b_{s,t}^i = \infty \\ 1, & \text{si } \texttt{max}_t^i = \texttt{min}_t^i \implies b_{s,t}^i = \texttt{max}_t^i \\ \frac{\texttt{max}_t^i - b_{s,t}^i}{\texttt{max}_t^i - \texttt{min}_t^i}, & \text{sinon} \end{cases}$$

Dans le cas où un solveur n'a trouvé aucune solution, sa récompense est de 0, et dans le cas où les bornes, la plus petite et la plus haute, sont égales (ce qui signifie que le solveur a trouvé l'unique solution connue au temps t), sa récompense est de 1. Sinon, on utilise l'opérateur *min-max*.

Avec ces précédentes variables en main, nous pouvons maintenant définir les opérateurs d'évaluation pour une campagne composée des solveurs Ψ sur un ensemble d'instances $\mathcal{I}$. Pour un solveur $s \in \Psi$ au temps t, voici les différentes évaluations absolues et relatives disponibles sur l'ensemble d'instances $\mathcal{I}$:

$$\begin{aligned} \mathcal{E}_{s,t}^{\texttt{opti}} &= \frac{1}{|\mathcal{I}|} \times \sum_{i \in \mathcal{I}} \mathbb{1}_{c_{s,t}^i} \\ \mathcal{E}_{s,t}^{\texttt{domi}} &= \frac{1}{|\mathcal{I}|} \times \sum_{i \in \mathcal{I}} \lfloor n_{s,t}^i \rfloor \\ \mathcal{E}_{s,t}^{\texttt{quali}} &= \frac{1}{|\mathcal{I}|} \times \sum_{i \in \mathcal{I}} n_{s,t}^i \\ \mathcal{E}_{s,t}^{\texttt{borda}} &= \frac{1}{|\mathcal{I}|} \times \sum_{i \in \mathcal{I}} \texttt{rang}(i, s, t) \end{aligned}$$

où $\mathbb{1}_\alpha$ retourne 1 si α est vrai, sinon 0. La première évaluation, $\mathcal{E}_{s,t}^{\texttt{opti}}$, correspond à une méthode absolue calculant le nombre de fois où le solveur s a complété la recherche (avec preuve d'optimalité ou d'insatisfaisabilité) au temps t sur l'ensemble des instance $\mathcal{I}$; la proportion d'optimalité est calculée en normalisant par le nombre d'instances. L'évaluation $\mathcal{E}_{s,t}^{\texttt{domi}}$ compte le nombre de fois où un solveur s trouve la meilleure borne connue (c'est-à-dire, quand $n_{s,t}^i = 1$) pour une instance i au temps t, et ce, sur l'ensemble des instances $\mathcal{I}$ et en normalisant par le nombre d'instances : elle retourne donc la proportion de dominance du solveur s sur l'ensemble $\mathcal{I}$ d'instances au moment t. À travers l'évaluation $\mathcal{E}_{s,t}^{\texttt{quali}}$, nous calculons la qualité moyenne des bornes trouvées à travers l'ensemble des instances de $\mathcal{I}$ pour un solveur s au moment t.

La dernière évaluation, $\mathcal{E}_{s,t}^{\texttt{borda}}$, est basée sur une procédure proposée dans la *compétition MiniZinc* [9]. La procédure de notation s'inspire du système de vote par comptage de Borda. Chaque instance est traitée comme un électeur qui classe les solveurs. Chaque solveur marque un nombre de points égal au nombre de solveurs qu'il a battus dans le classement (plus ou moins, selon certaines égalités entre solveurs que nous explicitons par la suite). $\mathcal{E}_{s,t}^{\texttt{borda}}$ retourne le rang moyen d'un solveur s au temps t sur l'ensemble des instances (considérées comme électeurs) $\mathcal{I}$. La fonction `rang` produit l'ensemble des duels entre le solveur s et les autres solveurs $s' \in \Psi \setminus \{s\}$ et retourne son rang (pas nécessairement un entier) :

9. Lien vers les règles de la compétition MiniZinc : https://www.minizinc.org/challenge2021/rules2021.html

$$\texttt{rang}(i, s, t) = \sum_{s' \in \Psi \setminus \{s\}} \texttt{duel}(i, s, s', t)$$

La comparaison de deux solveurs (s, s') se fait grâce à la fonction $\texttt{duel}(i, s, s', t)$ pour une instance i au temps t :

$$\texttt{duel}(i, s, s', t) = \begin{cases} \text{(i)} & \textit{Si un solveur a complété la recherche et pas l'autre} \\ & \mathbb{1}_{c^i_{s,t}}, \qquad \textbf{si } c^i_{s,t} \neq c^i_{s',t} \\ \text{(ii)} & \textit{Sinon, si la recherche est complète pour les deux} \\ & \frac{\texttt{tps}(c^i_{s',t})}{\texttt{tps}(c^i_{s,t}) + \texttt{tps}(c^i_{s',t})} \quad \textbf{si } c^i_{s,t} \wedge c^i_{s',t} \\ \text{(iii)} & \textit{Sinon, si l'un des solveurs n'a pas obtenu de solution} \\ & \mathbb{1}_{b^i_{s,t} < \infty}, \qquad \textbf{si } b^i_{s,t} + b^i_{s',t} = \infty \\ \text{(iv)} & \textit{Sinon, si les bornes sont différentes} \\ & \mathbb{1}_{b^i_{s,t} < b^i_{s',t}}, \qquad \textbf{si } b^i_{s,t} \neq b^i_{s',t} \\ \text{(v)} & \textit{Sinon, les bornes sont égales} \\ & \frac{\texttt{tps}(b^i_{s',t})}{\texttt{tps}(b^i_{s,t}) + \texttt{tps}(b^i_{s',t})} \quad \textbf{sinon} \end{cases}$$

Le départage de deux solveurs s et s' se fait en cinq conditions et retourne soit une valeur binaire, 0 ou 1, quand les deux solveurs sont clairement distinguables, ou alors un réel entre 0 et 1 lorsque nous ne pouvons discriminer les solveurs que par leur temps d'exécution. La première condition (i) vérifie si un solveur, et pas l'autre, a complété entièrement la recherche avec un parcours exhaustif de l'espace de recherche : le solveur à la recherche complète reçoit la récompense 1 et l'autre 0. La deuxième condition (ii) vérifie si les deux solveurs ont communément produit une recherche complète. Dans ce cas, la seule manière de pouvoir les discriminer correspond à la prise en compte de leur temps d'exécution avant résolution complète : la récompense correspond, pour un solveur s, au temps du solveur adverse s' divisé par leurs temps additionnés. `tps` retourne le temps de résolution nécessaire afin de parcourir l'entièreté de l'espace de recherche pour le solveur s et l'instance i ; t est nécessairement plus grand que le retour de cette fonction. La condition suivante (iii) vérifie si au moins un solveur n'a pas trouvé de borne : le ou les solveurs sans solution reçoivent la récompense 0, sinon 1. L'avant-dernière condition (iv) vérifie si les bornes sont différentes : le solveur à la meilleure borne reçoit la récompense 1, l'autre 0. Enfin, le dernier cas (v) correspond au scénario où les deux solveurs possèdent la même borne sans avoir complété la recherche. Dans ce cas, le départage grâce au temps (comme dans le cas de la condition (ii)) permet de récompenser les deux solveurs avec une valeur comprise entre 0 et 1. `tps` retourne le temps de résolution nécessaire pour trouver la borne donnée en paramètre pour le solveur s sur l'instance i ; t est nécessairement plus grand que le retour de cette fonction.

Dans cette section, cinq opérateurs d'évaluation des performances de solveurs en optimisation sur un ensemble d'instances $\mathcal{I}$ sont proposés. Par la suite, ces opérateurs permettent de produire des graphiques projetant l'efficacité, en tout temps t, des différents solveurs et permettent donc d'apprécier la qualité du solveur sur tout un temps de résolution avec quatre points de vue.

7.2.3 Évaluation du solveur ACE

Pour une première campagne expérimentale dans le domaine de l'optimisation et des solveurs de contraintes, nous reprenons les différentes configurations d'ACE proposées par Ψ_1. Soit Ψ_{15} :

$$\left\{ \texttt{ACE}^{r}_{\texttt{lc}=x} \;\middle|\; \begin{array}{l} r \in \{\texttt{def}, \texttt{luby}\} \\ x \in \{0, 2\} \end{array} \right\} \qquad (\Psi_{15})$$

Au même titre qu'en résolution de CSPs, le solveur ACE possède quelques particularités en optimisation que nous désactivons afin d'obtenir un solveur plus neutre pour les prochaines expérimentations. Le solveur ACE par défaut réinitialise périodiquement le score des variables de son heuristique, ainsi que sa suite de redémarrage (périodiquement aussi et à chaque nouvelle borne trouvée). Ainsi, nous proposons à nouveau d'expérimenter le solveur ACE par défaut et sa version neutre (c'est-à-dire sans réinitialisation) avec la suite de Luby. Par cette même occasion, nous faisons varier la valeur donnée au mécanisme lc (*Last Conflict*). Les campagnes qui suivent s'appuient toutes sur l'ensemble d'instances $\mathcal{I}_{\texttt{COP}}$.

Afin d'appréhender les résultats de ces quatre solveurs, nous produisons les graphiques contenus dans la figure 7.1 partagée en deux colonnes. Nous nous intéressons tout d'abord à la colonne de gauche correspondant à la fluctuation des quatre opérateurs d'évaluation ($\mathcal{E}^{\texttt{opti}}_{s,t}$, $\mathcal{E}^{\texttt{domi}}_{s,t}$, $\mathcal{E}^{\texttt{quali}}_{s,t}$ et $\mathcal{E}^{\texttt{borda}}_{s,t}$) en fonction du temps t. Une première observation, à propos de ces graphiques, est qu'ils nécessitent d'être plus précis en personnalisant les valeurs minimales et maximales de l'axe y. Par la suite, nous proposons une autre manière de clarifier ces informations en sélectionnant un solveur témoin s'. La colonne de droite de la figure 7.1 propose de soustraire à toute évaluation de solveur s de l'ensemble Ψ_{15}, l'évaluation du solveur s'. Ainsi, nous observons que le solveur par défaut se retrouve, inévitablement, en $y = 0$ et nous pouvons comparer bien plus aisément les performances des autres solveurs par rapport à ce solveur témoin. Les tendances des solveurs par rapport à ce solveur témoin sont ainsi conservés des graphiques de gauche vers les graphiques de droite avec un aperçu plus clair.

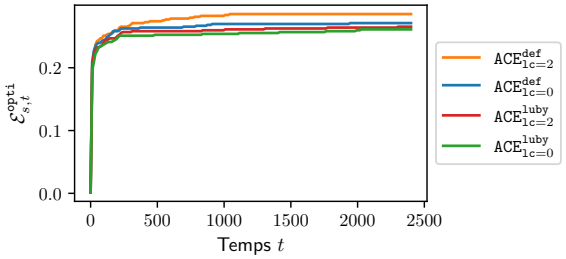
Nous nous concentrons, maintenant, entièrement sur la colonne de droite de la figure 7.1 afin d'étudier la campagne des solveurs Ψ_{15} sur l'ensemble $\mathcal{I}_{\texttt{COP}}$ d'instances. Le premier graphique (figure 7.1b), avec l'évaluation absolue sur le comptage du nombre d'optimalités (et insatisfaisabilités), nous montre que les solveurs ACE par défaut sont, pratiquement, toujours plus performants que les solveurs basés sur Luby. Plus particulièrement, les solveurs retenant les deux derniers conflits sont plus performants que leurs homologues neutres. Ainsi, le solveur témoin est ici le meilleur. Le second graphique (figure 7.1d) montre la progression de la dominance entre les solveurs. Un changement dans l'ordre des solveurs est remarqué : le solveur basé sur Luby, bien que plus faible au démarrage que son homologue par défaut, a tendance à l'égaler à l'approche du timeout. De façon moins prononcée, le troisième graphique (figure 7.1f) montre aussi cette tendance avec la prise en compte plus « globale » de la qualité des solutions obtenues pour chaque solveur. Enfin, le dernier graphique (figure 7.1h) confirme ces deux dernières tendances en montrant le classement moyen des solveurs (en comparaison au solveur témoin).

Par la suite, et en prolongement de la configuration utilisée dans les précédents chapitres, nous travaillons entièrement avec le solveur $\texttt{ACE}^{\texttt{luby}}_{\texttt{lc}=0}$ qui, dans les actuelles performances, possède une perte oscillant entre 5% et 10% par rapport au solveur témoin. Nous considérons dès à présent que $\texttt{ACE}^{\texttt{luby}}_{\texttt{lc}=0}$ est maintenant le solveur par défaut pour les campagnes qui suivent.

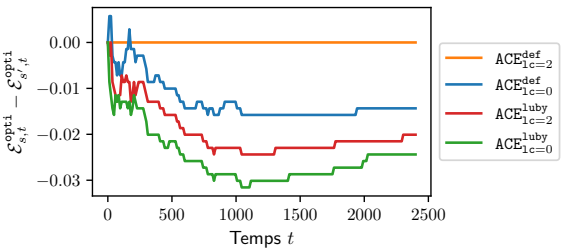
7.3 Descente agressive de borne [10]

Il n'est pas rare que les problèmes d'optimisation issus de l'industrie soient faiblement contraints, ce qui signifie qu'un grand nombre de solutions existent avec des qualités diverses dans différentes parties

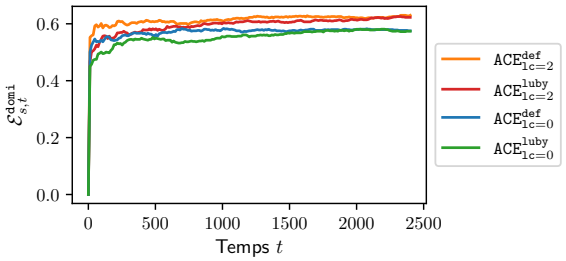
10. Cette section est une étude en collaboration avec un autre doctorant du CRIL : Thibault Falque.



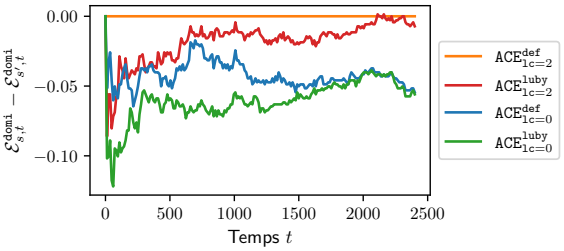
(a) Proportion d'*optimalité*



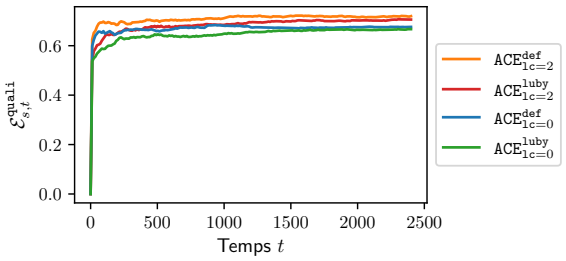
(b) Différence de proportion d'*optimalité* avec le solveur $s' = \texttt{ACE}^{\texttt{def}}_{\texttt{lc}=2}$



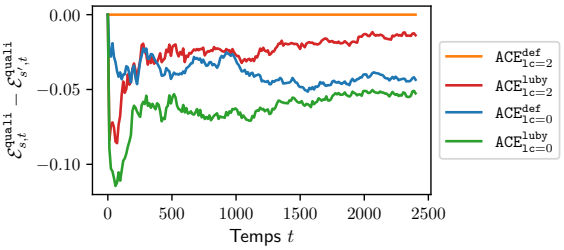
(c) Proportion de *dominance*



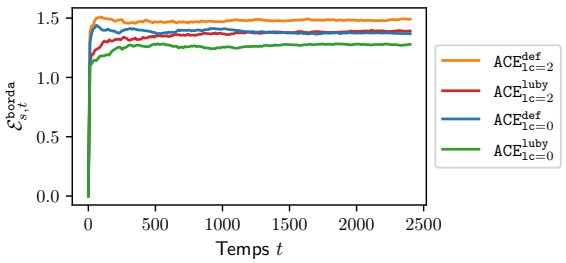
(d) Différence de proportion de *dominance* avec le solveur $s' = \texttt{ACE}^{\texttt{def}}_{\texttt{lc}=2}$



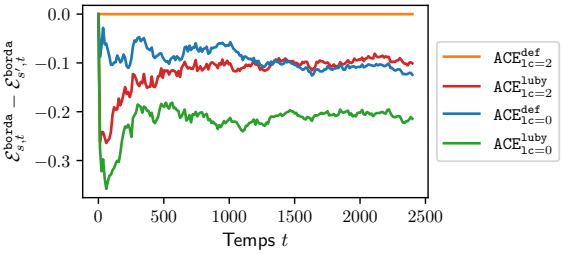
(e) *Qualité* moyenne des solutions



(f) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \texttt{ACE}^{\texttt{def}}_{\texttt{lc}=2}$



(g) Classement moyen de *Borda*



(h) Différence du classement moyen de *Borda* avec le solveur $s' = \texttt{ACE}^{\texttt{def}}_{\texttt{lc}=2}$

FIGURE 7.1 – Comparaison des configurations de base du solveur ACE [Ψ_{15}, $\mathcal{I}_{\text{COP}}$, ↻]

de l'espace de recherche. Dans de telles situations, l'application de la recherche avec retour en arrière peut être pénalisée parce que la descente des bornes (c'est-à-dire la séquence décroissante des bornes successivement trouvées) peut être très lente : la distance entre deux bornes successives peut être plutôt faible. Dans cette étude, nous proposons une approche pour tenter de réduire ce phénomène en modifiant les bornes d'une manière plus agressive : tant qu'elle est réussie, de nouvelles bornes pour la contrainte objectif sont calculées en suivant une croissance exponentielle. Bien-sûr, dans le cas où la borne rend le problème insatisfaisable ou très difficile à résoudre, le mécanisme de redémarrage nous permet de reprendre la recherche sur des pistes plus certaines.

Même si d'autres techniques de recherche existent dans la littérature, comme par exemple la méta-heuristique largement utilisée *Large Neighborhood Search* (LNS) [SHAW 1998], dans cet article, nous nous concentrons sur la recherche en profondeur avec retour en arrière, équipée du mécanisme de *solution(-based phase) saving* qui s'est avérée être très efficace [VION & PIECHOWIAK 2017, DEMIROVIC *et al.* 2018].

