UNIVERSITÉ D'ARTOIS

Solveurs de Contraintes Autonomes

MÉMOIRE

soutenu le 6 septembre 2018

pour l'obtention du

Master Informatique de l'Université d'Artois

Parcours Intelligence Artificielle

par

Hugues Wattez

Encadrants :	Frédéric Koriche	Professeur des Universités
	Christophe Lecoutre	Professeur des Universités
	Anastasia Paparrizou	Chargé de recherche

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 : Tél.: +33 (0)3 21 79 17 23 – Fax : +33 (0)3 21 79 17 70
http://www.cril.univ-artois.fr

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

Remerciements

Je souhaite remercier, pour leurs conseils et leur disponibilité, mes encadrants : Frédéric KORICHE, Christophe LECOUTRE et Anastasia PAPARIZZOU.

Je souhaite également remercier l'ensemble des enseignants, de l'IUT à la faculté Jean Perrin de Lens, pour les connaissances qu'ils m'ont prodiguées.

Enfin, je souhaite remercier mes lecteurs et correcteurs pour leur patience et leur participation à ce rapport.

Table des matières

Annexes

Introduction générale

Nous sommes dans un contexte où l'informatique est en continuelle expansion. Malgré cette croissance, certains problèmes demeurent et atteignent une complexité supérieure aux capacités actuelles de nos ordinateurs. Nous allons étudier l'un d'eux : le Problème de Satisfaction de Contraintes (CSP).

Les CSP sont des problèmes combinatoires dont la complexité peut croître très vite. En effet, les problèmes CSP font partie des problèmes NP-complets les plus connus. Pour remédier en partie à cette complexité, il existe des solveurs CSP. Ces solveurs permettent, à travers différentes stratégies de recherche, la résolution de certains d'entre eux en un temps raisonnable.

Un point important dans le choix d'une bonne stratégie concerne, par exemple, la sélection de l'heuristique de recherche. Une heuristique savamment choisie permet de réduire significativement la taille de l'espace de recherche. Cependant, sélectionner une heuristique n'est pas chose simple. De plus, de part leur nature, aucune heuristique ne peut être continuellement meilleure qu'une autre. Cela induit qu'utiliser une seule et même heuristique pour un large problème peut être désavantageux quant à l'efficacité de la recherche.

L'idée des travaux que nous menons tend à remplacer le rôle de l'humain par un mécanisme intelligent, permettant d'orienter automatiquement le solveur au cours de sa recherche. Ce mécanisme s'inspire du problème des Bandits Multi-Bras. Le solveur apprend et sélectionne lui-même une heuristique parmi un ensemble, le but étant d'automatiser le choix de l'heuristique produisant les meilleures performances. Nous voyons par la suite que l'heuristique n'est pas la seule caractéristique que nous tentons de rendre autonome.

Dans une première partie, nous étudions les CSP et les mécanismes employés par les solveurs pour résoudre efficacement les problèmes posés. Nous voyons aussi les algorithmes répondant au problème des Bandits Multi-Bras et leurs implantations actuelles dans les solveurs CSP. Dans une seconde partie, nous expérimentons notre solution associant un algorithme répondant au problème des Bandits Multi-Bras. Enfin, nous étudions le résultat des expérimentations afin de mieux les comprendre.

Notation

La Table 1 représente les différentes notations visibles dans ce rapport entre le domaine des contraintes et le domaine des Bandits Multi-Bras.

Domaine	Symbole	Signification
Contraintes	x, y, z	Variables
	$dom(x)$	Domaine de x
	$dom^{init}(x)$	Domaine initial de x
	S_i	Ensemble i
	$\mathcal{R}$	Relation
	c	Contrainte
	$scp(c)$	Scope de la contrainte c
	$rel(c)$	Relation décrivant l'ensemble des tuples autorisés par c
	P	Instance d'un problème de satisfaction de contraintes
	$sols(P)$	Ensemble des solution du problème P
	$vars(P)$	Ensemble des variables du problème P
	$ctrs(P)$	Ensemble des contraintes du problème P
	ϕ	Fonction d'AC-fermeture
Bandit	a_t	Action correspondant à l'instant t
	T	Nombre de tours jouables
	K	Nombre d'actions disponibles
	$r_t(a_t)$	Récompense à l'instant t pour l'action a_t
	$Regret_T$	Regret des actions sélectionnées par le joueur
	A	L'ensemble des actions
	R	Ensemble des fonction de récompense
	$\mathbb{P}(a_t)$	Probabilité associée à l'action a_t d'obtenir une récompense
	$\mathbb{E}$	Espérance mathématique
	$\mu(a)$	Moyenne réelle des récompenses $r(a)$ pour l'action a
	μ^*	Moyenne des récompenses de la meilleure action
	Δ_a	Intervalle de sous-optimalité du choix a
	$n_t(a)$	Nombre de fois où le joueur a sélectionné l'action a durant les t premières étapes
	$\widehat{\mu}_t$	Moyenne empirique de $r(a)$ sur les $n_t(a)$ jeux
	md_t	Nombre de mauvaises décisions prises lors de la t^{e} exécution
	$ng_{i,t}$	Nombre de *nogoods* de taille i générés lors de la t^{e} exécution

TABLE 1 – Les différentes notations du domaine de recherche

Première partie

Étude bibliographique

Chapitre 1

Le problème de satisfaction de contraintes CSP

Sommaire

1.1 Introduction au CSP

Comme nous l'avons dit, le problème de satisfaction de contraintes est un problème appartenant à la famille des problèmes NP-complets [CORMEN *et al.* 2009]. À travers les mécanismes que nous décrivons dans la suite, nous rendons compte de l'avancement actuel des solveurs CSP modernes, mais aussi des avancées que nous envisageons d'apporter.

Nous définissons dans un premier temps ce qu'est un réseau de contraintes, pour en arriver à la description globale du problème. Puis, nous étudions les mécanismes actuels permettant de résoudre en temps raisonnable ce genre de problème et insistons sur ceux que nous pensons perfectionner.

1.2 Réseaux de contraintes

1.2.1 Définitions et notations

Définition 1 (Variable)**.** *Une* variable *est une entité à laquelle on associe une valeur. Cette valeur appartient au domaine courant de x, noté $dom(x)$.*

Remarque 1. *Le domaine d'une variable évolue au cours du temps mais est toujours inclus dans le domaine initial de la variable. Nous le notons $dom^{init}(x)$.*

Exemple 1. *Soient les variables x et y ayant pour domaine courant $dom(x) = \{1, 2\}$ et $dom(y) = \{2, 3\}$. Leurs domaines initiaux pourraient être $dom^{init}(x) = dom^{init}(y) = \{1, 2, 3, 4\}$.*

Définition 2 (Tuple)**.** *Un* tuple *τ est une séquence de valeurs (séparées par des virgules et entourée par des parenthèses). 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 $\tau[i]$.*

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 2. *Soient x, y et z trois variables telles que $dom(x) = dom(y) = \{a, c\}$ et $dom(z) = \{a, b\}$. Alors, nous obtenons :*

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

Définition 4 (Relation)**.** *Une* relation *$\mathcal{R}$ 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 : $\mathcal{R} \subseteq \prod_{i=1}^{r} S_i$.*

Exemple 3. *Soit $\mathcal{R}$ définie sur $dom(x) \times dom(y) \times dom(z)$:*

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

Définition 5 (Contrainte)**.** *Une* contrainte c *est définie par un ensemble (totalement ordonné) de variables, appelé* scope *de* c *et noté* $scp(c)$*, et par une* relation $rel(c)$ *qui décrit l'ensemble des tuples autorisés par* c *pour les variables de son scope.*

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

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

Définition 6 (Contrainte en intention)**.** *Une contrainte* c *est* intentionnelle *(ou* définie en intention*), si celle-ci est décrite par une formule booléenne (prédicat) qui représente une fonction qui est définie à partir de* $\prod_{x \in scp(c)} dom^{init}(x)$ *et retournant* faux *ou* vrai*.*

Exemple 4. *Soient les variables* a*,* b*,* x*,* y *et* z *ayant pour domaine* $dom(a) = dom(b) = dom(x) = dom(y) = dom(z) = \{0, 1, 2\}$*.*

Une contrainte binaire en intension : $a = b + 1$*.*

Une contrainte ternaire en intension : $x = y \bigwedge y = z$*.*

Définition 7 (Contrainte en extension)**.** *Une contrainte* c *est* extensionnelle *(ou* définie en extension*) si elle est explicitement décrite :*

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

Exemple 5. *Si* $dom(x) \times dom(y) \times dom(z) = \{0, 1, 2\} \times \{0, 1, 2\} \times \{0, 1, 2\}$*, alors la contrainte ternaire de l'exemple 4 peut être définie positivement comme :*

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

Remarque 2. *Les contraintes en extension sont aussi appelées* contraintes tabulaires*.*

Définition 8 (Contrainte globale)**.** *Une* contrainte globale *est une contrainte définie par un type de relations qui représente une sémantique mathématique bien précise. Une contrainte globale peut être appliquée sur un nombre arbitraire de variables.*

Par exemple, la sémantique de *allEqual* est telle que toutes les variables doivent prendre la même valeur.