Nous montrons que cette approche peut rendre le solveur plus robuste, surtout au début de la recherche. Dans les sections qui suivent, nous présentons le principe de Descente Agressive de Borne (ABD — *Aggressive Bound Descent*) et avant de conclure, les résultats expérimentaux sont donnés.

7.3.1 Principe de la descente agressive de borne

Présentation Lors de la résolution d'une instance COP, la *descente de borne* est définie comme la séquence $D = \langle B_1, B_2, \ldots \rangle$ de bornes successives identifiées par l'algorithme de recherche. À un extrême, cette séquence ne contient qu'une seule valeur, la borne optimale. À un autre extrême, elle contient une grande séquence de valeurs, chacune étant proche de la précédente : la descente de borne est alors dite *lente*. C'est le cas lorsque la valeur moyenne de la séquence dérivée des gains (ou écarts) de bornes $G = \langle B_1 - B_2, B_2 - B_3, \ldots \rangle$ est petite (proche de 1).

Il est certain qu'une descente lente de borne indique qu'il y a une certaine marge d'amélioration dans la façon dont la recherche avec retour arrière est menée. En effet, l'énumération d'un grand nombre de solutions proches les unes des autres avant d'atteindre l'optimalité implique la résolution d'un grand nombre d'instances de problèmes de satisfaction dérivés et cela peut être pénalisant. C'est pourquoi nous proposons une politique *agressive* de descente de borne : la politique ABD. Au lieu de fixer la borne stricte de la contrainte objective à B lorsqu'une nouvelle solution de coût B est trouvée, nous proposons de la fixer à une valeur éventuellement inférieure B'.

Une première politique ABD simple pourrait consister à utiliser une différence statique entre B et B' : $B' = B - \Delta$ où Δ est une valeur entière positive fixe. Cependant, cette politique *statique* souffre clairement d'un manque d'adaptabilité, et de plus, fixer la bonne valeur pour Δ peut dépendre du problème et ne pas être très facile à réaliser.

C'est pourquoi nous proposons des politiques *dynamiques* pour ABD, inspirées des études concernant les suites utilisées par les politiques de redémarrage.

Pour définir des politiques ABD dynamiques, nous introduisons d'abord quelques suites classiques d'entiers strictement positifs, c'est-à-dire des fonctions $\texttt{abd} : \mathbb{N}^+ \to \mathbb{N}^+$. Bien que détaillé plus loin, le paramètre $i \geq 1$ de ces séquences correspond au nombre de mises à jour successives réussies de la borne, c'est-à-dire de mises à jour successives agressives de la borne de la contrainte objective tout en conservant la satisfiabilité (au sein d'un même run).

Plus précisément, quatre suites, dont les trois premières sont initialement dans la section 1.3.3 à propos des mécanismes de redémarrage, sont utilisées dans cette étude :

$$\texttt{exp}(i) = 2^{i-1}$$

$$\texttt{rexp}(i) = \begin{cases} 2^{k-1}, & \text{si } i = \frac{k(k+1)}{2} \\ 2^{i-\frac{k(k+1)}{2}-1}, & \text{si } \frac{k(k+1)}{2} < i < \frac{(k+1)(k+2)}{2} \end{cases}$$

$$\texttt{luby}(i) = \begin{cases} 2^{k-1}, & \text{si } i = 2^k - 1 \\ \texttt{luby}(i - 2^{k-1} + 1), & \text{si } 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

$$\texttt{prev}(i) = \begin{cases} 1, & \text{si } i = 1 \\ G_{i-1} \times 2, & \text{sinon} \end{cases}$$

où, dans l'Équation `prev`, G_i est la ième valeur de la suite de gains, telle que définie précédemment.

Nous ne présentons plus les trois premières fonctions étant bien connues depuis la présentation des mécanismes de redémarrages. Néanmoins, la dernière suite n'est pas extraite depuis la description de ces mécanismes : `prev`. Celle-ci est basée sur la suite des gains, et suit également une progression exponentielle.

Chaque suite dans $\texttt{ABD} = \{\texttt{exp}, \texttt{rexp}, \texttt{luby}, \texttt{prev}\}$ nous permet de définir un éponyme de la politique ABD.

Généralité Soit $\mathcal{T}$ l'arbre de recherche actuel construit par l'algorithme de recherche (c'est-à-dire pendant le run actuel). Soit D la descente de bornes produite depuis le début du run courant, et soit $\texttt{abd} \in \texttt{ABD}$. L'exécution courante peut rencontrer trois situations distinctes :

1. le run en cours est arrêté car le cutoff ou le timeout est atteint,
2. le run en cours est arrêté parce que les algorithmes de recherche indiquent qu'il n'existe plus de solution,
3. une nouvelle solution S est trouvée.

Tout d'abord, nous discutons du cas le plus intéressant : le troisième. La politique ABD stipule que lorsqu'une nouvelle solution S de coût B est trouvée, B est ajouté à D, et la borne de la contrainte objective est fixée à $B+1-\texttt{abd}(i)$, où $i = |D|$. En d'autres termes, la contrainte objectif devient : $\texttt{obj} < B+1-\texttt{abd}(i)$; notez que 1 est ajouté à B car les fonctions `abd` ne retournent que des valeurs supérieures ou égales à 1. Maintenant, nous donnons une description générale précise (traitant en particulier les deux premières situations ci-dessus) de la façon dont une politique ABD peut être implantée dans une recherche avec retour en arrière.

Exemple d'implantation La fonction `solve`, algorithme 4, tente de résoudre le COP $\mathcal{P}$ grâce à l'usage de la politique de descente agressive de bornes `abd` spécifiée.

Algorithme 4 : $\texttt{solve}(\mathcal{P}, \texttt{abd})$

Output : $\underline{B_\mathcal{P}}..\overline{B_\mathcal{P}}$, runStatus

1 $\underline{B_\mathcal{P}}..\overline{B_\mathcal{P}} \leftarrow -\infty..+\infty$;
2 **faire**
3 $\quad \mathcal{P}$, runStatus $\leftarrow \texttt{run}(\mathcal{P}, \texttt{abd})$;
4 **tant que** runStatus $=$ CONTINUE ;
5 **retourner** $(\underline{B_\mathcal{P}}..\overline{B_\mathcal{P}}$, runStatus) ;

Tout d'abord, les bornes inférieure et supérieure, désignées par $\underline{B_{\mathcal{P}}}$ et $\overline{B_{\mathcal{P}}}$, de la fonction objectif de $\mathcal{P}$ sont respectivement initialisées à $-\infty$ et $+\infty$ (ou toute autre valeur pertinente pouvant être précalculée). Pendant la recherche, ces bornes seront mises à jour (mais pour des raisons de simplicité, cela ne sera pas explicitement montré dans le pseudo-code). À la ligne 2, la séquence de runs (redémarrages) est lancée. Chaque fois qu'une nouvelle exécution est terminée, elle renvoie le réseau de contraintes (éventuellement mis à jour avec certaines contraintes ou certains *nogoods* qui ont été appris ; ceci sera discuté plus en détail plus tard) ainsi qu'un statut. Le statut prend l'une des valeurs suivantes : CONTINUE si le solveur est autorisé à poursuivre avec un nouveau run ; COMPLETE si le dernier run a exploré exhaustivement l'espace des solutions ; INCOMPLETE si le solveur a atteint la limite de temps sans avoir exploré entièrement l'espace de recherche. Enfin, la fonction renvoie les meilleures bornes trouvées (dans le cas où l'optimalité a été prouvée, nous avons $\underline{B_{\mathcal{P}}} = \overline{B_{\mathcal{P}}}$) et l'état final de la recherche.

La fonction `run`, algorithme 5, effectue une recherche, en suivant les politiques de redémarrage et de `abd`. Avant d'aller plus loin, nous devons introduire la notion de *safe/unsafe* : lorsqu'on demande au solveur de diminuer agressivement sa borne objective, nous pouvons entrer dans une partie de l'espace de recherche qui est UNSAT. Si l'insatisfaisabilité est prouvée pendant l'exécution actuelle, cela peut être dû à notre approche agressive, et par conséquent, nous devons traiter ce problème. Ce point est abordé ci-dessous.

La fonction commence par initialiser un compteur i à 1. Il correspond au nombre de fois où l'on a essayé de trouver une nouvelle solution pendant l'exécution courante. À la ligne 2, Σ_i est la séquence de décisions prises le long de la branche la plus à droite du run actuel, juste avant de commencer la prochaine tentative de trouver une nouvelle solution ; de cette façon, nous pouvons continuer à chercher à partir du même endroit (en pratique, nous reprenons simplement la recherche après l'avoir arrêtée). Initialement, aucune décision n'est prise (et donc, Σ_1 est l'ensemble vide). D'un point de vue pratique, comme nous le verrons, seules les deux dernières séquences Σ_i et Σ_{i-1} seront utiles (pour traiter les cas de résolution *safe* et *unsafe*).

Ensuite, l'algorithme effectue des parcours itératifs tant que de nouvelles solutions peuvent être trouvées. À chaque tour de boucle, la prochaine limite d'écart Δ est calculée en sollicitant la politique `abd` et i est incrémenté (lignes 4 et 5). Pour effectuer une partie de la recherche, la fonction `search_next_sol` est appelée, tout en considérant la séquence spécifiée de décisions de départ, et l'écart de borne spécifié. L'écart Δ est utilisé par `search_next_sol` pour calculer une borne supérieure temporaire B' qui remplace la borne supérieure courante $\overline{B_{\mathcal{P}}}$: on a $B' = \overline{B_{\mathcal{P}}} - \Delta + 1$, forçant alors la contrainte objectif à être $f < B'$ pendant cet appel à `search_next_sol`. Si une nouvelle solution de coût B (nécessairement, $B < B'$) est trouvée par `search_next_sol`, l'appel est arrêté, et la contrainte objectif est mise à jour pour devenir sans risque $f < B$. Sinon, l'appel est arrêté (parce que les limites de cutoff ou de temps sont atteintes), et la contrainte objectif est mise à jour pour devenir $f < \overline{B_{\mathcal{P}}}$ (en récupérant la précédente borne supérieure). Pour résumer, cette fonction met implicitement à jour les bornes d'optimisation avant de retourner la nouvelle séquence de décisions (l'endroit exact où la recherche s'est arrêtée) et un statut (local). Le statut local est soit SAT, auquel cas l'exécution peut se poursuivre avec une nouvelle itération de la boucle, soit une valeur parmi UNSAT, CUTOFF_REACHED et TIMEOUT.

Le run actuel exécute nécessairement certaines vérifications à partir de la ligne 8. À cette ligne, un booléen est défini, nous informant si l'exécution actuelle était sûre ou non, en ce qui concerne la dernière valeur Δ calculée. Lorsque la limite de temps globale est atteinte, le réseau de contraintes et l'état INCOMPLETE sont renvoyés (lignes 9 et 10). Lorsque le statut local notifie UNSAT, nous avons deux cas à considérer. Si la recherche en cours a été effectuée de façon *safe*, COMPLETE peut être retourné car l'espace de recherche est garanti d'avoir été entièrement exploré. Sinon, CONTINUE est retourné avec le réseau de contraintes $\mathcal{P}$ intégrant éventuellement quelques nouvelles contraintes (nogoods). La notation $\mathcal{P} \oplus nld(\Sigma_{i-1})$ indique que tous les nogoods qui peuvent être extraits de l'avant-dernière séquence de décisions (voir [LECOUTRE *et al.* 2006]) sont ajoutés à $\mathcal{P}$; ceci est valable car cette séquence était celle

Algorithme 5 : `run`$(\mathcal{P}, \texttt{abd})$

Output : $(\mathcal{P}, \text{status})$

```
1  i ← 1 ;
2  Σ_i ← ∅ ;
3  faire
4  |  Δ ← abd(i) ;
5  |  i ← i + 1 ;
6  |  Σ_i, status ← search_next_sol(P, Σ_{i−1}, Δ) ;
7  tant que status = SAT ;
8  safe ← Δ = 1 ;
   Le timeout de la résolution est atteint
9  si status = TIMEOUT alors
10 |  retourner (P, INCOMPLETE)
11 fin
   Le statut UNSAT avec Δ = 1
12 si status = UNSAT & safe alors
13 |  retourner (P, COMPLETE)
14 fin
   Le statut UNSAT avec Δ > 1
15 si status = UNSAT & ¬safe alors
16 |  retourner (P ⊕ nld(Σ_{i−1}), CONTINUE)
17 fin
   Le cutoff du run est atteint avec Δ > 1
18 si status = CUTOFF_REACHED & ¬safe alors
19 |  retourner (P ⊕ nld(Σ_{i−1}), CONTINUE)
20 fin
   Le cutoff du run est atteint avec Δ = 1
21 si status = CUTOFF_REACHED & safe alors
22 |  retourner (P ⊕ nld(Σ_i), CONTINUE)
23 fin
```

correspondant à la dernière solution trouvée. Lorsque l'état local notifie CUTOFF_REACHED, on peut aussi continuer, tout en considérant l'ajout de quelques nogoods de redémarrage, à partir de Σ_i ou Σ_{i-1}.

Nous concluons cette section par deux remarques. Premièrement, l'algorithme est introduit dans le contexte d'un schéma d'enregistrement de nogoods légers (seuls sont considérés les nogoods qui peuvent être extraits de la branche la plus à droite, lorsque la recherche est temporairement arrêtée). Cependant, il est possible de l'adapter à d'autres schémas d'apprentissage, en gardant la trace du moment exact où un nogood (clause) est inféré ; ceci est purement technique. Deuxièmement, il y a un cas spécifique concernant l'insatisfaisabilité : si jamais nous rencontrons une situation où $B' \leq \underline{B_{\mathcal{P}}}$ en essayant de fixer une nouvelle borne supérieure temporaire B' pendant l'exécution actuelle, la suite est réinitialisée en forçant le retour de $i = 1$ et B' est recalculé.

7.3.2 Travaux connexes

Recherche dichotomique de la borne Dans [STREETER & SMITH 2007], une approche connexe a été proposée : l'étude tente de faire converger le solveur d'optimisation plus rapidement vers des solutions de bonne qualité en appliquant de fortes restrictions sur les limites. Particulièrement dans cet article, une forme générale de stratégie de requête est définie, accompagnée de garanties théoriques, et dont quelques stratégies spécifiques sont décrites. Ces stratégies correspondent à une forme de restriction dichotomique faite en considérant les limites connues (*min* et *max*) ainsi que d'autres limites liées au temps (*t_min* et *t_max*) correspondant aux bornes pour lesquels un trop important budget de temps a été alloué. Dans notre cas, bien que nous prenons en compte la limite *min*, nous n'accordons pas autant de temps de résolution pour cette limite (cela nécessite un budget supplémentaire conséquent) et nous essayons de nous concentrer sur la limite *max* ; dans les faits, nous n'utilisons pas de mécanisme dichotomique pour trouver une nouvelle limite, mais plutôt des séquences (exponentiellement croissantes) pour mettre à jour rapidement la limite *max*. Par ailleurs, le solveur implantant ABD n'a pas besoin de modifier sa fonction de redémarrage car la fonction de redémarrage et la fonction `abd` sont indépendantes (et donc, le mécanisme ABD peut être considéré comme moins intrusif), là où le solveur de l'étude en question impose un budget à chaque requête.

Enfin, dans [STREETER & SMITH 2007], les auteurs concentrent principalement leur expérimentation sur des problèmes spécifiques d'ordonnancement alors qu'ici nous testons un large éventail de problèmes COP ; alors que des appels sont faits à un solveur SAT dans [STREETER & SMITH 2007], nous intégrons directement ABD dans un solveurs CP.

Branchement par division du domaine Comme autre approche connexe, nous pouvons aussi mentionner la division de domaine (par exemple [VAN HOEVE & MILANO 2004]) dont le rôle est de partitionner le domaine de la variable sélectionnée par l'heuristique de choix de variable, et de brancher sur les sous-domaines résultants. Lorsque la fonction objectif est simplement représentée par une variable autonome (dont la valeur doit être minimisée ou maximisée), une approche de division de domaine peut être réglée pour simuler une descente de borne agressive. Cependant, aucun contrôle n'est possible car aucun mécanisme ne permet d'abandonner des choix trop optimistes, contrairement à ABD qui est bien intégrée dans une politique de redémarrage. De plus, lorsque la fonction objectif a une forme plus générale qu'une variable (comme par exemple une somme ou une valeur minimum/maximum à calculer), il n'existe pas de correspondance directe (et introduire systématiquement une variable auxiliaire pour représenter l'objectif peut être très intrusif, voire source d'inefficacité, pour le solveur).

Heuristique primaire Enfin, notons que la question d'éviter les longues séquences de solutions qui s'améliorent lentement a été abordée en MIP (*Mixed-Integer Programming*) par l'introduction d'heuristiques primaires, qui visent à trouver et à améliorer les solutions réalisables au début du processus de résolution.

7.3.3 Expérimentations

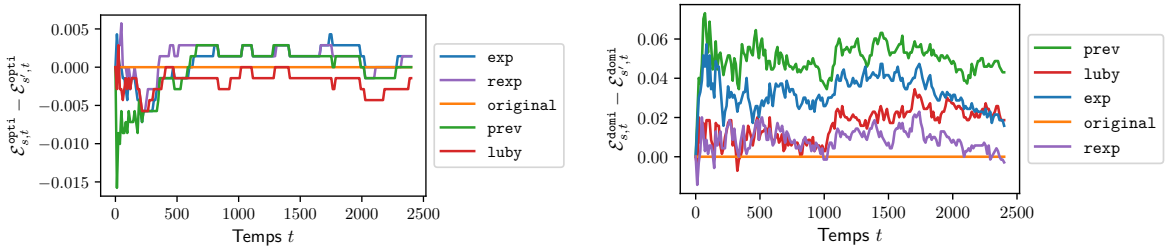
Cette section présente quelques résultats expérimentaux concernant l'approche ABD sur un large éventail de problèmes d'optimisation ($\mathcal{I}_{\texttt{COP}}$). L'approche a été implantée dans le solveur `ACE`. Dans le cas de ce solveur, le retour à un état historique de l'apprentissage n'est pas possible : certains mécanismes d'apprentissage au niveau des contraintes globales ne permettent pas de retour « simple » à l'état précédent. Ainsi, contrairement à l'implantation proposée plus haut, `ACE` conserve l'apprentissage de fin de run peu importe l'état de la variable *safe*. Si, à un moment donné, le solveur trouve une borne insatisfaisable et que la résolution est actuellement *unsafe*, celui-ci réinitialise entièrement le réseau de

contraintes (le domaine des variables est restauré et la base de nogoods est vidée) pour s'assurer de la véracité des dernières bornes trouvées par le solveur *unsafe*. Il est probable que cette implantation rende le processus agressif moins efficace.