Exemple 6. *La contrainte ternaire de l'exemple 4 peut être définie par :*

$$allEqual(x, y, z)$$

Remarque 3. *De nombreuses contraintes globales existent telles que* allDifferent*,* ordered*, etc.*

Définition 9 (Instance CSP)**.** *Une* instance P d'un Problème de Satisfaction de Contraintes *(CSP), aussi appelée Réseau de Contraintes (CN), est définie par :*

- *un ensemble fini de variables, noté* $vars(P)$*;*
- *un ensemble fini de contraintes, noté* $ctrs(P)$*, tel que* $\forall c \in ctrs(P), scp(c) \subseteq vars(P)$*.*

Définition 10 (Solution d'une instance CSP)**.** *Une* solution d'un réseau de contraintes P *correspond à l'assignation d'une valeur à chaque variable de* P *telle que toutes les contraintes de* P *sont satisfaites. L'ensemble des solutions de P est noté* $sols(P)$*.*

Remarque 4. *Pour simplifier, un couple* (x, a) *avec* $x \in vars(P)$ *et* $a \in dom(x)$ *sera appelée un* littéral de P *(ou une* valeur de P*).*

Définition 11 (Instance COP)**.** *Une* instance P d'un Problème d'Optimisation de Contraintes *(« Constraint Optimisation Problem », COP) est composée de :*

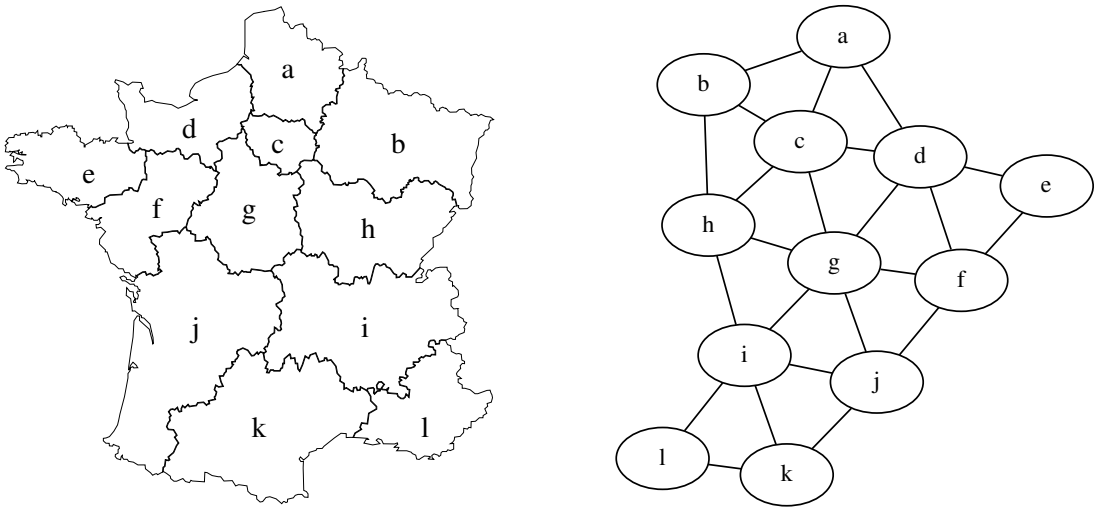
- *un ensemble fini de variables, noté* $vars(P)$*;*
- *un ensemble fini de contraintes, noté* $ctrs(P)$*, tel que* $\forall c \in ctrs(P), scp(c) \subseteq vars(P)$*;*
- *une fonction objectif o, aussi notée* $obj(P)$*, qui doit être maximisée ou minimisée.*

Remarque 5. *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 représentant la qualité de chacune. L'objectif est alors de trouver la solution maximisant ou minimisant cette fonction. Il s'agit la plupart du temps d'une fonction maximisant (resp. minimisant) la somme ou le produit de certaines variables du problème associées à des coefficients.*

Définition 12 (Solution d'une instance COP)**.** *Une* solution d'une instance COP*, notée* P*, correspond à une solution du problème sous-jacent CSP. Pour la minimisation (resp. maximisation), une* solution optimale *de* P *est une solution pour laquelle la valeur objectif est inférieure ou égale (resp. supérieure ou égale) à la valeur de toute autre solution.*

1.2.2 Exemple concret : le problème de coloration de carte

Le problème de coloration de carte est un exemple connu. Nous disposons d'une carte plane découpée en régions. L'objectif du problème est de colorier chaque région de façon à ce que deux régions adjacentes ne soient pas coloriées de la même couleur, le tout en minimisant le nombre de couleurs utilisées.



(a) Carte de France découpée en régions

(b) Graphe équivalent à la carte de France

FIGURE 1.1 – Représentations de la France en régions

Afin de simplifier ce problème, nous nous ramenons à une représentation classique sous forme de graphe (Figure 1.1b).

Il parait maintenant naturel de formaliser le problème sous forme d'un réseau de contraintes. En effet, nous pouvons représenter chaque région par une variable, comme nous le voyons dans les Figures

1.1a et 1.1b. Chaque variable est composée d'un domaine correspondant au nombre de couleurs disponibles. Nous savons par avance que nous n'aurons pas besoins de plus de quatre couleurs [WILSON & NASH 2003], ce qui nous permet de définir le domaine de chaque variable comme étant l'ensemble des valeurs suivantes : $\{0, 1, 2, 3\}$ où

— 0 correspond à la couleur noire ;
— 1 correspond à la couleur gris foncée ;
— 2 correspond à la couleur gris claire ;
— 3 correspond à la couleur blanche.

Tout comme nous le montre le Graphe 1.1b, chaque région est représentée par un sommet et chaque arête représente le fait que deux régions soient adjacentes.

De manière plus formelle, soit $G = (S, A)$ tel que $S = \{a, b, ..., l\}$ et $A = \{(x, y) : x, y \in S, x \neq y,$ x et y sont adjacents $\}$.

Voici notre instance CSP :

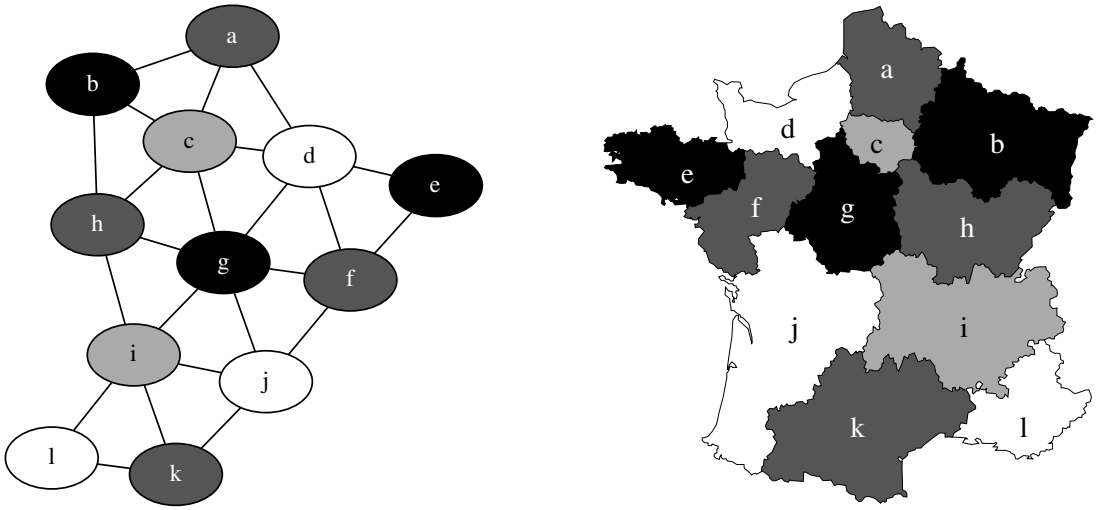
$$vars(P) = S = \{a, b, ..., l\} \text{ avec } \forall x \in P, dom(x) = \{0, 1, 2, 3\}$$

$$ctrs(P) = \left\{ \begin{array}{ll} c_{ab} : a \neq b & c_{ac} : a \neq c \\ c_{ad} : a \neq d & c_{bc} : b \neq c \\ c_{bh} : b \neq h & c_{cd} : c \neq d \\ c_{cg} : c \neq g & c_{ch} : c \neq h \\ c_{de} : d \neq e & c_{df} : d \neq f \\ c_{dg} : d \neq g & c_{ef} : e \neq f \\ c_{fg} : f \neq g & c_{fj} : f \neq j \\ c_{gh} : g \neq h & c_{gi} : g \neq i \\ c_{gj} : g \neq j & c_{hi} : h \neq i \\ c_{ij} : i \neq j & c_{ik} : i \neq k \\ c_{il} : i \neq l & c_{jk} : j \neq k \\ c_{kl} : k \neq l & \end{array} \right\}$$

Les précédentes contraintes interdisent à deux régions voisines de posséder la même couleur. Celles-ci sont représentées sous forme de contraintes binaires en intension avec le prédicat d'inégalité $\neq$.

P est cohérente, autrement dit, elle possède une solution. La figure 1.2a montre une solution : chacun des sommets est colorié en respectant les contraintes précédentes.

Par transformation inverse du graphe en notre carte de France originelle, nous obtenons une carte où toute région possède une couleur différente de ses régions voisines (figure 1.2b).