Afin de juger de l'efficacité de cette nouvelle proposition de progression agressive de la borne et de l'implantation qu'il en est faite sur ACE, nous proposons l'ensemble Ψ_{16} comme campagne expérimentale :

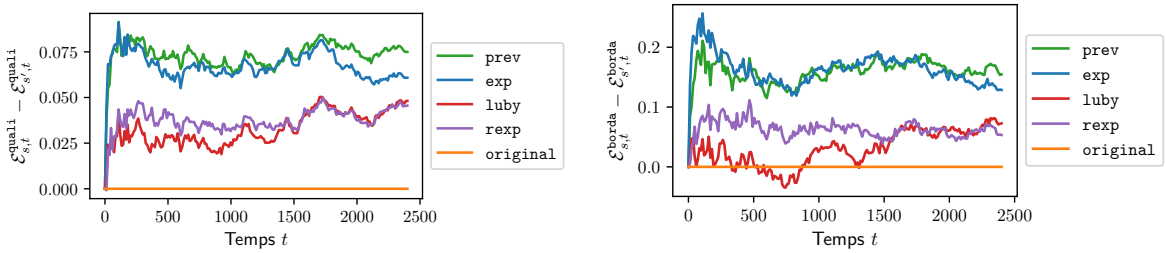
$$\left\{ \mathtt{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(\mathtt{abd}^{f}) \;\middle|\; f \in \{\mathtt{exp}, \mathtt{rexp}, \mathtt{luby}, \mathtt{prev}\} \right\} \tag{Ψ_{16}}$$

Cet ensemble étant maintenant défini, celui-ci est expérimenté sur l'ensemble $\mathcal{I}_{\mathtt{COP}}$ d'instances, dont la figure 7.2 propose une vue des nouvelles performances des solveurs implantant ABD et ses différentes suites, en comparaison avec le solveur par défaut (sans implantation d'ABD). La figure 7.2 met à disposition les quatre types d'évaluations (optimalité, dominance, qualité des solutions et classement de Borda) proposés plus haut.



(a) Différence de proportion d'*optimalité* avec le solveur $s' = \mathtt{original}$

(b) Différence de proportion de *dominance* avec le solveur $s' = \mathtt{original}$



(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \mathtt{original}$

(d) Différence du classement moyen de *Borda* avec le solveur $s' = \mathtt{original}$

FIGURE 7.2 – Comparaison des différentes politiques de descente agressive de borne [Ψ_{16}, $\mathcal{I}_{\mathtt{COP}}$, ♌]

Une première remarque est que le caractère optimal ne s'en retrouve que peu changé avec la politique ABD. Ceci n'est pas si étonnant car, la particularité de la politique ABD est d'être agressive avec des bornes dont la progression est lente. À l'approche de la borne optimale, la recherche demande au contraire une progression plus fine afin de ne pas dépasser la borne et rechercher inutilement du côté UNSAT de l'espace de recherche. Il est donc rassurant de voir à travers la figure 7.2a que cette optimalité ne fluctue, en moyenne, que d'à peine un demi pour-cent par rapport au solveur par défaut. Néanmoins, nous remarquons que la politique basée sur `prev` chute à -1.5% sur les premiers temps d'exécution.

Les graphiques sur la dominance et la qualité des solutions trouvées (figures 7.2b et 7.2c) nous montrent plus précisément l'impact de la politique agressive. Cette fois-ci, les politiques implantant ABD

sont constamment supérieur au solveur par défaut, notamment les deux suites à progression exponentielle : `exp` et `prev`. Ces graphiques sur la dominance et la qualité des bornes montrent respectivement, pour les meilleures politiques, entre 4% à 6% de performance supplémentaire et 6% à 8%.

La dernier graphique, prenant en compte globalement ces trois précédents critères, met aussi en avant les deux précédentes politiques ABD : `exp` et `prev`. Nous remarquons que ces deux politiques se placent, en moyenne, entre 0.1 et 0.2 place au-delà du solveur par défaut.

7.3.4 Conclusion et perspective de la politique ABD [11]

Bien que les performances ne dépassent pas les dix pour-cent dans ces expérimentations d'ABD, celles-ci semblent tout de même proposer une plus grande robustesse du solveur. Sous d'autres configurations telles que dans l'article [FALQUE *et al.* 2021a], où `ACE` conserve son comportement initial ($\texttt{ACE}^{\texttt{def}}_{\texttt{lc}=2}$), la progression est plus flagrante et progresse au-delà de 10%. Toujours dans cette étude, ABD est implanté dans le solveur Pseudo-Booléen `Sat4j` [LE BERRE & PARRAIN 2010], avec une gestion plus fine du caractère *safe* (grâce au principe de selecteur permettant de sélectionner simplement les connaissances à conserver ou non), et montre des performances montant jusque 20% par rapport au `Sat4j` par défaut. Dans l'ensemble de ces cas, nous remarquons une grande robustesse sur les premiers temps d'exécution (les 500 premières secondes), puis la configuration par défaut a tendance à rejoindre la configuration implantant ABD. Cet effet est explicable par le fait que ces deux types de progression sont toutes deux soumises à une amélioration logarithmique de la borne (une grande amélioration de la borne est possible en début de recherche, puis réduite avec le temps) : ce logarithme est plus agressif pour ABD mais subit fatalement un ralentissement, là où la progression logarithmique de la solution par défaut est plus douce et joint doucement la progression logarithmique d'ABD.

Quelques perspectives s'offrent à nous concernant l'état actuel de la politique ABD et l'application de la dichotomie telle que proposée dans [STREETER & SMITH 2007]. Cette étude introduit un langage de requête à un solveur SAT : « Existe-t-il une solution dont le coût est inférieur à B' ? ». À mi-chemin entre cette politique et celle d'ABD souhaitant préserver un maximum de budget de recherche côté bornes satisfaisables et au sein d'une même résolution, il pourrait être intéressant de lancer une requête dérivée de celle-ci, lors d'un run à budget fixé par la suite de redémarrage (Luby, par exemple) : « Sachant la progression de la borne depuis le début de la recherche et le budget du futur run, à combien fixe-t-on, de façon raisonnable, la prochaine borne B' ? ».

7.4 Apprentissage et perturbation d'heuristique

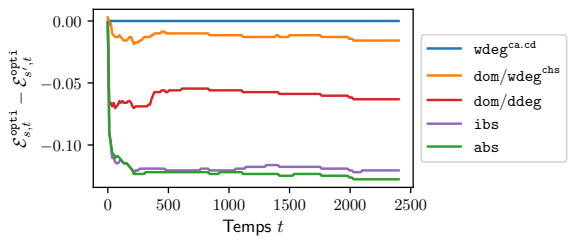
Nous revenons maintenant au sujet principal de ce manuscrit, en appliquant les résultats les plus prometteurs des précédents chapitre à celui-ci se focalisant sur les problèmes d'optimisation. Nous évaluons, dans un premier temps, l'ensemble $\mathcal{H}^{\texttt{base}}$ des heuristiques que nous nous proposons comme base de comparaison et comme ensemble d'heuristiques pour les politiques multi-heuristiques. Une fois ce point de vue global observé, nous appliquons les politiques, ayant eu les meilleurs bénéfices, pour l'apprentissage de la meilleure heuristique dans le cadre de la résolution sous optimisation. Enfin, nous essayons d'améliorer chacune des heuristiques à l'unité grâce aux politiques de perturbation.

7.4.1 Évaluation de $\mathcal{H}^{\texttt{base}}$

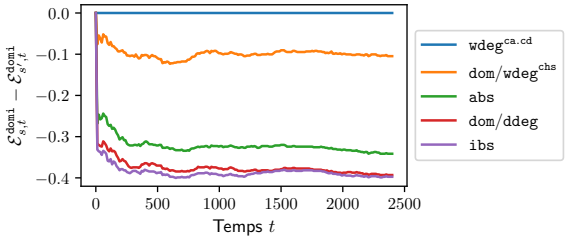
Commençons par comparer les heuristiques provenant de l'ensemble $\mathcal{H}^{\texttt{base}}$. Soit l'ensemble de solveurs Ψ_{17} :

11. Quand les lettres du sigle $\mathrm{A_1B_2D_4}$ sont indicées par leur position dans l'alphabet, nous pouvons remarquer la même progression (exponentielle) que certaines des suites exploitées par… ABD elle-même !

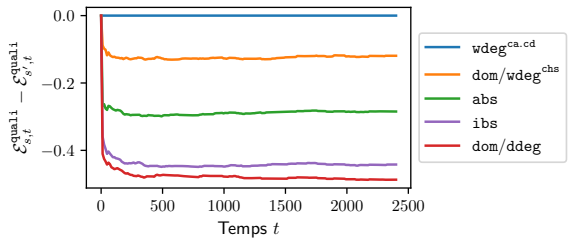
$$\left\{ \mathrm{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(h) \;\middle|\; h \in \mathcal{H}^{\mathtt{base}} \right\} \tag{Ψ_{17}}$$



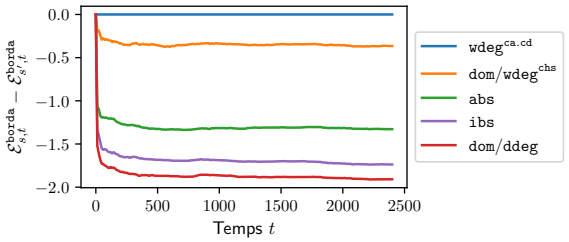
(a) Différence de proportion d'*optimalité* avec le solveur $s' = \mathtt{wdeg}^{\mathtt{ca.cd}}$



(b) Différence de proportion de *dominance* avec le solveur $s' = \mathtt{wdeg}^{\mathtt{ca.cd}}$



(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \mathtt{wdeg}^{\mathtt{ca.cd}}$



(d) Différence du classement moyen de *Borda* avec le solveur $s' = \mathtt{wdeg}^{\mathtt{ca.cd}}$

FIGURE 7.3 – Comparaison des heuristiques de $\mathcal{H}^{\mathtt{base}}$ [Ψ_{17}, $\mathcal{I}_{\mathtt{COP}}$, ↻]

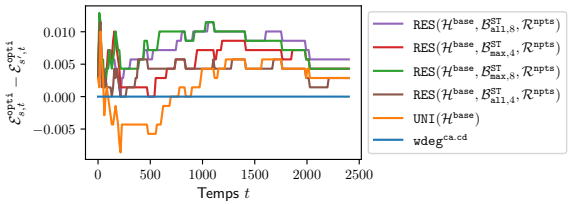
La campagne correspondant aux expérimentations de Ψ_{17} sur les instances $\mathcal{I}_{\mathtt{COP}}$ est représentée par la figure 7.3. À l'image des résultats que nous avons jusqu'ici, les heuristiques de l'ensemble $\mathcal{H}^{\mathtt{base}}$ se comportent et se classent de la même manière. Nous observons les deux meilleures heuristiques $\mathtt{wdeg}^{\mathtt{ca.cd}}$ et $\mathtt{dom/wdeg}^{\mathtt{chs}}$constamment aux deux meilleures places, peu importe le type d'évaluation. Nous voyons aussi que le caractère dynamique de l'heuristique `dom/ddeg` lui permet de bonnes performances en optimalité, mais récupère la dernière place pour les autres évaluations. `abs` et `ibs` ont des résultats intermédiaires.

7.4.2 Apprentissage de la meilleure heuristique

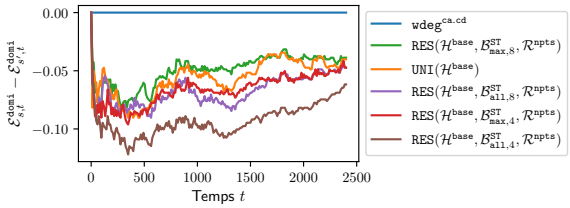
Dans cette section, nous nous intéressons à l'application du bandit ayant proposé les résultats dont le compromis entre efficacité et régularité était le meilleur — `ST`, accompagné de la fonction de récompense `npts`. Soit l'ensemble de solveurs Ψ_{18} :

$$\left\{ \mathrm{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(\mathtt{RES}(\mathcal{H}^{\mathtt{base}}, \mathcal{B}^{\mathtt{ST}}_{d,x}, \mathcal{R}^{\mathtt{npts}})) \;\middle|\; \begin{array}{l} d \in \{\mathtt{duel}^{\mathtt{all}}, \mathtt{duel}^{\mathtt{max}}\} \\ x \in \{4, 8\} \end{array} \right\} \tag{Ψ_{18}}$$

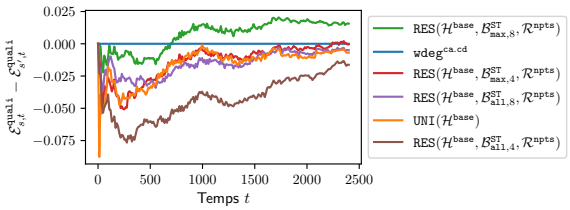
La figure 7.4 nous informe de la qualité des différents solveurs de Ψ_{18}. Globalement, nous remarquons qu'il semble compliqué de battre la meilleure heuristique (dont le solveur associé est choisi comme témoin dans la comparaison) dans cette campagne sous le thème de l'optimisation. La stratégie uniforme nous informe que cette fois-ci et sans apprentissage, il n'est pas possible d'égaler la meilleure heuristique. Les résultats de l'évaluation basée sur la proportion d'optimalité semble rapporter de nouveaux



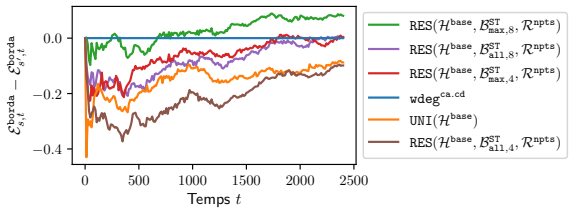
(a) Différence de proportion d'*optimalité* avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$



(b) Différence de proportion de *dominance* avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$



(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$



(d) Différence du classement moyen de *Borda* avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$

FIGURE 7.4 – Comparaison des politiques de sélection des heuristiques de $\mathcal{H}^{\texttt{base}}$ [$\Psi_{18}, \mathcal{I}_{\texttt{COP}}, \circlearrowleft$]

résultats négligeables par rapport à ce que propose le solveur témoin, il n'est donc pas intéressant de prendre en compte ce critère.

Pour ce qui est du critère de dominance, il est à notifier qu'il existe un biais dans cette mesure : lorsque des stratégies possèdent des caractéristiques équivalentes entre elles et qu'une autre possède un comportement bien à elle, et bien les stratégies au comportement similaire vont se faire concurrence pour récupérer les points de dominance, tandis que la stratégie au comportement à part récupère la dominance qu'elle acquiert naturellement. Pour cette mesure, il serait bien plus intéressant de comparer les solveurs par paires, plutôt que globalement. Les évaluations basées sur les critères de qualité et de classement de Borda ne sont pas sujets à ce biais : l'un car il compare un critère continu (la qualité d'une solution est un nombre réel entre dans $[0; 1]$ alors que la dominance est binaire) et l'autre car il agrège des comparaisons par paire.

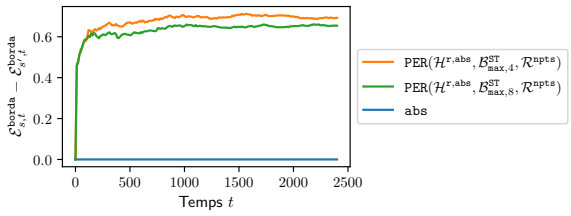
Les deux derniers critères semblent les plus appropriés pour pouvoir juger bon de la qualité des solveurs. Nous remarquons globalement que les solveurs apprenant la meilleure heuristique ont l'air particulièrement, et naturellement, plus en difficulté en début de résolution par rapport à l'heuristique seule $\texttt{wdeg}^{\texttt{ca.cd}}$. Cela se comprend car il est nécessaire d'explorer avant de commencer à exploiter la meilleure heuristique. Aux environs du timeout, les stratégies multi-heuristiques semblent tout de même rejoindre au moins les performances de la meilleure heuristique, mis-à-part les stratégies uniformes et celle basée sur l'opérateur $\texttt{duel}^{\texttt{all}}$ et l'échantillonnage fixé à 4, se rapprochant plus lentement de $\texttt{wdeg}^{\texttt{ca.cd}}$. La stratégie $\texttt{RES}(\mathcal{H}^{\texttt{base}}, \mathcal{B}^{\texttt{ST}}_{\texttt{max},8}, \mathcal{R}^{\texttt{npts}})$ semble tout de même légèrement se démarquer.

Il semblerait que cette campagne démarque notamment la qualité supérieure de l'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$ étant probablement proche de la politique optimale à suivre, bien que la meilleure politique multi-heuristique montre que des améliorations sont toujours possibles. Dans cette campagne proposant des instances sous optimisation, seul le défi de rejoindre la meilleure heuristique a été relevé.