(a) Graphe équivalent à la carte de France

(b) Carte de France découpée en régions

FIGURE 1.2 – Représentations de la France en régions colorées

1.3 Inférence

En programmation par contraintes, dès que la taille d'une instance devient importante, il est impossible de tester toutes les combinaisons de l'arbre de recherche : cela engendrerait une explosion combinatoire. C'est ici qu'intervient le principe d'inférence. Il permet de filtrer le réseau de contraintes en évitant de parcourir des zones inutiles.

Chaque contrainte peut être considérée comme un « sous-problème » à partir duquel des valeurs incohérentes peuvent être ignorées. Il existe différents niveaux d'inférence, par exemple :

— AC (Cohérence d'Arc) : toute valeur incohérente est identifiée et supprimée ;
— BC (Cohérence aux Bornes) : toute valeur incohérente correspondant aux bornes des domaines est identifiée et supprimée.

Il s'agit d'une liste non exhaustive des principes d'inférence.

1.3.1 Notion de support

Définition 13 (Support)**.** *Pour une contrainte c :*

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

1.3.2 Arc-cohérence

Définition 14 (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 (x, a) sur c, i.e., un support τ sur c tel que $\tau[x] = a$.*

Définition 15 (Valeur arc-incohérente)**.** *Une valeur (x, a) est* arc-incohérente *pour une contrainte c quand elle n'admet pas de support (x, a) sur c.*

Définition 16. *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 renforce/établie l'arc-cohérence sur c.*

Algorithme 1 : filtreAC(c : Contrainte)

```
1 pour chaque variable x ∈ scp(c) faire
2     pour chaque valeur a ∈ dom(x) faire
3         si ¬seekSupport(c, x, a) alors
4             supprimer a de dom(x)
```

L'Algorithme 1 représente le déroulement classique d'un filtrage avec la propriété AC. Il en existe d'autres utilisant une propriété plus forte telle que SAC, PIC, MaxRPC, etc.

1.3.3 Propagation de contraintes

Définition 17. *Un réseau de contraintes P est AC si, et seulement si, chacune des contraintes de P est AC.*

Définition 18. *Calculer la* AC-fermeture *sur un réseau de contraintes P est le fait de supprimer toutes les arc-incohérences de P.*

Nous pouvons trouver différentes implantations de l'algorithme *AC-fermeture* dans la littérature, telle que AC3 [MACKWORTH 1977], AC2001 [BESSIÈRE *et al.* 2005], etc.

1.3.4 *Nogood*

Définition 19. *Un* nogood *pour un réseau de contraintes P est une instanciation d'un sous-ensemble de variables de P qui ne peut pas être étendu à une solution.*

Définition 20. *Une consistance (locale) est une propriété définie dans les réseaux de contraintes. Typiquement, elle révèle certains* nogoods.

Remarque 6. *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.*

1.4 Stratégies de recherche

Pour un réseau de contraintes P où n est le nombre de variables, d est la taille du plus grand domaine, e est le nombre de contraintes et r est la taille de la plus grande contrainte, la méthode « Générer et Tester » (méthode générant l'ensemble des solutions candidates et vérifiant linéairement chacune d'elles) revient à une complexité de $O(d^n er)$. Une telle complexité n'est pas permise en pratique. Cela nous oblige donc à passer par des mécanismes d'inférence permettant la réduction de notre espace de recherche.

Il existe deux grandes méthodes de recherche :

— *complète* : une exploration complète de l'espace de recherche ;

— *incomplète* : une exploration partielle de l'espace de recherche.

Pour notre étude, nous nous concentrons uniquement sur la recherche complète.

1.4.1 Recherche avec retour en arrière

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 (*backtracking*) et une succession de décisions et de propagations.

Dans ce type de recherche, il existe deux façons de parcourir l'arbre de recherche. L'une d'elles (que nous employons pour cette étude), consiste en une recherche binaire. Chaque nœud de l'arbre est séparé en deux branches, couvrant deux décisions complémentaires (par exemple : $x = 1$ et $x \neq 1$).

Algorithme 2 : binary-ϕ-search(P : CN)

```
1 P ← φ(P)
2 si P = ⊥ alors
3 |   retourner false
4 si ∀x ∈ vars(P), |dom(x)| = 1 alors
5 |   retourner true
6 sélection d'une valeur (x, a) de P tel que |dom(x)| > 1
7 retourner binary-φ-search(P|x=a) ∨ binary-φ-search(P|x≠a)
```

ϕ représente la fonction d'*AC-fermeture*.

Il existe différentes variantes de l'Algorithme 2, dont les plus connus sont Backtracking (BT), Forward Checking [HARALICK & ELLIOTT 1980] (FC) et Maintaining Arc Consistency [SABIN & FREUDER 1994] (MAC).

Nous utilisons ce dernier durant les expérimentations. MAC maintient l'arc-cohérence tout au long de la recherche.

1.4.2 Recherche guidée par les heuristiques

Le but d'une heuristique est de minimiser la taille de l'arbre de recherche.

Il est important de savoir que l'ordre dans lequel les variables sont affectées lors d'une recherche avec retour en arrière, est reconnu comme étant un élément clé. L'usage d'une heuristique plutôt qu'une autre peut amener à une différence drastique en terme d'efficacité. Le simple fait d'introduire de l'aléatoire au sein d'une heuristique permet de large variabilité de performance.

De façon générale, quand nous menons une recherche avec retour en arrière, nous sollicitons une heuristique ordonnant le choix des variables suivantes, mais aussi une autre heuristique ordonnant le choix de la valeur de cette variable.

Les principales règles à respecter pour obtenir une heuristiques efficace sont :

— « Pour réussir, essaie en premier là où tu as le plus de chance de faillir » [HARALICK & ELLIOTT 1980] : il est toujours préférable d'affecter en premier les variables appartenant à la partie difficile du problème ;

— pour trouver rapidement une solution, il est préférable d'affecter la première valeur permettant d'amener au plus vite à une solution ;

— le choix des premières variables/valeurs initiales est particulièrement importante.

Dans ce cadre, une heuristique associe à chaque variable un score. De cette façon, nous pouvons choisir une variable possédant le score minimum ou maximum. En cas d'égalité, il existe un mécanisme de départage (*tie-break*), tel que *lexico*, correspondant au choix de la variable dont le nom se trouve lexicalement avant tout autre nom de variable.

Il existe trois grandes catégories d'heuristiques dans la recherche avec retour en arrière, décrites ci-dessous.

Heuristiques statiques

Les heuristiques statiques associent un score à chaque variable avant le début de la recherche. Ce score reste statique tout au long de la recherche. Nous pouvons citer :

— *lexico* comme vu précédemment ;

— *deg* associant un score à chaque variable égal au nombre de contraintes où la variable est impliquée (choix du plus haut score) ;
— *ddeg* associant à chaque variable, le rapport $\frac{taille\,du\,domaine\,de\,la\,variable}{nombre\,de\,contraintes}$ (choix du plus faible score).

Heuristiques dynamiques

Les heuristiques dynamiques calculent un score à chaque nœud de l'arbre de recherche. Nous pouvons citer *dom* considérant à chaque nœud, la taille du domaine de chaque variable. La variable au plus petit domaine est sélectionnée. L'heuristique *dom/ddeg* [BESSIÈRE & RÉGIN 1996] est dérivée de cette première et de l'heuristique statique *ddeg*.

Heuristiques adaptatives

Enfin, il existe les heuristiques adaptatives utilisant l'état courant des variables, mais aussi leur historique. Parmi les plus connues, nous avons :
— *wdeg* [BOUSSEMART *et al.* 2004] et *dom/wdeg*, donnant un poids à chaque contrainte en fonction de leur implication dans des conflits ;
— *impact* [REFALO 2004] prend en compte l'impact qu'a une variable sur la réduction de l'espace de recherche ;
— *activity* [MICHEL & VAN HENTENRYCK 2012] combine l'impact d'une variable et la fréquence à laquelle son domaine est modifié ;
— *lc [*LECOUTRE et al. *2009] (« last conflicts »)* retient les dernières affectations qui ont faillies (cette heuristique est utilisée conjointement à une autre).

Heuristiques pour les valeurs

De même, pour le choix d'une valeur à affecter à une variable, il existe des heuristiques comme *min-conflict* et *max-conflict* qui orientent la recherche vers la valeur qui est associée à peu ou beaucoup de conflits. D'autres heuristiques plus naïves existent, telles que *lexico* et *random*.

1.4.3 Redémarrage

Il faut garder en tête que les premières décisions sont primordiales. En effet, il est compliqué de se débarrasser de mauvaises décisions prises en début de recherche. C'est pourquoi, il est utile de reprendre régulièrement la recherche depuis la racine de l'arbre. Cette action permet d'élaguer une partie de l'arbre de recherche ne menant pas à une solution, de diversifier, de collecter des informations et de tester diverses heuristiques.

Le redémarrage se fait périodiquement en fonction du nombre de décisions prises, du nombre de *backtrack* ou encore en fonction du temps. Afin d'éviter de rechercher inutilement deux fois dans la même portion de l'espace de recherche (pour la cas d'une recherche complète), nous enregistrons des *nogoods*.

La possibilité de changer d'heuristique à chaque redémarrage est un point essentiel pour cette étude. En effet, c'est à ce passage précis que nous allons, par la suite, introduire un mécanisme qui se charge de choisir intelligemment cette composante et d'autres associées.

1.5 Conclusion

Nous constatons qu'il existe une grande diversité de configuration pour mener à bien une recherche. En effet, un solveur proposant, par exemple :

— 4 algorithmes de propagation de contraintes ;
— 5 heuristiques de choix de variables ;
— 3 heuristiques de choix de valeurs ;
— 5 tableaux de variables à positionner en début de recherche,

propose à l'utilisateur 300 configurations possibles. A moins que cet utilisateur soit un expert, il est primordial que cette tâche lui soit soustraite afin d'être correctement exécutée dans le but d'une recherche efficace.

Chapitre 2

Bandits Multi-Bras et solveurs CSP

Sommaire

2.1 Introduction aux Bandits Multi-Bras

Nous nous intéressons à l'apprentissage par renforcement. Un agent, dit autonome, apprend de ses actions grâce à des expériences menées dans son environnement. A chaque expérience, l'environnement donne une récompense quantitative (positive ou négative) à l'agent. Ainsi, à travers ses expériences et les récompenses associées, l'agent cherche à optimiser sa politique en maximisant la somme de ses récompenses.

L'apprentissage par renforcement fait donc face à un dilemme entre *exploration* et *exploitation*. 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 meilleure action.

La *mesure de regret* a été introduite afin de contrer ce problème d'équilibre : le regret représente la perte relative liée aux actions ne correspondant pas à la politique optimale.

Le problème des Bandits Multi-Bras [GITTINS 1989] illustre parfaitement le dilemme entre exploration et exploitation. Il s'agit d'un problème très souvent étudié en théorie des jeux, mais aussi en intelligence artificielle, plus particulièrement en apprentissage par renforcement, où il trouve une place fondamentale. Ce qui fait la particularité de ce problème est qu'il est prouvé que le regret grandit, au plus, de façon logarithmique en fonction du nombre d'étapes.

Les algorithmes que nous présentons dans la suite respectent ce seuil de regret et possèdent une politique simple et efficace.

2.2 Préliminaires

2.2.1 Problème des Bandits

Le problème des Bandits Multi-Bras (*Multi-Arm Bandits*, MAB) trouve son utilité originelle au sein des casinos avec les machines à sous. En effet, l'intitulé du problème n'est pas sans rappeler le bandit manchot étant une autre dénomination du jeu de hasard. Un joueur souhaite choisir une machine parmi d'autres afin de maximiser son gain sur une séquence de jeux.

Dans le cas des machines à sous, tout comme dans le choix d'une heuristique pour un solveur CSP, il nous faut trouver un équilibre entre exploration et exploitation. Un algorithme implantant un MAB se doit d'exploiter le plus souvent possible le bras lui donnant la récompense maximale. Néanmoins, si celui-ci ne visite pas assez les autres possibilités, il peut rater une opportunité. C'est pourquoi la recherche d'un équilibre est primordiale.

Le problème des machines à sous se résume en un jeu à somme nulle entre un joueur et son adversaire. A chaque instant t, le joueur choisit une action $a_t \in \{1, ..., T\}$ où T correspond au nombre d'actions (machines à sous) disponibles. Simultanément, l'environnement choisit une fonction de récompense r_t : $\{1, ..., T\} \to \mathbb{R}$. Ensuite, le joueur reçoit une récompense $r_t(a_t)$ et observe un *feedback*. Dans le cas où nous aurions accès à l'ensemble des informations, la récompense de chaque action serait dévoilée. Or, dans le cas du MAB, seule la récompense liée à notre action $r_t(a_t)$ est révélée.

Description globale

Le joueur a accès à un ensemble d'actions A (machines, heuristiques, etc.). L'environnement a accès à un ensemble $R \subseteq \mathbb{R}^A$ de fonctions de récompense.

Algorithme 3 : Abstraction du bandit

```
pour chaque t allant de 1 à T faire
    Choix de a_t ∈ A (possiblement aléatoire)
    L'environnement choisit r_t ∈ R
    Observation du feedback r_t(a_t)
```

Le but du joueur est de minimiser son *regret* :

$$Regret_T = \max_{a \in A} \sum_{t=1}^{T} r_t(a) - \sum_{t=1}^{T} r_t(a_t)$$

Consistance de Hannan

Un joueur est dit *Hannan consistant* si, pour chaque séquence de défaites choisie par l'environnement, le regret du joueur est toujours sous-linéaire en T :

$$Regret_T = o(T) \text{ tel que } \lim_{T \to \infty} \frac{Regret_T}{T} = 0$$

2.2.2 Terminologie

Il existe différentes variantes du problème des bandits, dépendant chacune du type d'actions, de récompenses et de *feedbacks*.

Une action peut être :
— *explicite* : A est un ensemble de K actions ;
— *implicite* : A est un espace combinatoire (sous-ensembles, chemins, permutations, etc.), construit à partir d'un ensemble K d'éléments atomiques (nœuds, arêtes, rangs, etc.), tel que $A \subseteq \{0,1\}^K$.

L'environnement peut être :
— *stochastique* : l'environnement est incertain, R est un ensemble de $|A|$ variables indépendantes et aléatoires avec des distributions fixes mais inconnues ;
— *concurrentiel* : l'environnement est adaptatif, il choisit les récompenses r_t dans R en fonction de l'historique des actions sélectionnées par le joueur.

Les *feedbacks* peuvent provenir d'un :
— *bandit* : à chaque étape t, le joueur observe uniquement une valeur $r_t(a_t)$;
— *semi-bandit* : à chaque étape t, le joueur observe le produit $r_t \circ a_t$ (ici a_t est une action combinatoire décrite par un vecteur).

Dans cette étude, nous ne nous intéressons qu'aux bandits stochastiques. Nous supposons que l'environnement est incertain et aléatoire mais qu'il existe bel et bien une heuristique meilleure que les autres.

2.3 Algorithmes des bandits stochastiques

2.3.1 Généralités

Dans le cadre des bandits stochastiques, nous pouvons simplifier le protocole en nous abstrayant de l'environnement (Algorithme 4). Celui-ci possède une distribution fixe de probabilités sur les différentes récompenses liées aux bras. Cette distribution fixe reste pour autant inconnue.

Algorithme 4 : Abstraction des bandits stochastiques

```
pour chaque t allant de 1 à T faire
    Choix de a_t ∈ A (possiblement aléatoire)
    Observation du feedback r_t(a_t) ~ ℙ(a_t)
```

$\mathbb{P}(a_t)$ correspond à la récompense exacte associée à l'action a_t.
Le but du joueur est de minimiser son *regret* :

$$PseudoRegret_T = \max_{a \in A} \sum_{t=1}^{T} \mathbb{E}[r_t(a)] - \sum_{t=1}^{T} \mathbb{E}[r_t(a_t)]$$

Il est à noter que l'espérance, notée $\mathbb{E}$, est relative au joueur tout comme au choix aléatoire de la récompense.

Définissons quelques notations :
— $\mu(a)$ est la moyenne (réelle) de la récompense $r(a)$;
— μ^* est la moyenne de la meilleure action ;
— $\Delta_a = \mu^* - \mu(a)$ est l'intervalle de sous-optimalité du choix a ;
— $n_t(a)$ est le nombre de fois où le joueur a sélectionné a durant les t premières étapes ;
— $\widehat{\mu}_t$ est la moyenne empirique de $r(a)$ sur les $n_t(a)$ jeux.
À l'état $t = 0$, pour toute action a, $n_t(a)$ et $\widehat{\mu}_t$ sont initialisés à 0

2.3.2 Epsilon-greedy

Epsilon-greedy [SUTTON & G. BARTO 1998] correspond à la stratégie la plus simple et naïve afin de venir à bout du problème des bandits stochastiques. Il suffit de renseigner une valeur $\varepsilon \in [0, 1]$, permettant de savoir dans quelle proportion nous souhaitons explorer ou exploiter. Typiquement, $\varepsilon = 0.1$ mais cette valeur peut varier en fonction des circonstances.

Algorithme 5 : epsilon-greedy

```
pour chaque t allant de 1 à T faire
    Choix d'un paramètre ε_t ∈ [0, 1]
    Jouer avec la probabilité (1 − ε_t) l'action qui maximise μ̂_{t−1}(a)
    Jouer avec la probabilité ε_t une action choisie aléatoirement
```

Proposition 1. *En admettant que* $d < \min_{j \neq i^*} \Delta_j$ *et qu'on initialise* $\varepsilon_t = \frac{n}{(d^2 t)}$ *où* n *correspond au nombre d'actions disponibles, epsilon-greedy accomplit un* pseudo-regret *en* $O(\frac{n \ln(T)}{d^2})$. *Une borne inférieure* d *basée sur l'intervalle minimal doit être connue à l'avance. Le regret est seulement logarithmique en T.*

Le joueur utilisant cet algorithme peut donc être considéré comme *Hannan consistant*.