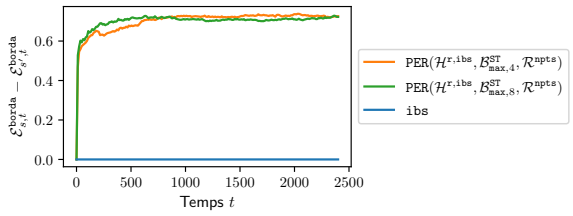
7.4.3 Perturbation d'heuristique

Pour cette dernière section expérimentale et dans la même idée que la précédente section, nous reprenons les politiques proposant les meilleures performances dans la résolution de CSPs concernant la perturbation et la diversification d'heuristique. Soit l'ensemble de solveurs Ψ_{19} :

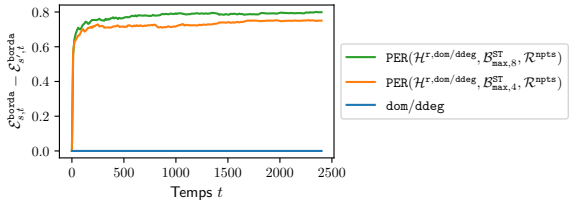
$$\left\{ \mathtt{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(\mathtt{PER}(\mathcal{H}^{r,h}, \mathcal{B}^{\mathtt{ST}}_{\mathtt{max},k}, \mathcal{R}^{\mathtt{npts}})) \;\middle|\; \begin{array}{l} h \in \mathcal{H}^{\mathtt{base}} \setminus \mathtt{wdeg}^{\mathtt{ca.cd}} \\ k \in \{4, 8\} \end{array} \right\} \cup \left\{ \mathtt{ACE}^{\mathtt{luby}}_{\mathtt{lc}=0}(\mathtt{RES}(\mathcal{H}^{tb_5(\mathtt{wdeg}^{\mathtt{ca.cd}})}, \mathcal{B}^{\mathtt{ST}}_{\mathtt{max},k}, \mathcal{R}^{\mathtt{npts}})) \;\middle|\; k \in \{4, 8\} \right\} \tag{Ψ_{19}}$$



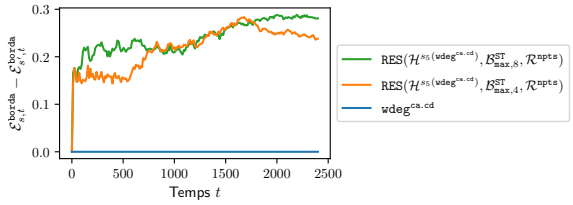
(a) Différence du classement moyen de *Borda* avec le solveur $s' = \mathtt{abs}$



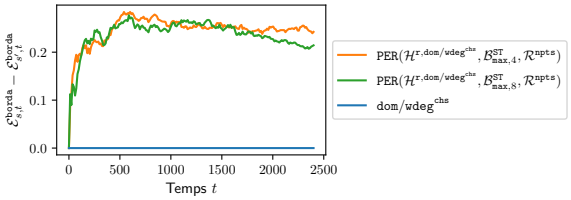
(b) Différence du classement moyen de *Borda* avec le solveur $s' = \mathtt{ibs}$



(c) Différence du classement moyen de *Borda* avec le solveur $s' = \mathtt{dom/ddeg}$



(d) Différence du classement moyen de *Borda* avec le solveur $s' = \mathtt{wdeg}^{\mathtt{ca.cd}}$



(e) Différence du classement moyen de *Borda* avec le solveur $s' = \mathtt{dom/wdeg}^{\mathtt{chs}}$

FIGURE 7.5 – Comparaison des politiques de perturbation pour chaque heuristique de l'ensemble $\mathcal{H}^{\mathtt{base}}$ [Ψ_{19}, $\mathcal{I}_{\mathtt{COP}}$, ↻]

La figure 7.5 est un résumé représentatif des résultats présentés plus exhaustivement dans l'annexe B.2. Dans cette figure sont présentés cinq graphiques représentant les effets de la perturbation sur les cinq heuristiques de l'ensemble $\mathcal{H}^{\mathtt{base}}$. Chaque graphique représente la différence de rang moyen entre les heuristiques d'une même famille et son témoin — l'heuristique d'origine. Dans le cadre des COPs, les conclusions sont les mêmes que pour le cadre des CSPs : les perturbations améliorent considérablement

l'heuristique dynamique `dom/ddeg` (près de 0.8 place supplémentaire en moyenne) et, avec une moins grande importance, les heuristiques `abs` et `ibs` (plus de 0.6 place supplémentaire en moyenne). Les heuristiques les plus efficaces sont elles aussi améliorées par ces perturbations en gagnant entre 0.2 et 0.3 place en moyenne. Les deux échantillonnages proposés apportent tous deux des performances équivalentes.

7.5 Conclusion

Ce chapitre nous apprend que le framework basé sur le bandit `ST` et la fonction de récompense `npts` résiste à l'extension aux problèmes de contraintes sous optimisation en proposant des performances, au moins, équivalentes à la meilleure heuristique (dans le cas de l'apprentissage de la meilleure heuristique) et réussit aussi à améliorer la qualité de résolution des solveurs en perturbant l'heuristique choisie par l'utilisateur. Ainsi, le framework actuel semble suffisamment robuste pour répondre à la question de l'autonomie dans les cas de résolution de CSPs comme de COPs, voire même au-delà des performances de la meilleure heuristique.

La résolution des COPs étant une extension du principe de résolution des CSPs, les perspectives en optimisation héritent directement des précédentes perspectives. Néanmoins, nous pouvons imaginer certaines études plus spécifiques dans ce cadre. Par exemple, il pourrait être intéressant de tester de nouvelles fonctions de récompense plus en lien avec la fonction objectif : prendre en compte la qualité de la solution obtenue, l'écart avec la nouvelle borne obtenue, le nombre de nouvelles bornes obtenues, *etc.* Le nombre de nouvelles bornes obtenues peut être trompeur. Nous l'expliquions dans l'aparté sur la descente agressive de borne (ABD) : lors d'un run donné, les solutions obtenues peuvent être très proches les unes des autres et ne proposent, à la fin du run, qu'une amélioration minime de la borne. Cela dit, cette fonction de récompense peut entrer en harmonie avec l'usage d'ABD : étant donné qu'ABD oblige la borne à grandir exponentiellement à chaque itération, le comptage de nouvelles bornes devient aussi informatif que le calcul de l'écart entre la nouvelle borne obtenue et l'ancienne.

Conclusion Générale

Les deux premiers chapitres de ce manuscrit présentent l'état de l'art de deux concepts distincts de l'intelligence artificielle : la *programmation par contraintes* [MONTANARI 1974, APT 2003, ROSSI *et al.* 2006, LECOUTRE 2009] et le *problème du bandit multi-bras* [GITTINS 1989]. Dans les chapitres suivants, nous traitons de différents sujets mettant en avant la problématique de ce manuscrit et apportant des contributions directes et indirectes à celle-ci.

Le chapitre 1 introduit la programmation par contraintes. Il y est expliqué que ce langage appartient au paradigme déclaratif de la programmation et que son utilisation est séparée en deux parties : la *modélisation* et la *résolution*. La modélisation donne lieu à la production d'un *problème de satisfaction de contraintes* et la résolution, mettant en scène un *solveur de contraintes*, tente de résoudre ce problème en trouvant une *solution*. Le but de la programmation par contraintes est d'abstraire l'utilisateur du langage machine en exprimant le problème dans un langage proche du langage naturel. La production d'une solution se fait indépendamment grâce à l'usage du solveur. Par extension de ce problème, nous introduisons aussi le *problème d'optimisation sous contraintes*. Malgré la volonté d'abstraire l'utilisateur de toute implication à la résolution de son problème, les solveurs de contraintes actuels ont la nécessité d'être convenablement configurés afin de résoudre efficacement un problème donné. Dans ce manuscrit, nous nous sommes essentiellement concentrés sur l'un des mécanismes clés des solveurs de contraintes : les heuristiques de choix de variable. Une heuristique finement sélectionnée permet une résolution efficace. Ainsi, la problématique suivante a été formulée : « Comment trouver la meilleure heuristique de choix de variable pour une instance de problème, étant donné un ensemble d'heuristiques fournies par le système de résolution ? ».

Toujours dans la description de la bibliographie, le chapitre 2 introduit le problème du bandit multi-bras. Ce problème s'inscrit dans le domaine de l'*apprentissage par renforcement*. Dans le cas où un joueur fait face à un ensemble d'actions, dont la probabilité de gain est fixe mais inconnue de celui-ci, il fait face au dilemme exploration-exploitation de ces actions afin de maximiser son gain sur une séquence de jeux. Il s'agit de rechercher un équilibre entre l'exploration de l'environnement — afin de trouver des actions profitables, tout en exploitant le plus souvent possible la, supposée, meilleure action. À ce même titre, la résolution des solveurs de contraintes est morcelée en plusieurs *runs* dirigée par un mécanisme de redémarrage de la recherche. Dans ce manuscrit, ce mécanisme laisse place, à chaque début de run, à une politique répondant au précédent problème en sélectionnant une heuristique parmi un ensemble d'heuristiques disponibles. Ainsi, au fil des runs, la politique du bandit explore chacune des heuristiques et essaye d'exploiter le plus souvent celle produisant la meilleure avancée dans le processus de résolution globale.

Études annexes à la problématique

Tout d'abord, ce manuscrit a été l'occasion de présenter *Metrics* [FALQUE *et al.* 2020, FALQUE *et al.* 2021b], une librairie tentant d'uniformiser les moyens d'extraction et d'analyse des solveurs de contraintes et, plus généralement, tout solveur ou logiciel nécessitant de produire des analyses en lien

avec le temps d'exécution basés sur un ensemble d'entrées (instances dans le cas des solveurs). Chaque campagne produite dans ce manuscrit est gérée par l'outil *Metrics* permettant de simplifier la présentation des expériences tout en donnant la possibilité, au lecteur, d'observer les analyses et de les reproduire. Actuellement, cet outils se focalise sur la réussite ou l'échec de la résolution de fichiers en entrée (prenant en compte le temps), mais s'efforce d'évoluer afin de prendre en compte un plus large panel de paradigmes de résolution ; comme nous pouvons le constater dans les campagnes en optimisation, une grande partie de l'analyse est décrite à l'extérieur de l'outil *Metrics* et sera, par la suite, implantée au sein de la librairie. Cet outils tend donc à élargir sa prise en compte de paradigmes de résolution tout en diversifiant son panel d'analyses, tableaux et graphiques associés.

Ensuite, la recherche de fonction de récompense capable de rendre compte de l'efficacité d'une heuristique nous a amené à proposer l'affinement de l'une d'entre elles : $\texttt{wdeg}^{\texttt{ca·cd}}$ [WATTEZ *et al.* 2019b, WATTEZ *et al.* 2021a]. En effet, la recherche d'une métrique sur l'état de la recherche pour produire une fonction de récompense suffisamment informative à l'échelle d'un run nous a menée à l'affinement de l'information prélevée à chaque conflit pour l'heuristique basée sur la pondération des contraintes rendant le solveur ACE d'autant plus robuste en terme de satisfaction qu'en terme d'optimisation. La formulation générique de $\texttt{wdeg}^{\alpha}$ (heuristique VI) annonce de possibles perspectives quant à de futurs affinements de l'information apportée à chaque conflit du solveur. Cette formulation peut être rendue plus généraliste en prenant en compte les principes de vieillissement de l'une des extensions de l'heuristique à pondération de contraintes : $\texttt{wdeg}^{\texttt{chs}}$ [CHERIF *et al.* 2020b].

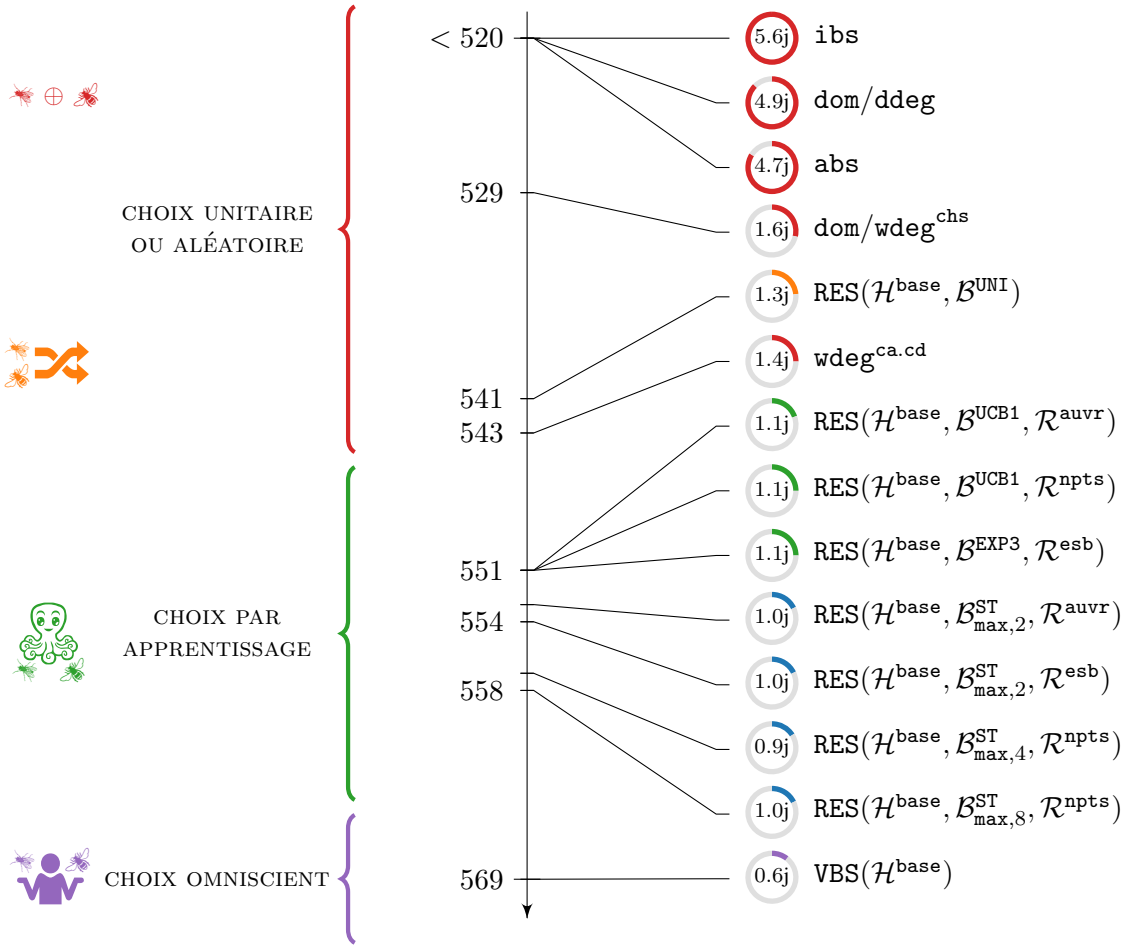
Enfin, l'étude de la problématique étendue aux problèmes d'optimisation nous a aussi donné l'occasion d'étudier une forme agressive de descente de borne (ABD) [FALQUE *et al.* 2021a]. À l'origine, cette étude nous a démontré de bonnes performances sur le solveur ACE par défaut, notamment sur les premiers temps d'exécution. Légèrement plus faible dans la version adaptée d'ACE pour ce manuscrit, la solution ABD nous a tout de même apporté les mêmes tendances améliorant la qualité de la résolution sous optimisation. Au-delà du paradigme de la programmation par contraintes, ABD a fait l'objet d'une implantation dans un solveur pseudo-Booléen ayant montré de bonnes performances à son tour. En perspective, nous pensons ce mécanisme exportable à d'autres paradigmes de résolution toujours sous le thème de l'optimisation.

Apprentissage de la meilleure heuristique

Le chapitre 3 est une introduction à l'évaluation des solveurs de contraintes mettant en avant la problématique de ce manuscrit. Dans ce chapitre, nous avons observé la disparité entre les différentes heuristiques proposées par le solveur de contraintes ACE. Cette disparité et la problématique associée, laisse place à différents défis que nous rappelons : le premier (i), et le plus crucial d'entre eux, est de gagner en autonomie en obtenant, a minima, les performances de la meilleure heuristique de l'ensemble mis à disposition par le solveur ; le deuxième défi (ii) est de dépasser les capacités de la meilleure heuristique ; le dernier défi (iii), plus utopique, est d'être équivalent ou supérieur au choix optimal, c'est-à-dire, au choix qu'un oracle, connaissant à l'avance les performances des heuristiques (en fonction des instances), donnerait.

Dans un premier temps, ce chapitre a été l'occasion de proposer de nouvelles fonctions de récompense capable de rendre compte de l'efficacité d'un run lors de la résolution du solveur de contraintes : les fonctions esb [PAPARRIZOU & WATTEZ 2020] et npts [WATTEZ *et al.* 2020] découlent de cette première étude. À celles-ci, nous avons emprunté et ajouté une autre fonction de récompense issue de la bibliographie : auvr [CHERIF *et al.* 2020b]. Nous avions remarqué que le comportement médian de ces fonctions relataient assez fidèlement de l'efficacité globale des heuristiques. Dans certains cas, nous voyions aussi qu'il pouvait être intéressant de prendre en compte les valeurs maximales produites par ces fonctions de récompense.

Dans un second temps, le chapitre 4 a été une première occasion d'inclure les politiques répondant au problème du bandit multi-bras dans le cadre des solveurs de contraintes et de la sélection automatique des heuristiques de choix de variable [WATTEZ *et al.* 2020]. Le chapitre 5 est une nouvelle contribution aux politiques de bandit répondant avec une meilleure adaptation aux difficultés rencontrées avec les précédentes implantations. Afin de se rendre compte de l'efficacité de ces différentes politiques dans le cadre des solveurs de contraintes, la figure 6 résume les principaux résultats concernant l'apprentissage de la meilleure heuristique. Parallèlement à cette figure, nous schématisons les liens créés, lors de l'introduction générale, avec l'analogie de *la mouche et l'abeille*.



(a) Analogie de la mouche et l'abeille

(b) Nombre de résolutions et temps de résolution des solveurs associés aux stratégies mono et multi-heuristiques

FIGURE 6 – La der' des der' : liens entre l'analogie de la mouche et l'abeille, et l'apprentissage de la meilleure heuristique [sections 4.4.3 et 5.3.1]

Cette figure place sur un axe pointant vers le bas, le nombre croissant de résolutions produit par les principales politiques de cette étude sur l'ensemble $\mathcal{I}_{\text{CSP}}$ d'instances : partant des heuristiques à l'unité (en rouge) jusqu'au meilleur solveur (virtuel) en violet servant de solveur témoin possédant les mêmes capacités qu'un oracle. Entre deux se situent les politiques multi-heuristiques, dont le choix uniforme (en orange), celles implantant les bandits de la bibliographie (en vert) et le nouveau bandit adapté au contexte des solveurs de contraintes (en bleu). À chaque politique est associée une autre métrique correspondant au nombre de jours d'exécution pour venir à bout de la campagne : les instances qu'aucun solveur n'a

réussi à résoudre sont ignorées dans la comptabilisation de ce temps.

Tout en haut de l'axe de résolution, nous observons les heuristiques d'origine : de la plus faible à la plus robuste partant de moins de 400 résolutions jusque 543 résolutions. Non-loin de cette extrémité représentée par la meilleure heuristique, nous observons que la politique uniforme se place à 2 instances de cet optimum, réussissant à relever le premier défi qu'était d'égaler la meilleure heuristique de l'ensemble proposé par le solveur. En lien avec l'analogie de la mouche et l'abeille, nous retrouvons les mêmes résultats obtenus par l'expérience (de pensée) des deux heuristiques appartenant à ces deux insectes. Le choix aléatoire et uniforme, de façon séquentielle, entre les deux insectes produisaient un résultat semblable à la sélection du meilleur d'entre eux.

L'implantation des bandits de l'état de l'art (chapitre 4) a montré de nouvelles performances par rapport à la meilleure heuristique et la politique de choix uniforme. À partir de ces nouvelles performances, nous pouvons considérer le deuxième défi comme accompli et le troisième défi comme entamé : avec ces classiques bandits, nous sommes maintenant à 30% du chemin allant de la meilleure heuristique vers la politique optimale et utopique : le `VBS`. Dans l'analogie de la mouche et l'abeille, nous avions proposé de prendre en compte, à intervalle régulier, la distance séparant l'insecte/heuristique sélectionné par rapport au goulot de la bouteille afin de se rendre compte de la progression de chacun. Cet apprentissage [12] apporte une information conséquente et montre une efficacité accrue à s'adapter et à résoudre les différentes instances de problème.

La prise en compte des différents problèmes d'équité entre runs (de différentes tailles, notamment dans le cas du mécanisme de redémarrage suivant la suite de Luby), d'interprétation des distributions de récompense et de l'efficacité naturelle du choix uniforme nous a permis de produire un nouveau modèle de bandit plus adapté au contexte des solveurs de contraintes suivant la suite de Luby (chapitre 5). Cette suite, une fois réinterprétée en un tournoi sous forme d'arbre binaire et produisant des duels entre heuristiques, a permis une nouvelle progression : nous remarquons dans la figure 6 que nous sommes maintenant à plus de mi-chemin du solveur optimal avec un temps de résolution passant sous la barre d'une journée.

Pour le dernier défi de ce manuscrit, nous retrouvons, seul, le solveur virtuel et optimal. Le `VBS` est considéré comme le choix omniscient dans le cas de l'analogie entre la mouche et l'abeille. En effet cette politique, dans le cas de l'analogie tout comme dans le cas des solveurs de contraintes, nécessite une connaissance *a posteriori* d'expériences.

Dans le chapitre 7, nous avons tenté une extension de ces travaux vers les solveurs de contraintes sous optimisation. Une première remarque sur cette étude fut que la stratégie uniforme était ici bien moins efficace que dans le cas précédent, ne réussissant pas à décrocher le premier défi qu'est l'autonomie et la capacité de rejoindre les performances de la meilleure heuristique. Les politiques basées sur un bandit ont, elles, réussi à dépasser légèrement les capacités de la meilleure heuristique. Il semble que dans le cas de l'optimisation, la disparité entre heuristiques est bien plus importante que dans le cas précédent. Cela expliquerait le comportement actuel des politiques multi-heuristiques. Certaines perspectives, visant à mieux spécifier la fonction de récompense en se rapprochant de la fonction objectif de ce genre de problème, pourraient vraisemblablement améliorer leurs capacités.

Au-delà de l'amélioration de la politique de sélection actuelle ou de la fonction de récompense, nous nous intéressons à une perspective d'extension des travaux actuels. Comme il a été dit dans la conclusion du chapitre 5, la capacité des heuristiques apprenantes (notamment $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca.cd}}$) est très versatile : d'un run à un autre, l'ordonnancement de leurs variables peut fortement fluctuer, provoquant ainsi de fortes fluctuations dans leur capacité à produire des conflits rapidement. Une idée serait de considérer l'état historique de l'heuristique plutôt qu'uniquement son état final. Ainsi, si l'état historique

12. La pieuvre, visible dans la figure 6, est une représentation commune du problème du bandit multi-bras : chacune de ses tentacules est interprétée comme un bras pouvant actionner le levier d'une des machines à sous disponibles.

d'une heuristique a produit une très bonne récompense lors d'un run de budget x, il semblerait plus judicieux de proposer cette heuristique dans ce même état historique au run suivant de taille $2x$ plutôt que l'heuristique dans son état actuel.

Perturbation d'heuristique

En lien avec l'usage initial de l'analogie de la mouche et l'abeille, le chapitre 6 illustre bien ce principe. Là où l'abeille, correspondant à une heuristique (non-aléatoire) choisie par l'utilisateur, possède une philosophie/politique assez fixe et adoptant possiblement un comportement l'orientant vers la mauvaise direction, la mouche correspondant à l'heuristique de choix aléatoire, peut se débrouiller pour se frayer un chemin par l'aléa de son parcours. En plus de ce comportement, nous profitons dans cette implantation, dans les solveurs de contraintes, des aléas provoqués par l'heuristique de choix aléatoire pour venir perturber l'apprentissage de l'heuristique choisie par l'utilisateur. Le parallèle à l'analogie serait une abeille capable d'apprendre à partir du vol aléatoire de la mouche.

Dans cette idée de perturbation, nous avons proposé, dans le chapitre 6, deux manières de perturber une heuristique choisie par l'utilisateur [PAPARRIZOU & WATTEZ 2020]. La première (a) est celle que nous avons décrit plus haut, où l'heuristique de choix aléatoire vient perturber l'heuristique choisie par l'utilisateur. La seconde manière de perturber une heuristique donnée (b), est de l'instancier plusieurs fois en diversifiant son tie-breaker : ce tie-breaker possède un rôle primordial sur les premiers temps de résolution où l'heuristique apprenante ne possède, possiblement, aucune information sur le réseau de contraintes. Ce tie-breaker permet ainsi un démarrage des différentes instanciations de cette même heuristique à différents endroits de l'espace de recherche.

La première méthode (a) de perturbation a donné de bons résultats sur l'ensemble des heuristiques sauf $\texttt{wdeg}^{\texttt{ca}\cdot\texttt{cd}}$. De manière globale, ce sont les heuristiques les moins adaptatives qui ont eu le meilleur comportement en retour de cette perturbation. En contre-partie, la seconde méthode (b) a moins profité à ces précédentes heuristiques hormis $\texttt{wdeg}^{\texttt{ca}\cdot\texttt{cd}}$ qui a pu progresser grâce à la diversification. Concernant l'extension aux problèmes de contraintes sous optimisation (chapitre 7) et de façon plus marquée que pour l'apprentissage de la meilleure heuristique, la perturbation d'heuristique dans ce cadre offre constamment de meilleurs résultats que dans le cas des heuristiques d'origine.

Les perspectives à ce sujet s'orientent vers la seconde méthode de perturbation (b). Actuellement, nous choisissons arbitrairement les heuristiques servant de tie-breakers en instanciant des heuristiques de choix aléatoire avec des racines différentes pour leur fonction de randomisation. Or, rien ne nous certifie que deux tie-breakers aléatoires n'ont pas mis, mutuellement et indirectement, les variables d'une même partie compliquée du problème en priorité. Une manière simple de possiblement éviter cela lorsque nous souhaitons $K = 2$ heuristiques est d'utiliser le tie-breaker `lexico` d'une part et `anti-lexico` d'autre part. Dans le cas plus complexe de l'instanciation de plus deux heuristiques, une solution pourrait être, par exemple, d'appliquer un algorithme de clusterisation (type K-*mean*) du graphe d'implication du réseau de contraintes afin d'extraire des variables à mettre en valeur pour le tie-breaker. Pour chacune des K instanciations, il suffit de mettre en priorité, dans l'ordonnancement, les variables d'un des K clusters, puis d'ajouter ensuite les autres par voisinage.

Perspectives au-delà de la problématique

Toujours au sujet de l'apprentissage et la perturbation d'heuristique, il serait intéressant de sortir du cadre de la programmation par contraintes afin d'exporter ces frameworks dans d'autres paradigmes de résolution. La chapitre 7 nous donne un premier aperçu de ce que cela pourrait donner dans le cadre de l'optimisation (tout en restant sous le paradigme des contraintes). Une étude récente montre que l'export

du principe du framework RES, apprenant la meilleure heuristique, vers les solveurs SAT semblent aussi porter ses fruits [CHERIF *et al.* 2020a, CHERIF *et al.* 2021].

Hors du domaine des bandits, il pourrait aussi être intéressant d'étendre la structure arborescente du modèle de tournoi vers un algorithme génétique capable, non pas de discriminer les heuristiques par le biais de duels, mais de récupérer les meilleures performances de chacune d'elles en les fusionnant. Ainsi, ce modèle alternerait entre la diversification/mutation d'une heuristique pendant l'étape actuelle d'échantillonnage et la fusion d'heuristiques dans leur meilleur état historique (obtenu lors des périodes de mutation).

Dans un autre niveau de gain d'autonomie, il serait intéressant de remplacer l'expert produisant les heuristiques par un mécanisme d'apprentissage autonome. Grâce à l'extraction de l'état du solveur et du réseau de contraintes à des moments clés de la résolution (à chaque nœud de recherche, à chaque conflit, à chaque solution obtenue, *etc.*) il serait intéressant qu'un mécanisme d'apprentissage déduise par lui-même une nouvelle heuristique d'ordonnancement des variables qui lui semble viable pour venir rapidement à bout d'une résolution. Des travaux s'orientant vers cette thématique apparaissent au sein des solveurs SAT [KURIN *et al.* 2020] et aussi dans la résolution de contraintes [SONG *et al.* 2020] par application de principe d'apprentissage profond par renforcement.

Enfin, d'un point de vue plus général, le principe heuristique étant extrait de la nature des êtres vivants et s'étant amélioré avec le temps par procédé évolutionniste et par apprentissage, il ne serait pas étonnant que l'avenir des heuristiques, dans sa contre-partie artificielle, trouve un nouvel élan d'efficacité à travers ces mêmes principes évolutionnistes et d'apprentissage proposés par les algorithmes génétiques et d'apprentissage profond sus-mentionnés.

Annexes

Annexe A

Implantations dans le Solveur ACE

Sommaire

A.1 Description de l'implantation des stratégies multi-heuristiques

La figure A.1 propose un diagramme de classes des différentes implantations proposées à travers ce manuscrit concernant l'apprentissage et la perturbation d'heuristique. Ci-après, nous énumérons les différentes classes et les décrivons :

- *Heuristique* est une classe abstraite que nous retrouvons par défaut au sein du solveur ACE proposant l'ensemble des méthodes décrites à travers les sections 3.2.4 et 4.2.2 permettant de décrire le comportement global d'une heuristique. La méthode *bestVar(float equiv)* permet de gérer le paramètre d'équivalence introduit dans la section 6.2.2.
- *ABS*, *IBS*, *DomDdeg*, *WdegCACD* et *DomWdegCHS* correspondent aux classes concrètes des heuristiques `abs`, `ibs`, `dom/ddeg`, $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca.cd}}$ héritant de la classe *Heuristique* et décrites dans les sections 1.4.1 et 3.3.3.
- *MultiHeuristiques* est une classe abstraite décrivant le comportement global d'une stratégie multi-heuristiques. Cette classe hérite d'*Heuristique* et est composée d'une liste d'instances d'*Heuristique*.
- *PolitiqueSélection* est une classe abstraite composant la classe *MultiHeuristiques* et permet de gérer le comportement générique d'une politique de sélection. Celle-ci prend en paramètre un simple entier K correspondant au nombre de choix disponibles.
- *PolitiqueUniforme* et *PolitiqueGrimes* sont deux concrétisations directes de la classe *PolitiqueSélection*. Ces politiques sont décrites dans les sections 2.2.4 et 6.2.3.
- *PolitiqueBandit* est une nouvelle classe abstraite héritant de la classe *PolitiqueSélection*. Celle-ci nécessite l'instanciation de l'objet *FonctionRécompense*.
- *EGreedy*, *UCB1*, *MOSS*, *TS*, *EXP3* et *ST* sont les concrétisations de la classe *PolitiqueBandit* et sont décrites dans les sections 2.2 et 5.2.
- *FonctionRécompense* est l'abstraction des classes concrètes *AUVR*, *ESB* et *NPTS* permettant de gérer les récompenses données aux des bandits.
- Enfin, le liant concrétisant l'implantation de l'ensemble de ces classes sont les concrétisations des frameworks *MultiHeuristiquesParNoeud*, *MultiHeuristiquesParRun* et *MultiHeuristiquesPerturbéesParRun* décrits dans les sections 4.2.2, 4.3 et 6.3.1.

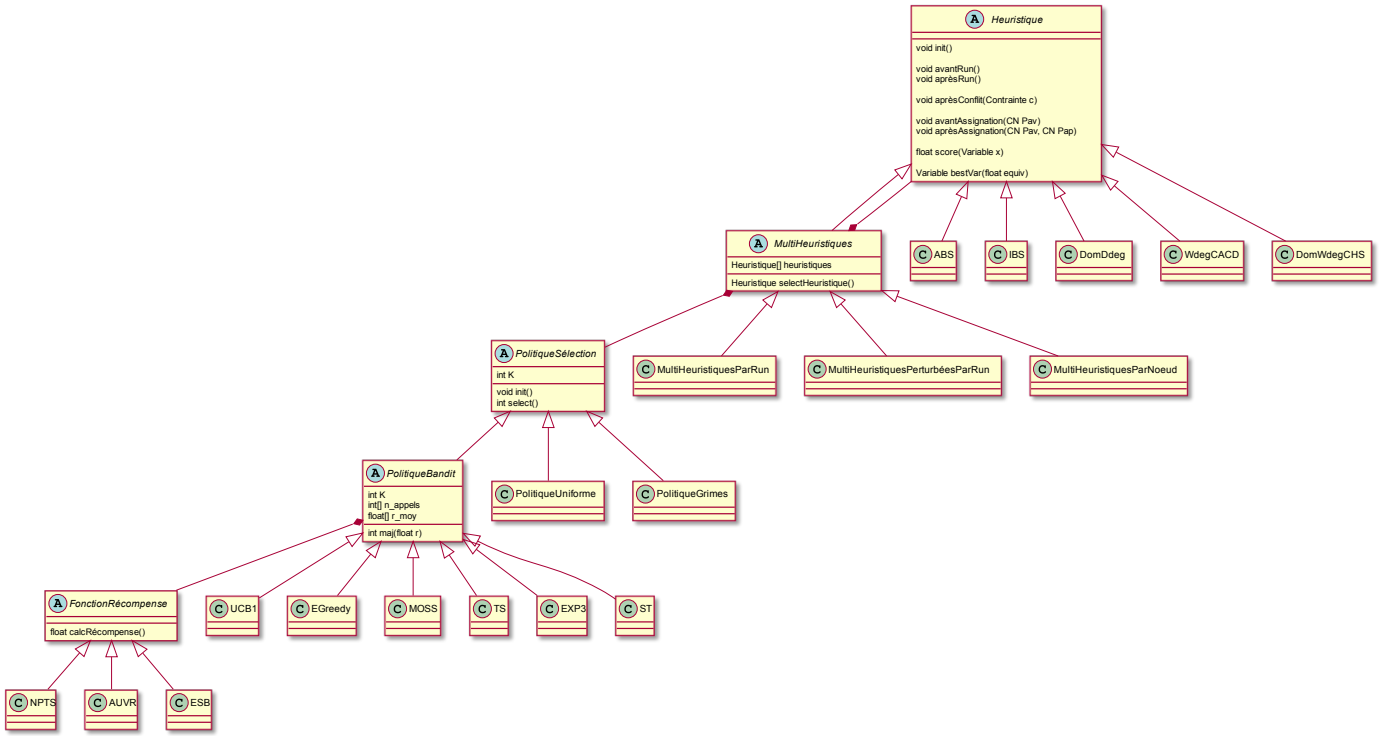


FIGURE A.1 – Diagramme de classes de l'implantation des stratégies multi-heuristiques dans ACE

Deux patrons de conception se dissimulent dans cette implantation orientée objet :

— *MultiHeuristiques* complète le comportement du patron de conception *Composite* : cette classe hérite de la même classe dont elle est, elle-même, composée.

— *PolitiqueSélection* et *FonctionRécompense* correspondent, tous deux, au patron de conception *Stratégie* donnant un comportement/stratégie particulier aux abstractions *MultiHeuristiques* et *PolitiqueBandit*.

Annexe B

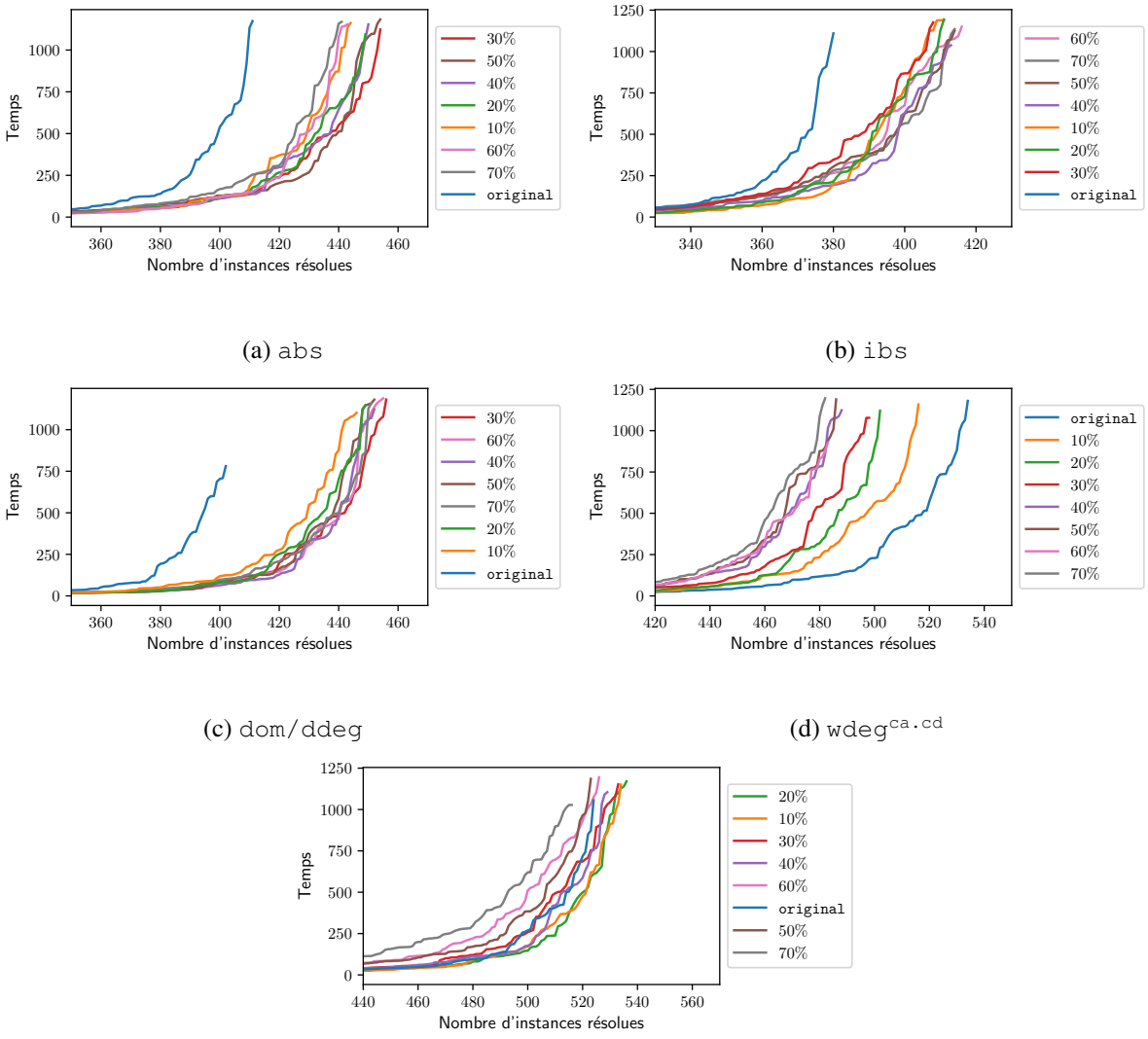
Extensions d'Analyse

Sommaire

Les analyses faisant parties de cette annexe correspondent à des analyses volumineuses pour lesquelles nous avons sélectionné les meilleurs résultats dans le corps du manuscrit.