2.3.3 UCB standard

Avec ucb [AUER *et al.* 2002], il n'est plus nécessaire de renseigner un ε afin de faire varier l'exploration et l'exploitation. En effet, tout est compris dans la principale formule de l'algorithme suivant :

Algorithme 6 : ucb

```
pour chaque t allant de 1 à T faire
    Choix de l'action i maximisant μ̂_{t−1}(i) + sqrt(8 ln(t) / n_{t−1}(i))
```

$\widehat{\mu}_{t-1}(i)$ correspond à la partie exploitation de l'algorithme, tandis que $\sqrt{\frac{8 \ln(t)}{n_{t-1}(i)}}$ correspond à la partie exploration.

Proposition 2. *UCB accomplit un* pseudo-regret *de* $O(\sqrt{KT})$.

2.4 Implantation actuelle des bandits dans les solveurs

2.4.1 Propagation de contraintes

La *propagation de contraintes* est l'un des procédés permettant de réduire efficacement la taille de l'arbre de recherche. Il existe différents niveaux de propagation, correspondant à différents algorithmes (1.3.3). En pratique, il est connu que changer de niveau de propagation au cours de la recherche produit de bien meilleurs résultats que dans le cas statique.

Dans ces circonstances, l'application du MAB permet de dynamiser ce choix au cours de la recherche sans pour autant faire appel à un expert ou prédéfinir des paramètres.

Le procédé appliqué dans [BALAFREJ *et al.* 2015] correspond au positionnement de différents MAB à différentes profondeurs de l'arbre de recherche. Ainsi, chaque MAB choisit le niveau de propagation

le plus adéquat en fonction de sa position dans l'arbre, à l'aide de l'implantation d'ucb. La fonction de récompense est basée sur le temps nécessaire au renforcement de la cohérence locale.

Nous retrouvons la description du MAB faite à travers cette étude. Plus nous nous enfonçons dans la recherche, plus les différents MAB gagnent en connaissances et plus fines deviennent les décisions dans le choix du niveau de propagation optimal.

Les résultats de cette étude démontrent que cette approche apporte une bien meilleure efficacité et rend le solveur plus stable.

2.4.2 Choix d'heuristique

L'étude proposée dans [XIA & YAP 2018] reprend le sujet même de notre recherche : l'usage des bandits pour le choix d'une heuristique dans le but de réduire l'espace de recherche.

Pour leur étude, deux algorithmes des bandits stochastiques ont été implantés sont l'un d'entre eux est ucb. À chaque nœud de la recherche, une heuristique est choisie par l'algorithme et une récompense lui est décernée. Cette récompense est basée sur le nombre de nœuds parcourus dans le sous-arbre de recherche.

Les conclusions de l'étude montrent que l'usage dynamique des différentes heuristiques d'un solveur CSP le rend plus robuste et permet de meilleures performances que dans le cas d'un choix statique. De plus, l'usage des bandits dans le choix d'une fonction de propagation tout comme dans le choix d'une heuristique, évite l'implication d'un expert prédéfinissant ces paramètres.

Deuxième partie

Pratique

Chapitre 1

Expérimentation du solveur CSP autonome

Sommaire

1.1 Introduction

Dans la partie expérimentale de ce stage, nous nous limitons à l'implantation de simples bandits stochastiques au sein du solveur CSP AbsCon (https ://www.cril.univ-artois.fr/ lecoutre/#/softwares). Notre approche consiste à appeler l'algorithme des bandits à l'endroit où l'on s'occupe des redémarrages. Il s'agit d'un emplacement stratégique assurant des appels réguliers au bandit.

Toujours dans le but de bien délimiter notre périmètre d'expérimentation pour cette première approche, nous limitons le changement de la configuration du solveur à l'adaptation de l'heuristique de choix de variables et au choix du tableau de variables apparaissant en priorité à la base de l'arbre de recherche.

1.2 Architecture

1.2.1 Conceptualisation

Nous pouvons considérer le problème du choix de configuration d'un solveur CSP comme étant semblable au problème des Bandits Multi-Bras. En effet, les deux notions s'associent de cette manière :

— on définit K actions $a_1, ..., a_K$ où chacune d'elles représente une configuration possible du solveur ;

— on appelle T fois le bandit (choix de l'action et mise à jour) correspondant aux T redémarrages du solveur ;
— on cherche à minimiser un regret sur le choix de la configuration, ce qui se ramène à apprendre la configuration qui minimise le temps de recherche (ou la quantité de nœuds visités).

1.2.2 Implantation

Pour cette étude, nous disposons du solveur de contraintes AbsCon. Il est très simplement paramétrable et dispose de différents algorithmes implantant les heuristiques nécessaires à notre étude.

Nous choisissons d'implanter deux algorithmes de bandits stochastiques au sein de ce solveur : *epsilon-greedy* et *ucb*. Pour cette première approche, nous choisissons de placer le bandit à l'endroit où se trouve le mécanisme de redémarrage. Ainsi, à chaque redémarrage, le solveur récompense le bandit et lui demande la nouvelle configuration à utiliser par la suite :

1. (initialisation) on initialise la structure de l'algorithme du bandit à l'exécution du solveur (chaque action possède un score égal à 0) ;
2. (choix de l'action) après chaque redémarrage (première exécution comprise), le bandit choisit une configuration (une action) et associe celle-ci au solveur jusqu'au prochain redémarrage ;
3. (mise à jour) après chaque exécution (avant de reprendre la recherche), nous mettons à jour la moyenne des récompenses de la configuration précédemment associée au solveur.

1.2.3 Choix d'une fonction de récompense

Une question primordiale concerne le choix de la fonction de récompense. Le but étant d'accélérer au mieux la recherche, plusieurs caractères du solveur sont désignés comme étant de bon indicateur à prendre en compte pour récompenser le bandit. Nous pouvons comptabiliser :
— le nombre de mauvaises décisions (noté md_t où $1 \leq t \leq T$ correspond à la t^{e} exécution) ;
— les différentes tailles de *nogoods* produits (noté $ng_{i,t}$ où $2 \leq i \leq 10$ correspond aux nombre de *nogoods* de taille i produits à la t^{e} exécution).

Fonction de récompense basée sur les mauvaises décisions

Cette fonction de récompense candidate prend en compte les mauvaises décisions prises durant la recherche. Une mauvaise décision correspond au choix de la valeur d'une variable ayant mené à une incohérence. Plus le nombre de mauvaises décisions est faible, plus l'exécution courante est considérée comme correcte. Ainsi, nous pouvons associer à la fonction de récompense $r_t(a_t)$ la récompense $-mg_t$ pour l'action a_t.

Fonction de récompense basée sur les *nogoods*

L'autre fonction candidate à étudier prend en compte l'usage des *nogoods* pour la fonction de récompense. Il est à noter que, plus le *nogood* produit est petit, plus grande est son importance. Ainsi, nous pouvons associer à la fonction de récompense $r_t(a_t)$ la somme :

$$\Sigma_{i=2}^{10} \frac{ng_{i,t}}{2^{i-2}}$$

pour l'action a_t. Cette somme donne un poids deux fois plus important aux *nogoods* de taille i qu'à ceux de taille $i+1$.

1.3 Expérimentation

1.3.1 Architecture

Notre but est de maintenant vérifier l'efficacité du solveur dont la configuration est dynamisée par l'usage d'un bandit stochastique (ucb et epsilon-greedy). En plus de l'usage d'un bandit, nous testons nos deux propositions de fonction de récompense. Pour ces expérimentations, nous faisons aussi varier la façon dont nous rendons le solveur autonome :

— seule l'heuristique du choix de variables est adaptative (cercle représenté par H dans la Figure 1.1) ;
— seul le tableau de variables à placer en début de recherche est adaptatif (cercle représenté par T dans la Figure 1.1) ;
— le produit cartésien des deux précédentes caractéristiques formant une configuration combinatoire est adaptatif (représenté comme l'intersection des cercles H et T dans la figure 1.1).

La dernier cercle représente n'importe quel autre paramètre d'AbsCon (heuristique de choix de valeurs, fonction de propagation, etc.). Comme dit précédemment, nous nous concentrons uniquement sur les heuristiques de choix de variables, les tableaux de variables et la combinaison des deux.

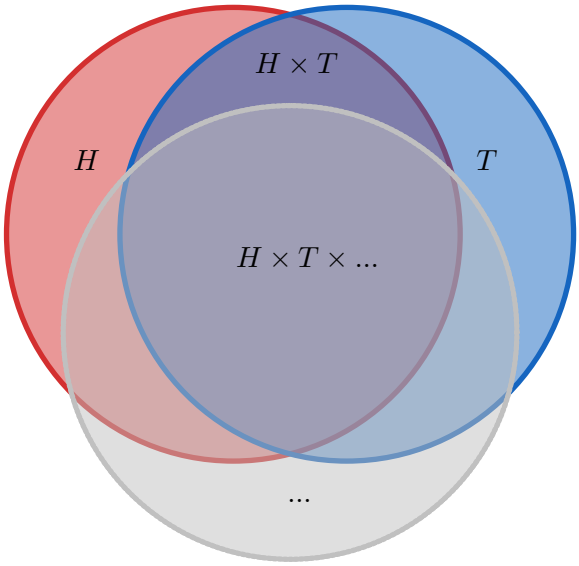


FIGURE 1.1 – Représentation des configurations du solveur AbsCon

En plus de ces expérimentations, nous testons aussi le solveur avec les différentes heuristiques classiques sans bandit. Et enfin, un bandit choisissant ses bras de façon aléatoire est aussi expérimenté.

La fonction de propagation est fixe : il s'agit de AC.

Deux campagnes sont menées. La première d'entre elles teste l'ensemble des solutions précédemment citées avec six problèmes : *CarSequencing*, *Crossword*, *Eternity*, *MisteryShopper*, *QuasiGroup* et *SportsScheduling* totalisant 81 instances. La seconde campagne totalise 366 instances de 25 problèmes différents. Cette dernière est moins exhaustive en terme de diversité de configurations testées, c'est pourquoi nous nous basons essentiellement sur la première campagne pour cette partie de l'étude.

Les instances sont exécutées sur le cluster du CRIL. Les noeuds utilisés disposent d'un processeur Intel i7-4770 cadencé à 3.40GHz et d'une capacité de 8GB de RAM. Le temps maximum d'exécution est fixé à 1200 secondes.

1.3.2 Résultats

Configuration				temps moyen cpu (18 instances communes)	temps moyen cpu (#instances résolues)
Statique			activity	5.70	102.27 (33)
			ddeg	81.75	118.14 (30)
			impact	6.13	117.30 (32)
			mem	109.74	190.56 (24)
			wdeg	15.20	138.77 (41)
Adaptative	Epsilon Greedy	nogoods	arrays	5.50	52.56 (36)
			varh	6.78	87.14 (36)
			varh-arrays	4.72	33.27 (37)
		wrong	arrays	5.36	27.44 (36)
			varh	6.93	12.44 (31)
			varh-arrays	4.44	47.70 (37)
	UCB	nogoods	arrays	4.99	34.89 (37)
			varh	9.79	98.57 (35)
			varh-arrays	4.96	32.60 (37)
		wrong	arrays	6.34	59.70 (38)
			varh	8.69	123.85 (38)
			varh-arrays	5.11	40.66 (37)
	Rand-arm		arrays	5.27	31.58 (37)
			varh	8.51	23.17 (34)
			varh-arrays	5.42	28.55 (37)

TABLE 1.1 – Résultat de l'expérimentation du solveur autonome

La Table 1.1 récapitule les résultats obtenus lors des expérimentations. Le tableau est séparé en deux parties :

— la partie supérieure affiche les résultats des cinq heuristiques statiques ;
— la partie inférieure compare les différents bandits implantés.

Les deux premiers bandits nécessitent une fonction de récompense :

— *nogoods* représente la fonction de récompense basée sur le scorage des différents *nogoods* ;
— *wrong* représente la fonction de récompense basée sur la somme des mauvaises décisions prises lors d'une recherche.

Rand-arm correspond au bandit choisissant de façon aléatoire ses bras. La dernière colonne, celle représentant les paramètres des bandits, correspond à la configuration que nous souhaitons rendre autonome dans le solveur :

— *arrays* correspond aux différents tableaux composant un problème que nous souhaitons placer en priorité dans la recherche ;
— *varh* correspond aux différentes heuristiques disponibles.

Ainsi, nous pouvons choisir de dynamiser le solveur avec chacune de ces caractéristiques à l'unité, ou alors de les combiner : *varh-arrays*.

1.3.3 Interprétation des résultats

Les résultats sont mitigés.

Du côté des points positifs, nous pouvons observer que les instances communes (c'est-à-dire, résolues communément entre toutes les exécutions d'Abscon) ont tendance à conclure leur recherche plus rapidement dans le cas des configurations adaptatives plutôt que dans le cas statique. Autre point important, nous constatons que les configurations combinatoires des bandits ont tendance à produire de meilleures résultats que dans le cas où seul le choix d'heuristique (resp. le choix de tableau de variables) est adaptatif.

D'un autre côté, nous remarquons le manque de robustesse des bandits face à l'heuristique statique *wdeg*. En effet, celle-ci résout à elle-même 41 des 81 instances proposées alors que les bandits tournent autour de 36 à 38 résolutions. Un autre point négatif concerne le fait que le bandit aléatoire est pratiquement aussi efficace que les bandits stochastiques.

1.4 Conclusion

Cette expérimentation a mis en évidence le manque de robustesse du solveur implantant les bandits stochastiques. De plus, les résultats de ceux-ci sont à peine meilleurs que la version aléatoire du bandit. Malgré ce manque, nous en concluons tout de même que rendre combinatoire la configuration adaptative du solveur (tableau de variables et heuristique de choix de variable combinés) peut le rendre plus performant.

Il nous faut maintenant trouver un moyen de comprendre l'inefficacité actuelle de l'implantation des bandits.

Chapitre 2

Étude complémentaire

Sommaire

2.1 Introduction

Au vu de la précédente expérimentation montrant la défaillance du modèle de solveur CSP autonome, il peut être intéressant d'étudier les différents facteurs pouvant remettre en cause le bon fonctionnement du bandit implanté.

2.2 Étude de corrélation

Pour qu'un bandit soit efficace, il est primordial de choisir avec soin sa fonction de récompense. Si celle-ci n'est pas représentative d'une bonne exécution du solveur, alors cette récompense peut biaiser le bandit, voir même le rendre moins efficace qu'un bandit aléatoire.

C'est pourquoi, une étude de corrélation entre les caractères du solveur composant nos fonctions de récompense et la bonne exécution de celui-ci semble être une bonne piste pour découvrir un certain dysfonctionnement.

2.2.1 Protocole

Grâce à la deuxième campagne, nous sommes en possession d'une base de données des différentes exécutions d'AbsCon. A travers elle, nous essayons de mettre en évidence, l'existence ou non, de corrélations entre les caractères comprenant les mauvaises décisions, les *nogoods* et le temps cpu d'une instance. Nous utilisons uniquement les exécutions avec les heuristiques classiques.

Dans l'ensemble des résultats, nous ne gardons que les instances dont la recherche a abouti. Nous disposons aussi de l'ensemble des statistiques de redémarrage comprenant : les mauvaises décisions et les *nogoods*. Nous calculons la moyenne des dix premiers redémarrages de chaque instance (la moyenne des mauvaises décisions et des *nogoods*). Ainsi, nous procédons à l'étude de corrélation entre ces différents caractères et le temps cpu nécessaire à la résolution du problème.

2.2.2 Choix de la mesure de corrélation

Il existe différentes manières de mesurer une corrélation. Nous nous concentrons sur deux d'entre elles : les mesures de Pearson et Spearman [ZOU *et al.* 2003]. La mesure de Pearson évalue la relation linéaire entre deux variables (la variation de l'une est proportionnelle à la variation de l'autre). La mesure de Spearman évalue la relation monotone entre deux variables. Cette mesure prend en compte une relation qui n'est pas forcément linéaire.

L'analyse préalable permettant de choisir l'une de ces mesures consiste à tracer les nuages des points correspondant aux données à étudier.

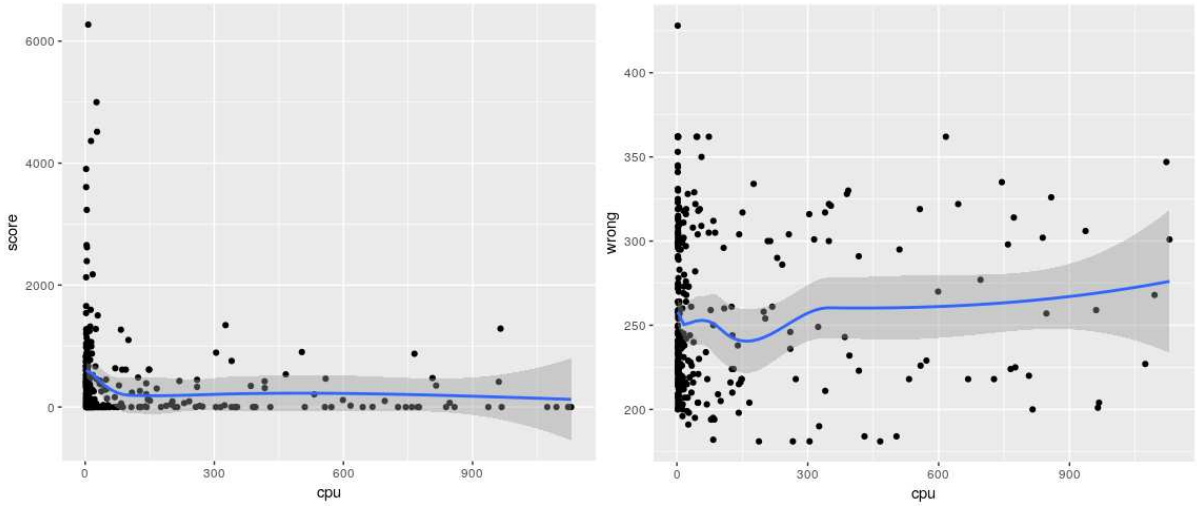


FIGURE 2.1 – Nuages de points comparant le cpu avec le score des nogoods et les mauvaises décisions

Avant même de passer à l'étape de l'analyse de corrélations, nous observons une irrégularité entre les deux caractéristiques. Nous pouvons tout de même remarquer une légère tendance croissante et décroissante (non-linéaire) pour chacune d'elle. Cette tendance nous impose de choisir la mesure de Spearman pour évaluer la force de corrélation unissant ces caractères.