B.1 Perturbation d'Heuristique (analyses CSP étendues)

B.1.1 Politiques de perturbation statiques



(a) `abs` (b) `ibs`

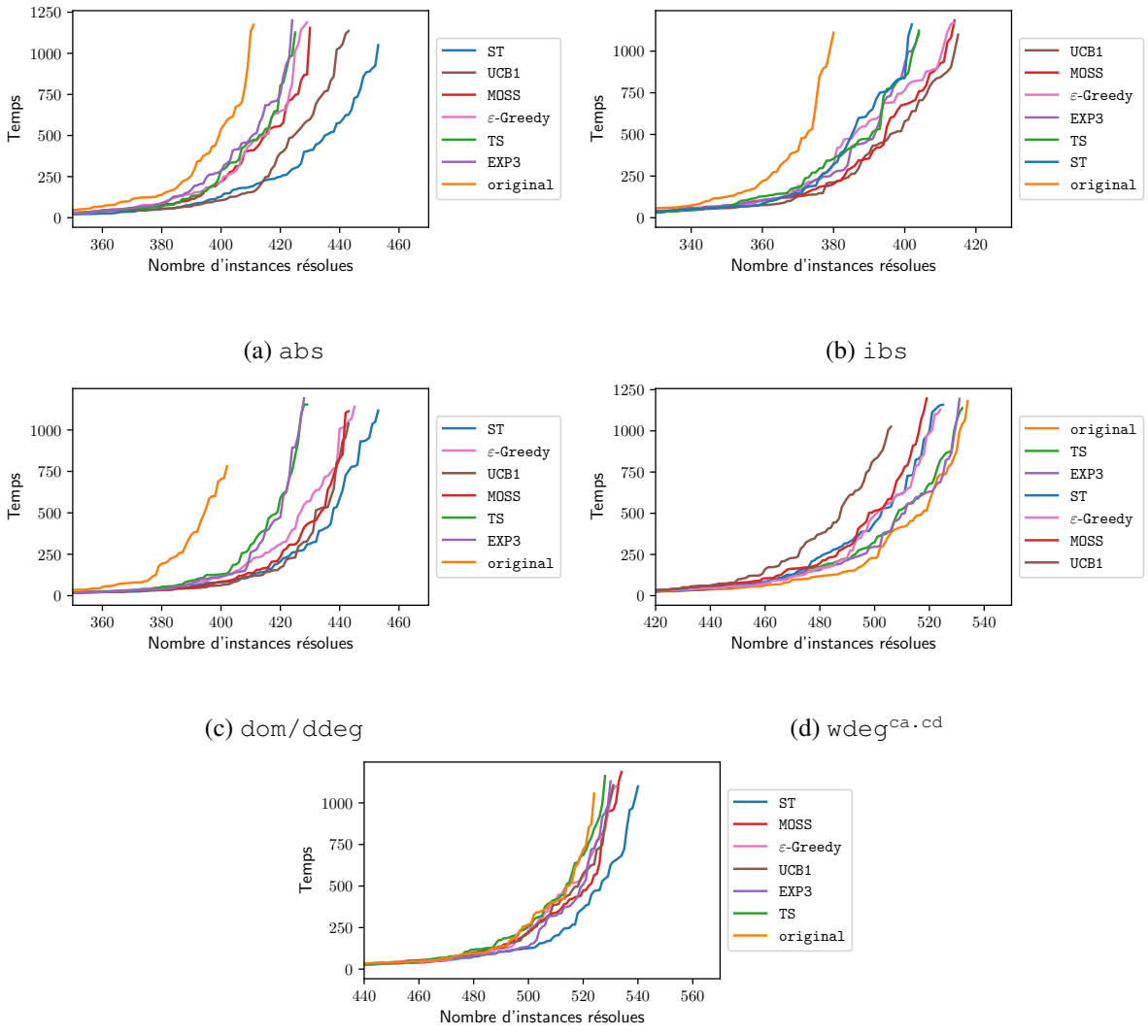
(c) `dom/ddeg` (d) $\texttt{wdeg}^{\texttt{ca.cd}}$

(e) $\texttt{dom/wdeg}^{\texttt{chs}}$

FIGURE B.1 – Politiques statiques pour chaque heuristique de $\mathcal{H}^{\texttt{base}}$ [Ψ_9, $\mathcal{I}_{\texttt{CSP}}$, ↻]

Dans la figure B.1, les différentes heuristiques $\mathcal{H}^{\texttt{base}}$ perturbées par la politique de perturbation statique sont expérimentées. Nous extrayons de cette figure le meilleur représentant de chaque heuristique : par exemple `abs` perturbée à 30% semble la politique la plus efficace à l'approche du timeout.

B.1.2 Politiques de perturbation dirigées par un bandit

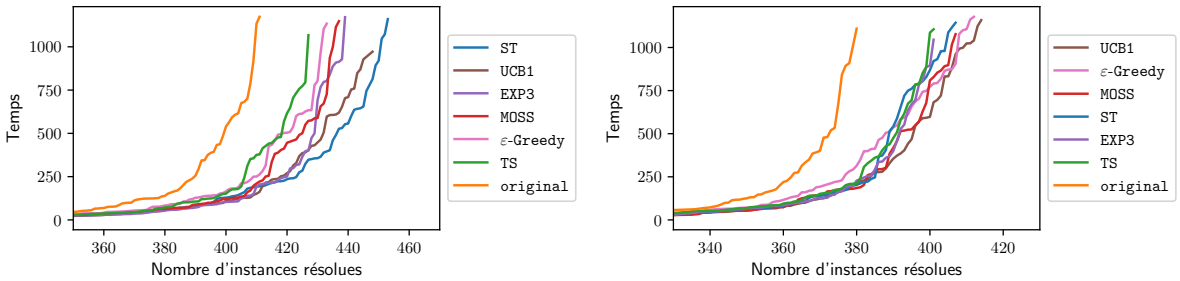


(a) `abs` (b) `ibs`

(c) `dom/ddeg` (d) `wdeg`$^{\texttt{ca.cd}}$

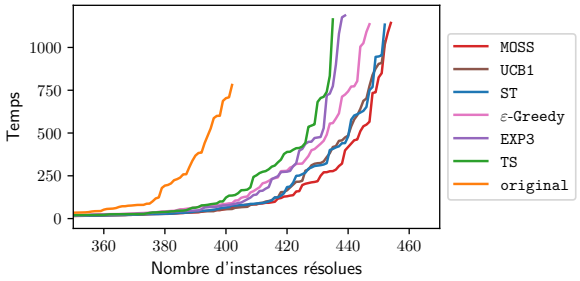
(e) `dom/wdeg`$^{\texttt{chs}}$

FIGURE B.2 – Aperçu de la perturbation des heuristiques par le framework `PER` et la fonction de récompense `auvr` sur 1 200 secondes [Ψ_{11}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

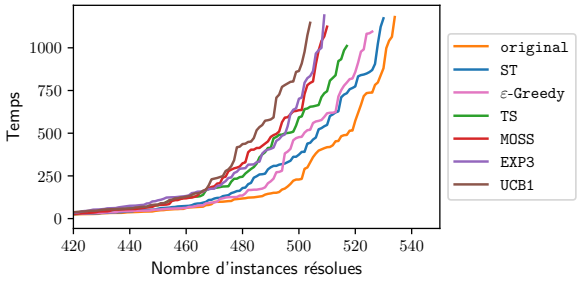


(a) `abs`

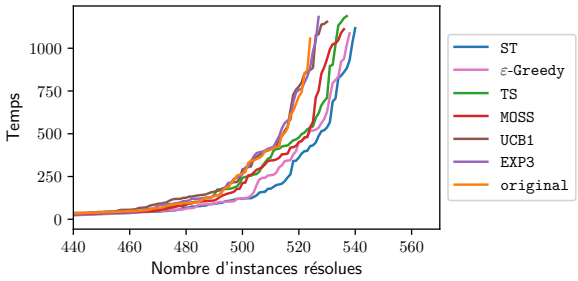
(b) `ibs`



(c) `dom/ddeg`

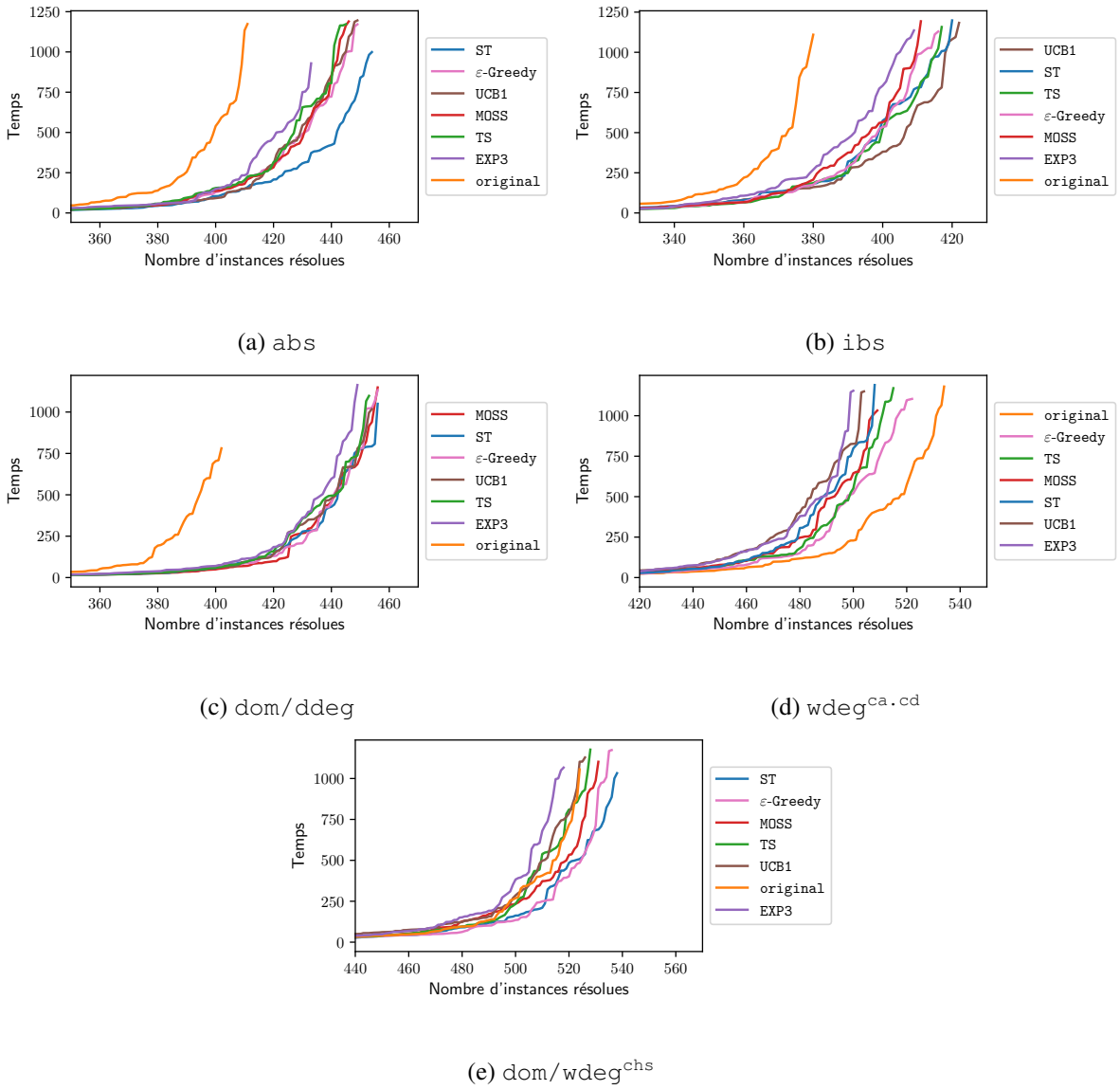


(d) $\texttt{wdeg}^{\texttt{ca.cd}}$



(e) $\texttt{dom/wdeg}^{\texttt{chs}}$

FIGURE B.3 – Aperçu de la perturbation des heuristiques par le framework `PER` et la fonction de récompense `esb` sur 1 200 secondes [Ψ_{11}, $\mathcal{I}_{\texttt{CSP}}$, $\circlearrowleft$]



(a) abs

(b) ibs

(c) dom/ddeg

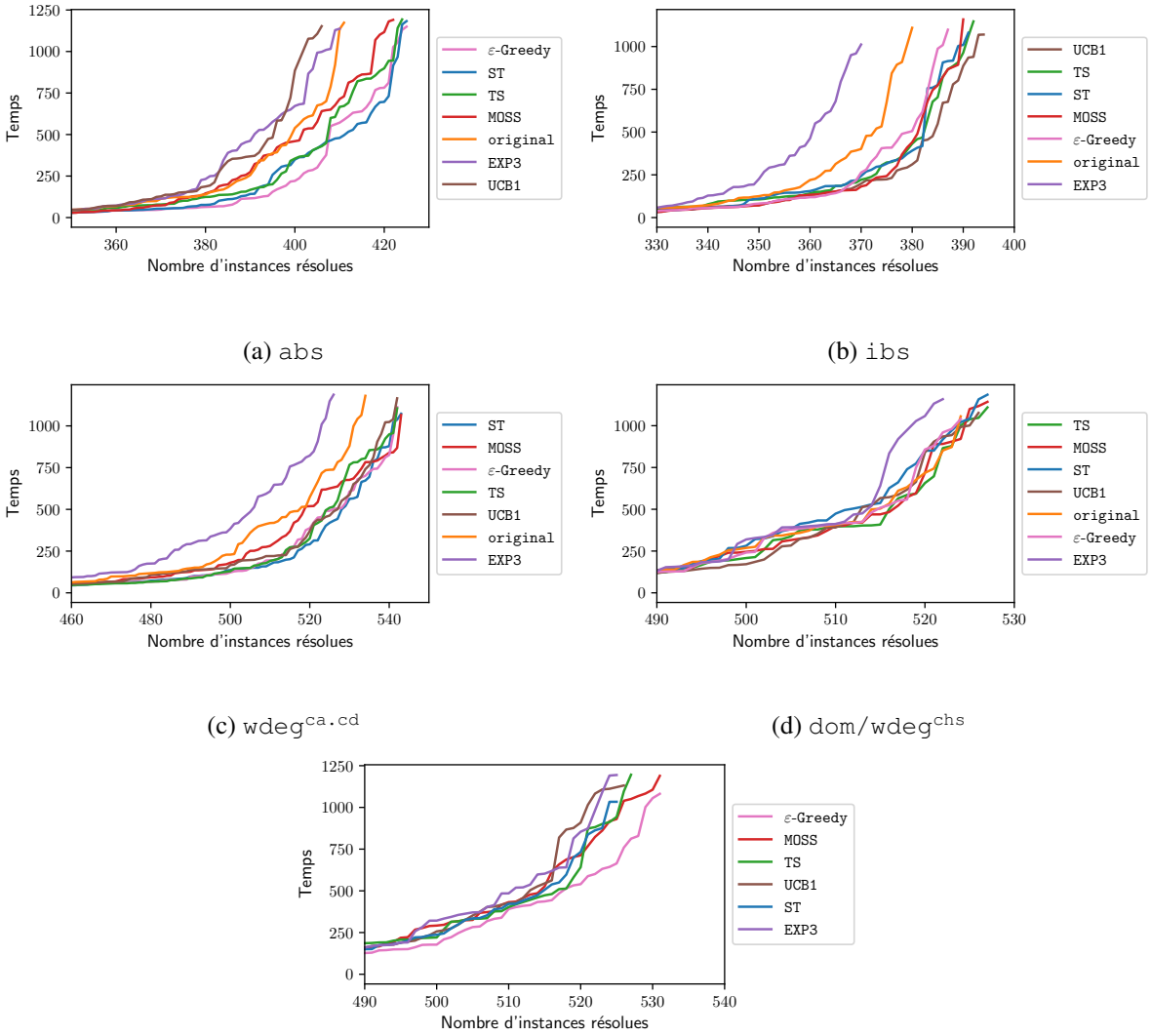
(d) wdeg$^{ca.cd}$

(e) dom/wdeg^{chs}

FIGURE B.4 – Aperçu de la perturbation des heuristiques par le framework PER et la fonction de récompense npts sur 1 200 secondes [Ψ_{11}, $\mathcal{I}_{CSP}$, ♉]

Pour les figures B.2, B.3 et B.4, représentants les performances des trois fonctions de récompense auvr, esb et npts appliquées sur l'ensemble des bandits de l'état de l'art au sein du framework PER, nous extrayons de la même manière les meilleurs représentant de chacun des cactus plots. Cela représente trois configurations pour chaque heuristique correspondant aux trois fonctions de récompenses (en cas de proximité dans les performances, il se peut que plus de trois configurations soient extraites).