2.2.3 Résultats

Après avoir appliqué l'algorithme de Spearman sur les données extraites d'AbsCon, le résultat de l'étude nous est donné sous forme d'un corrélogramme (Annexe A). L'intersection entre deux caractères correspond à leur corrélation. Plus la couleur (bleue ou rouge) de celui-ci est intense, plus la corrélation est forte. La couleur rouge (resp. bleue) montre une corrélation négative (resp. positive). Un coefficient de corrélation est compris entre -1 et 1.

Nous constatons que les critères choisis comme fonction de récompense pour les bandits stochastiques semblent ne pas être fortement corrélés au temps cpu (Table 2.1).

	#mauvaises décisions	Score des nogoods	#filtres appliqués
Temps cpu	-0.06	$-0,41$	$0,53$

TABLE 2.1 – Corrélations entre les différentes caractéristiques d'un solveur CSP

Nous pouvons aussi constater que le critère correspondant au nombre de filtres semble être le critère le plus corrélé avec le temps cpu. Cependant, lors de la première campagne, nous avions introduit cette caractéristique dans la fonction de récompense mais n'avions pas eu de meilleurs retours.

Nous constatons que les critères initialement choisis pour la fonction de récompense ne sont pas assez représentatifs d'une exécution réussie du solveur. Bien que le critère correspondant aux filtres soit plus corrélé, celui-ci n'induit pas une plus grande robustesse au solveur. C'est pourquoi, nous en concluons que la fonction de récompense n'est pas la seule fautive concernant le problème d'implantation.

2.3 Les bandits mis au banc d'essai

Un autre point semble important concernant l'efficacité d'un bandit : combien de récompenses faut-il lui attribuer pour que celui-ci converge vers l'action provoquant un regret négligeable ? Sachant que, pour Abscon, 20 minutes d'exécution limite le bandit à une centaine d'appels en moyenne au niveau du point de redémarrage.

2.3.1 Protocole

Afin de juger de l'efficacité de nos bandits stochastiques, nous les testons avec des bras dont la récompense est prédéfinie. Dans ces tests, chaque bandit possède six bras dont les probabilités de récompense sont de : 0.02, 0.015, 0.01, 0.015, 0.2 et 0.01. Ainsi, l'avant dernier bras possède une chance sur cinq d'obtenir une récompense : 1. Dans les autre cas, la récompense est égale à 0.

2.3.2 Résultats

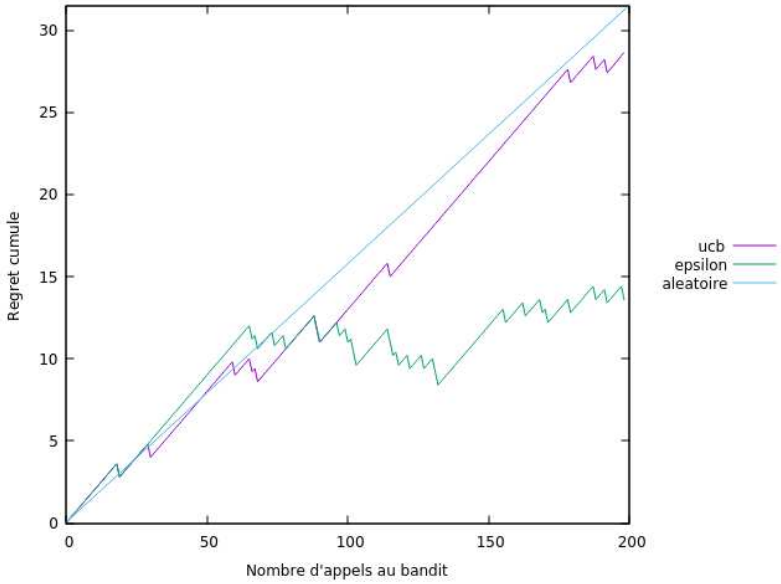


FIGURE 2.2 – Regret cumulé des bandits (ucb, epsilon et aléatoire) sur 200 itérations

La Figure 2.2 montre le regret cumulé des deux bandits stochastiques (ucb et epsilon-greedy). Une troisième courbe représente le regret cumulé d'un bandit aléatoire. Nous pouvons remarquer que pour 200 itérations, ucb reste très semblable au bandit aléatoire. Quant à epsilon-greedy, il n'est que légèrement meilleur à partir d'une centaine d'itérations.

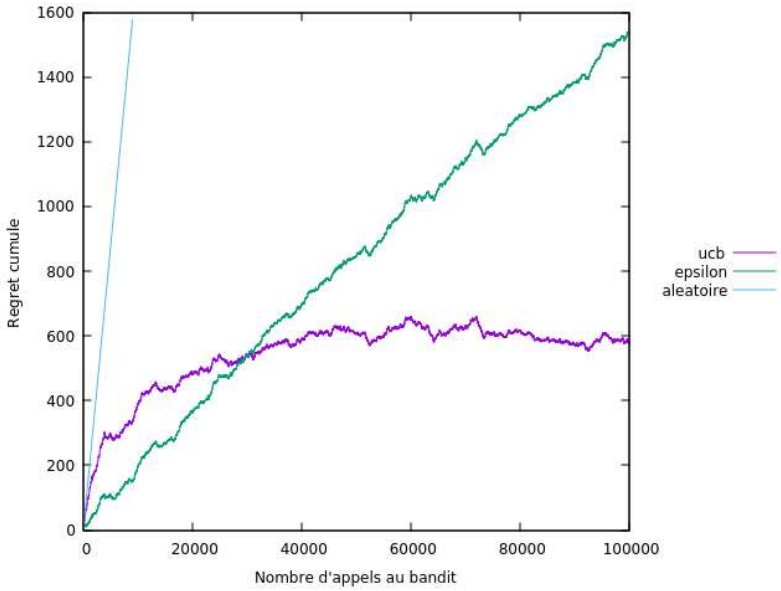


FIGURE 2.3 – Regret cumulé des bandits (ucb, epsilon et aléatoire) sur 100000 itérations

La Figure 2.3 représente le premier exemple avec 100000 itérations. Nous remarquons qu'une marge se crée entre le bandit aléatoire et les deux autres. De plus, nous pouvons remarquer qu'au bout de 30000

itérations ucb devient meilleur qu'epsilon-greedy avec un regret cumulé devenant constant (le bandit a convergé vers le bras optimal). Le comportement d'epsilon-greedy n'est pas étonnant car, tout au long de sa recherche, il conserve toujours la même proportion entre exploration et exploitation.

2.4 Conclusion

À travers ces études complémentaires, nous en concluons que les résultats de la précédente étude sur le solveur CSP autonome ne sont pas si étonnants. En effet, nous remarquons que la déficience des implantations est autant due à une fonction de récompense ne remplissant pas entièrement son rôle, qu'à un nombre d'appels aux bandits bien insuffisant.

Conclusion générale

Dans ce mémoire, nous nous sommes appliqués à expliquer l'enjeu de rendre les solveurs de contraintes CSP autonomes. Les utilisateurs de ce genre de solveur n'étant pas forcément experts, il leur est compliqué de configurer convenablement ce dernier.

Ce problème s'apparente totalement à celui des Bandits Multi-Bras. En effet, pour l'utilisateur classique, il s'agit de trouver la configuration la plus optimale afin de rechercher efficacement la réponse à son problème. Ainsi, l'utilisateur fait face à un dilemme entre l'exploration de chaque configuration et l'exploitation de celle qui lui permet d'obtenir rapidement une solution.

A travers nos premières implantations des Bandits Multi-Bras, nous nous sommes heurtés à deux problèmes majeurs. L'un d'eux correspond au choix d'une fonction de récompense qui n'a pas permis de donner assez d'information au bandit afin qu'il s'oriente vers la bonne configuration. L'autre problème correspond à un nombre d'observations insuffisants pour mener à bien l'apprentissage.

Malgré ces problèmes, nous avons pu remarquer que le bandit avait tendance à rendre la résolution plus rapide quand celui-ci avait accès à plus d'un paramètre du solveur. Cette observation s'avère rassurante dans le sens où, rendre le solveur plus autonome (en lui donnant accès à plus de paramètres), le rend aussi plus performant.

A l'issue de ce stage, plusieurs perspectives s'offrent à nous.

L'une consiste à augmenter considérablement le nombre d'observations avec une fréquence de redémarrage plus importante (nécessite le changement de la politique par défaut d'AbsCon). Avec ceci, il serait intéressant de tester de nouveaux algorithmes répondant au précédent problème : Thompson Sampling [THOMPSON 1933] et Exp3 [SELDIN & LUGOSI 2017]. Ce dernier, Exp3, est un bandit concurrentiel. Grâce à celui-ci, nous pourrons partir du principe que, dans un même problème, la configuration optimale est changeante. La configuration pourra ainsi d'adapter durant la résolution.

Une autre perspective consiste à placer le bandit à différents niveaux de l'arbre de recherche. Cette approche nous permettra d'obtenir un nombre exponentiel d'observations et ainsi un meilleur apprentissage de l'agent.