B.1.3 Perturbation d'heuristique par le tie-breaker



(a) `abs` (b) `ibs`

(c) $\texttt{wdeg}^{\texttt{ca.cd}}$ (d) $\texttt{dom/wdeg}^{\texttt{chs}}$

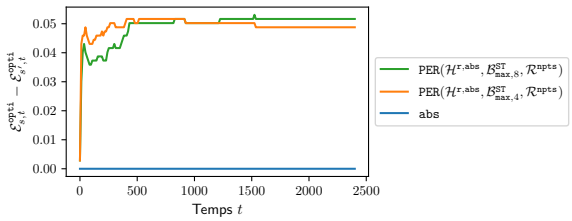
(e) $\texttt{dom/wdeg}^{\texttt{chs}}$

FIGURE B.5 – Aperçu de la perturbation de la diversification des heuristiques avec le framework `RES` et la fonction de récompense `npts` sur 1 200 secondes [Ψ_{12}, $\mathcal{I}_{\texttt{CSP}}$, ↻]

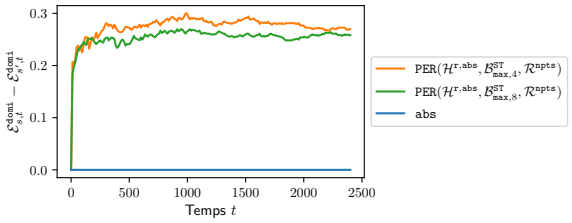
Dans la même optique, il est extrait depuis les cactus plots de la figure B.5 le ou les deux meilleur(s) représentant(s) pour chaque heuristique. Ici, les heuristiques sont perturbés par diversification de leur tie-breaker.

B.2 Perturbation d'Heuristique (analyses COP étendues)

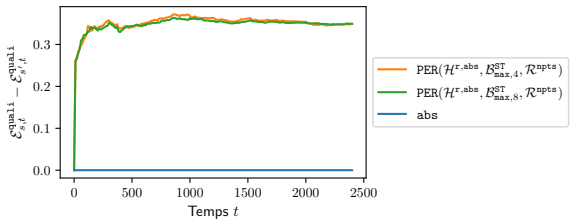
Les cinq figures qui suivent représentent les performances des cinq heuristiques de $\mathcal{H}^{\texttt{base}}$ perturbées de différentes manières. Chaque figure est divisée en quatre sous-figures représentant quatre critères de comparaison.



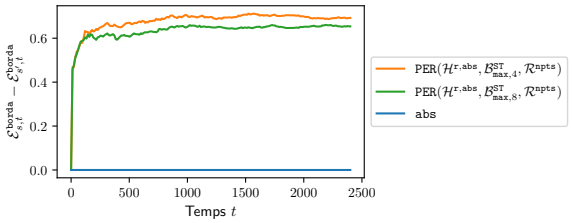
(a) Différence de proportion d'*optimalité* avec le solveur $s' = \texttt{abs}$



(b) Différence de proportion de *dominance* avec le solveur $s' = \texttt{abs}$

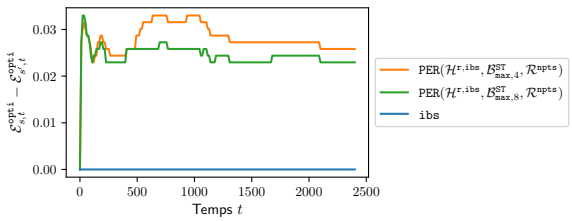


(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \texttt{abs}$

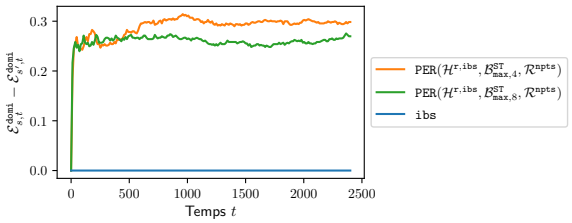


(d) Différence du classement moyen de *Borda* avec le solveur $s' = \texttt{abs}$

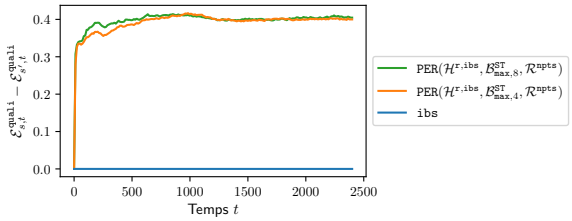
FIGURE B.6 – Perturbation de l'heuristique `abs` [Ψ_{19}, $\mathcal{I}_{\texttt{COP}}$, ♁]



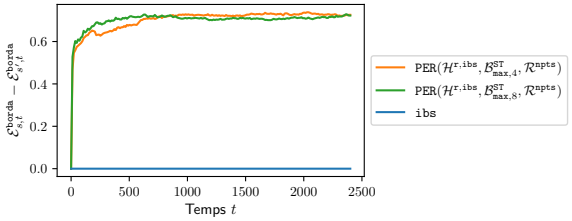
(a) Différence de proportion d'*optimalité* avec le solveur $s' = \texttt{ibs}$



(b) Différence de proportion de *dominance* avec le solveur $s' = \texttt{ibs}$

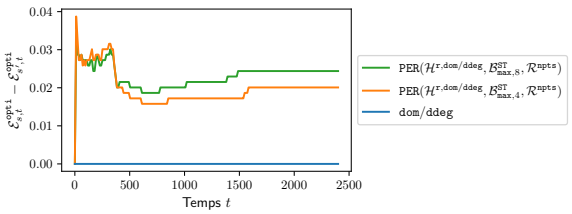


(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \texttt{ibs}$

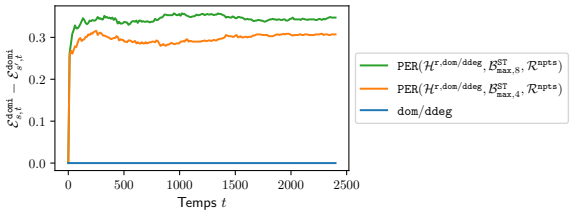


(d) Différence du classement moyen de *Borda* avec le solveur $s' = \texttt{ibs}$

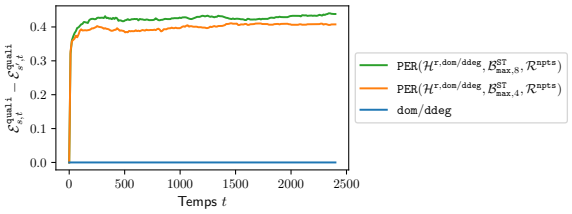
FIGURE B.7 – Perturbation de l'heuristique `ibs` [Ψ_{19}, $\mathcal{I}_{\texttt{COP}}$, ♁]



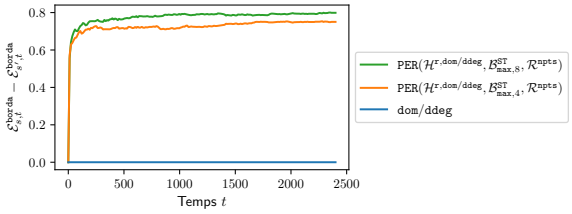
(a) Différence de proportion d'*optimalité* avec le solveur $s' = \texttt{dom/ddeg}$



(b) Différence de proportion de *dominance* avec le solveur $s' = \texttt{dom/ddeg}$

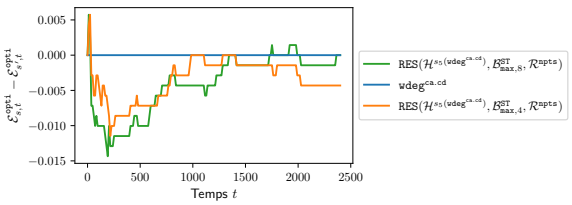


(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \texttt{dom/ddeg}$

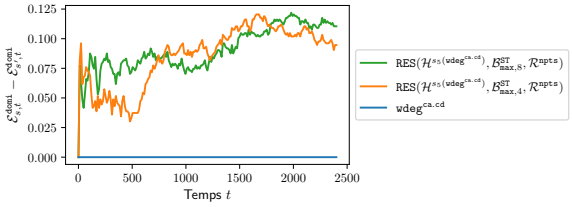


(d) Différence du classement moyen de *Borda* avec le solveur $s' = \texttt{dom/ddeg}$

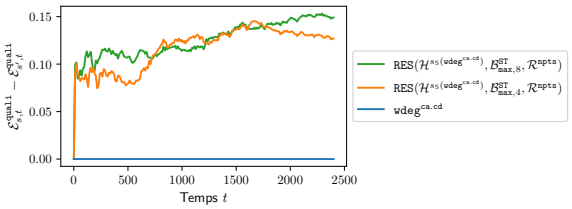
FIGURE B.8 – Perturbation de l'heuristique $\texttt{dom/ddeg}$ [Ψ_{19}, $\mathcal{I}_{\text{COP}}$, ⟳]



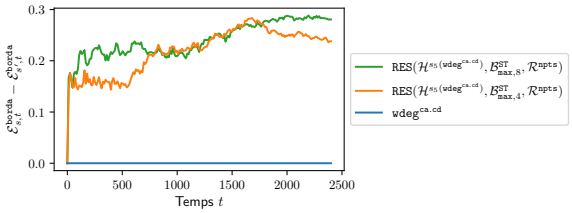
(a) Différence de proportion d'*optimalité* avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$



(b) Différence de proportion de *dominance* avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$

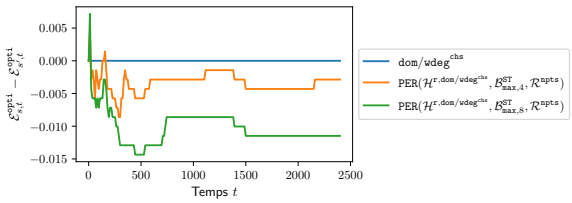


(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$

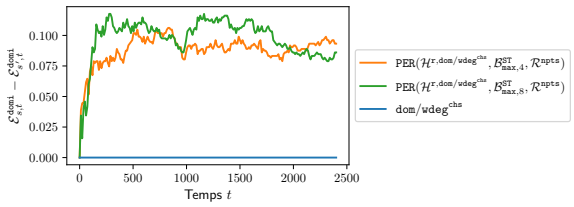


(d) Différence du classement moyen de *Borda* avec le solveur $s' = \texttt{wdeg}^{\texttt{ca.cd}}$

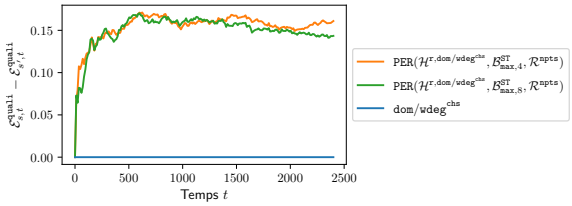
FIGURE B.9 – Perturbation de l'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$ [Ψ_{19}, $\mathcal{I}_{\text{COP}}$, ⟳]



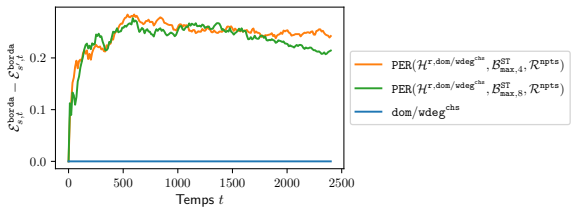
(a) Différence de proportion d'*optimalité* avec le solveur $s' = \texttt{dom/wdeg}^{\texttt{chs}}$



(b) Différence de proportion de *dominance* avec le solveur $s' = \texttt{dom/wdeg}^{\texttt{chs}}$



(c) Différence de la *qualité* moyenne des solutions avec le solveur $s' = \texttt{dom/wdeg}^{\texttt{chs}}$



(d) Différence du classement moyen de *Borda* avec le solveur $s' = \texttt{dom/wdeg}^{\texttt{chs}}$

FIGURE B.10 – Perturbation de l'heuristique $\texttt{dom/wdeg}^{\texttt{chs}}$ [Ψ_{19}, $\mathcal{I}_{\texttt{COP}}$, ↻]

Index

Bibliographie

[AMADINI *et al.* 2016] R. AMADINI, M. GABBRIELLI, et J. MAURO. « Portfolio Approaches for Constraint Optimization Problems ». *Annals of Mathematics and Artificial Intelligence*, 76(1) :229–246, février 2016. 4.2.1

[APT 2003] K. R. APT. *Principles of Constraint Programming*. Cambridge University Press, 2003. (document), 1.1, 7.5

[AUDIBERT & BUBECK 2009] J-Y. AUDIBERT et S. BUBECK. « Minimax Policies for Adversarial and Stochastic Bandits ». Dans *COLT*, pages 217–226, Montreal, Canada, 2009. 12

[AUER *et al.* 2002a] P. AUER, N. CESA-BIANCHI, et P. FISCHER. « Finite-Time Analysis of the Multiarmed Bandit Problem ». *Machine Learning*, 47(2) :235–256, mai 2002. 13

[AUER *et al.* 2002b] P. AUER, N. CESA-BIANCHI, Y. FREUND, et R. SCHAPIRE. « The Nonstochastic Multiarmed Bandit Problem ». *SIAM Journal on Computing*, 32(1) :48–77, 2002. 2.2.3

[BALAFREJ *et al.* 2015] A. BALAFREJ, C. BESSIERE, et A. PAPARRIZOU. « Multi-Armed Bandits for Adaptive Constraint Propagation ». Dans *Proceedings of IJCAI'15*, pages 290–296, 2015. 4.2.2, 4.5.2

[BALCAN *et al.* 2018] M.-F. BALCAN, T. DICK, T. SANDHOLM, et E. VITERCIK. « Learning to Branch ». Dans *Proceedings of ICML'18*, pages 353–362, 2018. 4.2.1

[BELDICEANU *et al.* 2007] N. BELDICEANU, M. CARLSSON, S. DEMASSEY, et T. PETIT. « Global Constraint Catalogue : Past, Present and Future ». *Constraints*, 12(1) :21–62, mars 2007. 1.2.1

[BESSIÈRE & RÉGIN 1996] C BESSIÈRE et J.C. RÉGIN. « MAC and Combined Heuristics : Two Reasons to Forsake FC (and CBJ ?) On Hard Problems ». Dans *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming*, CP'96, pages 61–75, Berlin, Heidelberg, août 1996. Springer-Verlag. 20

[BESSIERE *et al.* 2004] C. BESSIERE, B. ZANUTTINI, et C. FERNANDEZ. « Measuring Search Trees ». Dans *Proceedings of ECAI'04 Workshop on Modelling and Solving Problems with Constraints*, pages 31–40, 2004. 9

[BESSIÈRE *et al.* 2005] C. BESSIÈRE, J.C. RÉGIN, R. H. C. YAP, et Y. ZHANG. « An Optimal Coarse-Grained Arc Consistency Algorithm ». *Artificial Intelligence*, 165(2) :165–185, 2005. 1.3.1

[BOUSSEMART *et al.* 2004] F. BOUSSEMART, F. HEMERY, C. LECOUTRE, et L. SAIS. « Boosting Systematic Search by Weighting Constraints ». Dans *Proceedings of ECAI'04*, pages 146–150, 2004. 12, 20

[BOUSSEMART *et al.* 2016] F. BOUSSEMART, C. LECOUTRE, et G. Audemard C. PIETTE. « XCSP3 : An Integrated Format for Benchmarking Combinatorial Constrained Problems ». *CoRR*, abs/1611.03398, 2016. 3.2.2, 3.2.4

[BOUSSEMART *et al.* 2020] F. BOUSSEMART, C. LECOUTRE, G. AUDEMARD, et C. PIETTE. « XCSP3-core : A Format for Representing Constraint Satisfaction/Optimization Problems ». *CoRR*, abs/2009.00514, 2020. 3.2.2, 3.2.4

[BUBECK & CESA-BIANCHI 2012] S. BUBECK et N. CESA-BIANCHI. *Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems*. Foundations and Trends in Machine Learning. Now Publishers, 2012. 35, 36, 2.2.3

[BUBECK *et al.* 2009] S. BUBECK, R. MUNOS, et G. STOLTZ. « Pure Exploration in Multi-Armed Bandits Problems ». Dans R. GAVALDÀ, G. LUGOSI, T. ZEUGMANN, et S. ZILLES, éditeurs, *Algorithmic Learning Theory*, pages 23–37, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 2.4.3

[CHEN & FRAZIER 2017] B. CHEN et P.I. FRAZIER. « Dueling Bandits with Weak Regret ». *CoRR*, abs/1706.04304, 2017. 5.2.1

[CHERIF *et al.* 2020a] M. S. CHERIF, D. HABET, et C. TERRIOUX. « Combining VSIDS and CHB Using Restarts in SAT ». Dans *Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP'21)*, octobre 2020. 7.5

[CHERIF *et al.* 2020b] M. S. CHERIF, D. HABET, et C. TERRIOUX. « On the Refinement of Conflict History Search Through Multi-Armed Bandit ». Dans *IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 264–271, Baltimore, United States, novembre 2020. IEEE. 11, 3.4.1, 3.5.1, 6.3.3, 7.5

[CHERIF *et al.* 2021] M. S. CHERIF, D. HABET, et C. TERRIOUX. « Un Bandit Manchot Pour Combiner CHB et VSIDS ». Dans *Actes Des 16èmes Journées Francophones de Programmation Par Contraintes (JFPC)*, Nice, France, juin 2021. 7.5

[DECHTER 1990] R. DECHTER. « Enhancement Schemes for Constraint Processing : Backjumping, Learning, and Cutset Decomposition ». *Artificial Intelligence*, 41(3) :273–312, 1990. 20

[DEMIROVIC *et al.* 2018] E. DEMIROVIC, G. CHU, et P. STUCKEY. « Solution-Based Phase Saving for CP : A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers ». Dans *Proceedings of CP'18*, pages 99–108, 2018. 7.2.1, 7.3

[EPSTEIN & PETROVIC 2007] S. EPSTEIN et S. PETROVIC. « Learning to Solve Constraint Problems ». Dans *ICAPS-07 Workshop on Planning and Learning*, 2007. 4.2.1

[FALQUE *et al.* 2020] T. FALQUE, R. WALLON, et H. WATTEZ. « Metrics : Towards a Unified Library for Experimenting Solvers ». Dans *11th International Workshop on Pragmatics of SAT (POS'20)*, juillet 2020. (document), 7.5

[FALQUE *et al.* 2021a] T. FALQUE, C. LECOUTRE, B. MAZURE, et H. WATTEZ. « Descente Agressive de la Borne en Optimisation sous Contraintes ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021. (document), 7.3.4, 7.5

[FALQUE *et al.* 2021b] T. FALQUE, R. WALLON, et H. WATTEZ. « Metrics : Mission Expérimentations ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021. (document), 3.2.3, 7.5

[FREIRE *et al.* 2016] J. FREIRE, N. FUHR, et A. RAUBER. « Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041) ». *Dagstuhl Reports*, 6(1) :108–159, 2016. 3.2.3

[GAGLIOLO & SCHMIDHUBER 2007] M. GAGLIOLO et J. SCHMIDHUBER. « Learning Restart Strategies ». Dans *Proceedings of IJCAI'07*, pages 792–797, 2007. 4.2.2

[GITTINS 1989] J.C. GITTINS. *Multi-Armed Bandit Allocation Indices*. Wiley-Interscience Series in Systems and Optimization. Wiley, 1989. (document), 2.1, 7.5

[GOMES *et al.* 1998] C. P. GOMES, B. SELMAN, et H. KAUTZ. « Boosting Combinatorial Search Through Randomization ». Dans *Proceedings of AAAI '98*, pages 431–437, 1998. 1.3.3, 6.2.2, 6.4.1

[GOMES *et al.* 2000] C. GOMES, B. SELMAN, N. CRATO, et H. KAUTZ. « Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems ». *Journal of Automated Reasoning*, 24(1) :67–100, 2000. 1.3.3, 6.1, 6.2.2

[GRIMES & WALLACE 2007] D. GRIMES et R. WALLACE. « Sampling Strategies and Variable Selection in Weighted Degree Heuristics ». Dans *Proceedings of CP'07*, pages 831–838, 2007. 6.2.3

[HABET & TERRIOUX 2021] D. HABET et C. TERRIOUX. « Conflict History Based Heuristic for Constraint Satisfaction Problem Solving ». *Journal of Heuristics*, 27(6) :951–990, juin 2021. 11

[HARALICK & ELLIOTT 1980] R. HARALICK et G. ELLIOTT. « Increasing Tree Search Efficiency for Constraint Satisfaction Problems ». *Artificial Intelligence*, 14 :263–313, 1980. 9, 1.4

[HARVEY & GINSBERG 1995] W. D. HARVEY et M. L. GINSBERG. « Limited Discrepancy Search ». Dans *Proceedings of IJCAI'95*, pages 607–615, 1995. 6.2.2

[HOGG *et al.* 1996] T. HOGG, B. A. HUBERMAN, et C. P. WILLIAMS. « Phase Transitions and the Search Problem ». *Artificial Intelligence*, 81(1) :1–15, 1996. 6.1

[HURLEY *et al.* 2014] H. HURLEY, L. KOTTHOFF, Y. MALITSKY, et B. O'SULLIVAN. « Proteus : A Hierarchical Portfolio of Solvers and Transformations ». Dans *Proceedings of CPAIOR'2014*, pages 301–317, 2014. 4.2.1

[KIM *et al.* 2018] Y.M. KIM, J.B. POLINE, et G. DUMAS. « Experimenting with Reproducibility : A Case Study of Robustness in Bioinformatics ». *GigaScience*, 7(7) :giy077, juillet 2018. 3.2.3

[KURIN *et al.* 2020] V. KURIN, S. GODIL, S. WHITESON, et B. CATANZARO. « Improving {SAT} Solver Heuristics with Graph Networks and Reinforcement Learning », 2020. 7.5

[LATTIMORE & SZEPESVÁRI 2020] T. LATTIMORE et C. SZEPESVÁRI. *Bandit Algorithms*. Cambridge University Press, 2020. 2.1

[LE BERRE & PARRAIN 2010] D. LE BERRE et A. PARRAIN. « The Sat4j Library, Release 2.2 ». *JSAT*, 7 :59–6, janvier 2010. 7.3.4

[LECOUTRE & SZCZEPANSKI 2020] C. LECOUTRE et N. SZCZEPANSKI. « PYCSP3 : Modeling Combinatorial Constrained Problems in Python ». *CoRR*, abs/2009.00326, 2020. 3.2.2, 3.2.4

[LECOUTRE *et al.* 2006] C. LECOUTRE, L. SAIS, S. TABARY, et V. VIDAL. « Last Conflict-Based Reasonning ». Dans *Proceedings of ECAI'06*, pages 133–137, 2006. 20, 23

[LECOUTRE *et al.* 2007] C. LECOUTRE, L. SAIS, S. TABARY, et V. VIDAL. « Recording and Minimizing Nogoods from Restarts ». *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 1 :147–167, 2007. 34

[LECOUTRE 2009] C. LECOUTRE. *Constraint Networks : Techniques and Algorithms*. ISTE/Wiley, 2009. (document), 1.1, 7.5

[LIANG *et al.* 2016] J. LIANG, V. GANESH, P. POUPART, et K. CZARNECKI. « Exponential Recency Weighted Average Branching Heuristic for SAT Solvers ». Dans *AAAI*, 2016. 11

[LIBERATORE 2000] P. LIBERATORE. « On the Complexity of Choosing the Branching Literal in DPLL ». *Artificial Intelligence*, 116(1) :315–326, 2000. 1.4

[LOTH *et al.* 2013] M. LOTH, M. SEBAG, Y. HAMADI, et M. SCHOENAUER. « Bandit-Based Search for Constraint Programming ». Dans *Proceedings of CP'13*, pages 464–480, 2013. 4.2.2

[LUBY *et al.* 1993] M. LUBY, A. SINCLAIR, et D. ZUCKERMAN. « Optimal Speedup of Las Vegas Algorithms ». *Information Processing Letters*, 47(4) :173–180, 1993. 1.3.3, 4.3.2

[MACKWORTH 1977] A. K. MACKWORTH. « Consistency in Networks of Relations ». *Artificial Intelligence*, 8(1) :99–118, 1977. 1.2.1, 1.3.1, 1.3.1

[MERCHEZ *et al.* 2001] S. MERCHEZ, C. LECOUTRE, et F. BOUSSEMART. « AbsCon : A Prototype to Solve CSPs with Abstraction ». Dans *Proceedings of CP'01*, pages 730–744, 2001. 3.2.4

[MICHEL & HENTENRYCK 2012] L. MICHEL et P. V. HENTENRYCK. « Activity-Based Search for Black-Box Constraint Programming Solvers ». Dans *Proceedings of CPAIOR'12*, pages 228–243, 2012. 14, 12, 20, 3.4.1

[MONTANARI 1974] U. MONTANARI. « Network of Constraints : Fundamental Properties and Applications to Picture Processing ». *Information Science*, 7 :95–132, 1974. (document), 1.2.1, 7.5

[NADEL 1988] Bernard A. NADEL. Tree Search and ARC Consistency in Constraint Satisfaction Algorithms. Dans Laveen KANAL et Vipin KUMAR, éditeurs, *Search in Artificial Intelligence*, pages 287–342. Springer New York, New York, NY, 1988. 9

[O'MAHONY *et al.* 2008] E. O'MAHONY, E. HEBRARD, A. HOLLAND, C. NUGENT, et B. O'SULLIVAN. « Using Case-Based Reasoning in an Algorithm Portfolio for Constraint Solving ». Dans *Proceedings of AICS'10*, 2008. 4.2.1

[PAPARRIZOU & WATTEZ 2020] A. PAPARRIZOU et H. WATTEZ. « Perturbing Branching Heuristics in Constraint Solving ». Dans *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP'20)*, pages 496–513, septembre 2020. (document), 3.5.1, 7.5, 7.5

[PERRON 2010] L. PERRON. « http ://code.google.com/p/or-tools/ », 2010. 12

[PILAT & FUKASAKU 2007] D. PILAT et Y. FUKASAKU. « OECD Principles and Guidelines for Access to Research Data from Public Funding ». *Data Science Journal*, 6 :4–11, juin 2007. 3.2.3

[PISINGER & ROPKE 2010] D. PISINGER et S. ROPKE. « Large Neighborhood Search ». Dans *Handbook of Metaheuristics*, pages 399–419. Springer, 2010. 6.2.1

[RADLINSKI & JOACHIMS 2007] F. RADLINSKI et T. JOACHIMS. « Active Exploration for Learning Rankings from Clickthrough Data ». Dans *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 570–579, New York, NY, USA, 2007. Association for Computing Machinery. 2.4.1

[RADLINSKI *et al.* 2008] F. RADLINSKI, M. KURUP, et T. JOACHIMS. « How Does Clickthrough Data Reflect Retrieval Quality ? ». Dans *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, CIKM '08, pages 43–52, New York, NY, USA, 2008. Association for Computing Machinery. 2.4

[REFALO 2004] P. REFALO. « Impact-Based Search Strategies for Constraint Programming ». Dans *Proceedings of CP'04*, pages 557–571, 2004. 14, 12, 20, 3.4.1

[ROSSI *et al.* 2006] F. ROSSI, P. V. BEEK, et T. WALSH. *Handbook of Constraint Programming*. Elsevier, 2006. (document), 1.1, 7.5

[ROUSSEL 2011] O. ROUSSEL. « Controlling a Solver Execution : The Runsolver Tool ». *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :139–144, 2011. 3.2.2

[RUSSO & ROY 2016] D. RUSSO et B. V. ROY. « An Information-Theoretic Analysis of Thompson Sampling ». *J. Mach. Learn. Res.*, 17 :68 :1–68 :30, 2016. 12

[RUSSO *et al.* 2018] D. J. RUSSO, B. VAN ROY, A. KAZEROUNI, I. OSBAND, et Z. WEN. « A Tutorial on Thompson Sampling ». *Found. Trends Mach. Learn.*, 11(1) :1–96, juillet 2018. 12

[SABIN & FREUDER 1994] D. SABIN et E. C. FREUDER. « Contradicting Conventional Wisdom in Constraint Satisfaction ». Dans *Proceedings of CP'94*, pages 10–20, 1994. 9

[SELMAN *et al.* 1992] B. SELMAN, H. LEVESQUE, et D. MITCHELL. « A New Method for Solving Hard Satisfiability Problems ». Dans *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 440–446. AAAI Press, 1992. 6.2.1

[SELMAN *et al.* 1994] B. SELMAN, H. A. KAUTZ, et B. COHEN. « Noise Strategies for Improving Local Search ». Dans *Proceedings of AAAI '94*, pages 337–343, 1994. 6.2.1

[SHAW 1998] P. SHAW. « Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems ». Dans *Proceedings of CP'98*, pages 417–431, 1998. 7.3

[SMITH & GRANT 1998] B. SMITH et S. GRANT. « Trying Harder to Fail First ». Dans *Proceedings of ECAI'98*, pages 249–253, Brighton, UK, 1998. 1.4

[SONG *et al.* 2020] W. SONG, Z. CAO, J. ZHANG, et A. LIM. « Learning Variable Ordering Heuristics for Solving Constraint Satisfaction Problems », 2020. 7.5

[SÖRENSSON & EÉN 2005] N. SÖRENSSON et N ; EÉN. « MiniSat v1.13 - a SAT Solver with Conflict-Clause Minimization ». 2005. 1.3.3

[STREETER & SMITH 2007] M.J. STREETER et S.F. SMITH. « Using Decision Procedures Efficiently for Optimization ». Dans M.S. BODDY, M. FOX, et S. THIÉBAUX, éditeurs, *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, pages 312–319. AAAI, 2007. 7.3.2, 7.3.4

[SUI *et al.* 2018] Y. SUI, M. ZOGHI, K. HOFMANN, et Y. YUE. « Advancements in Dueling Bandits ». Dans *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 5502–5510, juillet 2018. 2.4.1

[SUTTON *et al.* 2018] R.S. SUTTON, A.G. BARTO, et F. BACH. *Reinforcement Learning : An Introduction*. MIT Press, 2018. 13

[THOMPSON 1933] W. R. THOMPSON. « On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples ». *Biometrika*, 25(3-4) :285–294, décembre 1933. 12

[VAN BEEK 2006] P. VAN BEEK. Backtracking Search Algorithms. Dans *Handbook of Constraint Programming*, Chapitre 4, pages 85–134. Elsevier, 2006. (document), 6.1

[VAN HOEVE & MILANO 2004] W.J. VAN HOEVE et M. MILANO. « Postponing Branching Decisions ». Dans *Proceedings of ECAI'04*, pages 1105–1106, 2004. 7.3.2

[VION & PIECHOWIAK 2017] J. VION et S. PIECHOWIAK. « Une Simple Heuristique Pour Rapprocher DFS et LNS Pour Les COP ». Dans *Proceedings of JFPC'17*, pages 39–45, 2017. 7.2.1, 7.3

[WATTEZ *et al.* 2019a] H. WATTEZ, F. KORICHE, C. LECOUTRE, A. PAPARRIZOU, et S. TABARY. « Heuristiques de recherche : un bandit pour les gouverner toutes ». Dans *Actes des 15es Journées Francophones de Programmation par Contraintes (JFPC'19)*, Albi, France, juin 2019. (document)

[WATTEZ *et al.* 2019b] H. WATTEZ, C. LECOUTRE, A. PAPARRIZOU, et S. TABARY. « Refining Constraint Weighting ». Dans *Proceedings of the 31st International Conference on Tools with Artificial Intelligence (ICTAI'19)*, pages 71–77, novembre 2019. (document), 3.3.3, 3.4.1, 7.5

[WATTEZ *et al.* 2020] H. WATTEZ, F. KORICHE, C. LECOUTRE, A. PAPARRIZOU, et S. TABARY. « Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts ». Dans *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI'20)*, Frontiers in Artificial Intelligence and Applications 325, IOS Press 2020, ISBN 978-1-64368-100-9, Santiago de Compostela (virtual), Spain, août 2020. (document), 3.5.1, 4.4.1, 4.5.2, 7.5

[WATTEZ *et al.* 2021a] H. WATTEZ, F. KORICHE, C. LECOUTRE, A. PAPARRIZOU, et S. TABARY. « Focus sur les heuristiques basées sur la pondération de contraintes ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021. (document), 7.5

[WATTEZ *et al.* 2021b] H. WATTEZ, F. KORICHE, et A. PAPARRIZOU. « Perturbation des heuristiques de branchement dans la résolution de contraintes ». Dans *Actes des 16es Journées Francophones de Programmation par Contraintes (JFPC'21)*, juin 2021. (document)

[WILSON & NASH 2003] R. WILSON et C. NASH. « Four Colours Suffice : How the Map Problem Was Solved ». *The Mathematical Intelligencer*, 25 :80–83, décembre 2003. 1.2.2

[XIA & YAP 2018] W. XIA et R. H. C. YAP. « Learning Robust Search Strategies Using a Bandit-Based Approach ». Dans *Proceedings of AAAI'18*, pages 6657–6665, 2018. 4.2.2, 4.4.1, 4.4.1, 4.5, 4.5.2

[XU *et al.* 2008] L. XU, F. HUTTER, H. H. HOOS, et K. LEYTON-BROWN. « SATzilla : Portfolio-based Algorithm Selection for SAT ». *Journal of Artificial Intelligence Research*, 32 :565–606, juillet 2008. 4.2.1

[YUE & JOACHIMS 2009] Y. YUE et T. JOACHIMS. « Interactively Optimizing Information Retrieval Systems as a Dueling Bandits Problem ». Dans *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1201–1208, New York, NY, USA, 2009. Association for Computing Machinery. 2.4, 2.4.2

Résumé

La programmation par contraintes est déclarative : l'utilisateur modélise le problème (sous forme de contraintes) sans se soucier de la partie résolution. La résolution est déléguée à un programme générique appelé solveur de contraintes.

Les solveurs de contraintes utilisent des mécanismes issus de l'intelligence artificielle pour guider ses choix (heuristiques), et pour réduire efficacement l'espace de recherche (inférence). Les solveurs de contraintes récents sont équipés de plusieurs heuristiques. Le choix de la meilleure heuristique pour résoudre une instance de problème donnée est actuellement délégué à l'utilisateur.

L'objectif de cette thèse est de rendre les solveurs de contraintes plus autonomes. Nous voulons enlever toute charge cognitive à l'utilisateur concernant le processus de résolution. Pendant la recherche de solutions, le solveur redémarre régulièrement la recherche pour éviter les impasses coûteuses. Nous proposons de sélectionner automatiquement l'heuristique la plus prometteuse après chaque redémarrage.

En réponse à ce problème de choix séquentiel, nous nous inspirons des politiques traitant du problème du bandit multi-bras. De telles politiques sont issues de l'apprentissage par renforcement et permettent d'explorer un ensemble d'actions tout en exploitant, le plus souvent possible, l'action la plus prometteuse.

Après avoir défini comment incorporer cet outil d'apprentissage automatique dans un solveur, nous proposons plusieurs applications pour rendre les solveurs de contraintes modernes plus autonomes. Premièrement, nous appliquons les politiques proposées dans la littérature sur les bandits multi-bras pour apprendre la meilleure heuristique parmi un ensemble d'heuristiques fournies par le solveur. Sur la base de cette première application, nous proposons une nouvelle politique qui est plus cohérente avec la structure des solveurs actuels. Toujours en utilisant des politiques de bandit, nous proposons d'adapter la quantité d'aléa nécessaire pour perturber une heuristique donnée et la rendre plus efficace. Enfin, nous étendons ces résultats aux solveurs de contraintes sous optimisation.

Mots-clés : Heuristique de Branchement, Apprentissage, Solveurs et Outils, Évaluation et Analyse, Optimisation.

Abstract

Constraint programming is declarative : the user models the problem (in the form of constraints) without worrying about the solving part. The solving is delegated to a generic program called a constraint solver.

Constraint solvers use mechanisms from artificial intelligence to guide its choices (heuristics), and to efficiently reduce the search space (inference). Recent constraint solvers are equipped with several heuristics. Choosing the best heuristic to solve a given problem instance is currently delegated to the user.

The goal of this thesis is to make constraint solvers more autonomous. We want to remove any cognitive burden from the user regarding the solving process. While searching for solutions, the solver often restarts the search to avoid expensive dead-ends. We propose to automatically select the most promising heuristic after each restart.

In response to this sequential choice problem, we take inspiration from policies addressing the multi-armed bandit problem. Such policies are derived from reinforcement learning and allow to explore a set of actions while exploiting the most profitable one as often as possible.

After having defined how to incorporate this machine learning tool in a solver, we propose several applications to make modern constraint solvers more autonomous. First, we apply the policies proposed in the multi-armed-bandits literature to learn the best heuristic among a set of heuristics provided by the solver. Based on this first application, we propose a new policy that is more consistent with the structure of current solvers. Still using bandit policies, we propose to adapt the amount of randomization needed to perturb a given heuristic and make it more efficient. Finally, we extend these results to constraint optimization solvers.

Keywords : Branching Heuristic, Machine Learning, Solvers and Tools, Evaluation and Analysis, Optimization.

CENTRE DE RECHERCHE EN INFORMATIQUE DE LENS – CNRS UMR 8188
Université d'Artois, rue Jean Souvraz, S.P. 18 F-62307, Lens Cedex France
Secrétariat : +33 (0)3 21 79 17 23
http://www.cril.univ-artois.fr