Enfin, nous pensons aussi appliquer l'usage du bandit sur les problèmes d'optimisation. Ceux-ci nous donnent accès à une information plus précise sur la qualité de la résolution, nous permettant d'ajuster au mieux la fonction de récompense du bandit.

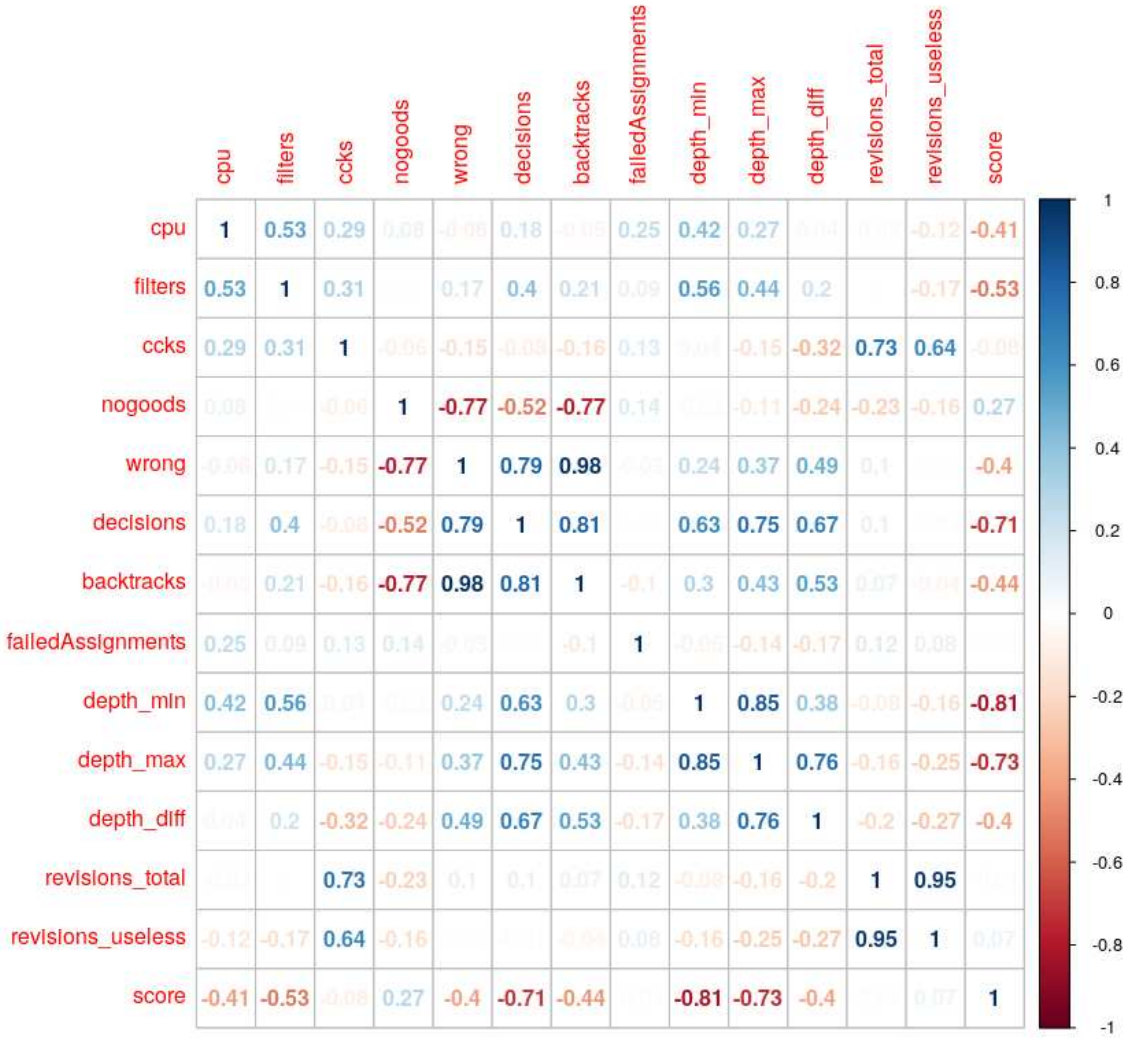
Ce stage m'a beaucoup appris sur la méthodologie de recherche. J'ai été captivé par l'intensité de l'apprentissage au début de l'étude mais aussi par les applications qui en résultaient, avec en plus, cette sensation d'avoir apporté mon humble pierre à l'édifice.

Cette immersion dans le monde de la recherche a définitivement confirmé mon attrait pour celle-ci.

Annexes

Annexe A

Corrélogramme des caractéristiques d'AbsCon évalué par la mesure de Spearman



Bibliographie

[AUER *et al.* 2002] Peter AUER, Nicolò CESA-BIANCHI, et Paul FISCHER. « Finite-time Analysis of the Multiarmed Bandit Problem ». *Machine Learning*, 47(2) :235–256, May 2002.

[BALAFREJ *et al.* 2015] Amine BALAFREJ, Christian BESSIÈRE, et Anastasia PAPARRIZOU. « Multi-Armed Bandits for Adaptive Constraint Propagation ». Dans *The Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015.*, Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015., Buenos Aires, Argentina, juillet 2015.

[BESSIÈRE & RÉGIN 1996] Christian BESSIÈRE et Jean-Charles RÉGIN. « MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems ». Dans Eugene C. FREUDER, éditeur, *Principles and Practice of Constraint Programming — CP96*, pages 61–75, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[BESSIÈRE *et al.* 2005] Christian BESSIÈRE, Jean-Charles RÉGIN, Roland H.C. YAP, et Yuanlin ZHANG. « An optimal coarse-grained arc consistency algorithm ». *Artificial Intelligence*, 165(2) :165 – 185, 2005.

[BOUSSEMART *et al.* 2004] Frédéric BOUSSEMART, Fred HEMERY, Christophe LECOUTRE, et Lakhdar SAIS. « Boosting Systematic Search by Weighting Constraints ». Dans *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI'04, pages 146–150, Amsterdam, The Netherlands, The Netherlands, 2004. IOS Press.

[CORMEN *et al.* 2009] Thomas H. CORMEN, Charles E. LEISERSON, Ronald L. RIVEST, et Clifford STEIN. « *Introduction to Algorithms, Third Edition* », pages 1048–1105. The MIT Press, 3rd édition, 2009.

[GITTINS 1989] J.C. GITTINS. *Multi-armed bandit allocation indices*. Wiley-Interscience series in systems and optimization. Wiley, 1989.

[HARALICK & ELLIOTT 1980] Robert M. HARALICK et Gordon L. ELLIOTT. « Increasing tree search efficiency for constraint satisfaction problems ». *Artificial Intelligence*, 14(3) :263 – 313, 1980.

[LECOUTRE *et al.* 2009] Christophe LECOUTRE, Lakhdar SAÏS, Sébastien TABARY, et Vincent VIDAL. « Reasoning from last conflict(s) in constraint programming ». *Artificial Intelligence*, 173(18) :1592 – 1614, 2009.

[MACKWORTH 1977] Alan MACKWORTH. « Consistency in Networks of Relations ». 8 :99–118, 02 1977.

[MICHEL & VAN HENTENRYCK 2012] Laurent MICHEL et Pascal VAN HENTENRYCK. « Activity-Based Search for Black-Box Constraint Programming Solvers ». Dans Nicolas BELDICEANU, Narendra JUSSIEN, et Éric PINSON, éditeurs, *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, pages 228–243, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[REFALO 2004] Philippe REFALO. « Impact-Based Search Strategies for Constraint Programming ». Dans Mark WALLACE, éditeur, *Principles and Practice of Constraint Programming – CP 2004*, pages 557–571, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[SABIN & FREUDER 1994] Daniel SABIN et Eugene C. FREUDER. « Contradicting conventional wisdom in constraint satisfaction ». Dans Alan BORNING, éditeur, *Principles and Practice of Constraint Programming*, pages 10–20, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[SELDIN & LUGOSI 2017] Yevgeny SELDIN et Gábor LUGOSI. « An Improved Parametrization and Analysis of the EXP3++ Algorithm for Stochastic and Adversarial Bandits ». *CoRR*, abs/1702.06103, 2017.

[SUTTON & G. BARTO 1998] Richard SUTTON et Andrew G. BARTO. « Reinforcement Learning : An Introduction ». 9 :1054, 02 1998.

[THOMPSON 1933] W. R. THOMPSON. « On the Likelihood that one Unknown Probability Exceeds Another in View of the Evidence of Two Samples ». *Biometrika*, 25 :285–294, 1933.

[WILSON & NASH 2003] Robin WILSON et Charles NASH. « Four colours suffice : how the map problem was solved ». 25 :80–83, 12 2003.

[XIA & YAP 2018] Wei XIA et Roland H. C. YAP. « Learning Robust Search Strategies Using a Bandit-Based Approach ». Dans *AAAI*, 2018.

[ZOU *et al.* 2003] Kelly H. ZOU, Kemal TUNCALI, et Stuart G. SILVERMAN. « Correlation and Simple Linear Regression ». *Radiology*, 227(3) :617–628, 2003. PMID : 12773666.