Tirage de solutions par ajout de contraintes tables aléatoires

Mathieu Vavrille, Charlotte Truchet, Charles Prud'Homme

Tirage de solutions par ajout de contraintes tables aléatoires

Mathieu Vavrille[†*] **Charlotte Truchet**[†] **Charles Prud'homme**[‡]

[†]Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes
[‡]TASC, IMT-Atlantique, LS2N-CNRS, F-44307 Nantes
[†]prenom.nom@univ-nantes.fr [‡]prenom.nom@imt-atlantique.fr

Résumé

Les solveurs de contraintes actuels mettent à disposition des utilisateurs des algorithmes efficaces pour traiter les problèmes de satisfaction et d'optimisation combinatoires. Ceux-ci sont inadaptées à de nouveaux usages, comme celui du tirage aléatoire de solutions. Nous proposons ici un algorithme pour tirer des solutions aléatoirement, se basant sur l'ajout de contraintes tables générées aléatoirement, sans modifier le modèle du problème. Nous avons implémenté cette méthode de résolution en utilisant un solveur de contraintes existant. Nos expériences montrent que cet algorithme est une amélioration par rapport à une stratégie de branchement aléatoire en terme de qualité de l'aléatoire du tirage.

Abstract

Constraint solvers feature efficient algorithms to handle satisfaction and optimisation combinatorial problems. These features are not suited to new usages, such as the sampling of solutions. We propose here an algorithm to randomly sample solutions, based on the addition of randomly generated table constraints, without modifying the model of the problem. We implemented this method of resolution using an existing constraint solver. Our experiments show that this algorithm is an improvement over a random branching strategy in terms of quality of the randomness of the sampling.

1 Introduction

Les solveurs de contraintes s'améliorent constamment en terme d'expressivité, de facilité d'utilisation et de performances, ce qui les ouvre à de nouvelles utilisations. Classiquement les solveurs permettent de trouver une, ou la meilleure solution pour un problème. Parmi les applications des solveurs de contraintes il y a par exemple les problèmes de planification ou de configuration. Or, pour ces applications, il est fréquent que l'utilisateur souhaite avoir plusieurs solutions significativement différentes pour faciliter l'aide à la décision. Il peut également exiger une forme d'équité entre les solutions, si les décisions prises à partir de ces solutions ont des impacts humains. On se trouve donc face à deux nouvelles problématiques : renvoyer plusieurs solutions variées, ou choisir équitablement une solution (tirage aléatoire uniforme). Les mécanismes internes des solveurs de contraintes ne peuvent pas garantir ces propriétés car la recherche en profondeur renvoie séquentiellement des solutions proches.

Dans cet article, nous nous intéressons au problème de tirer aléatoirement des solutions, uniformément parmi l'ensemble des solutions. En effet, des solutions renvoyées aléatoirement permettent d'avoir une certaine diversité, car il est peu probable de toujours avoir des solutions proches, et garantit une certaine équité car la recherche n'est pas biaisée par la modélisation du problème. Il est toujours possible de choisir aléatoirement variables et valeurs avec la stratégie de recherche RandomVarDom mais les solutions sont loin d'être renvoyées uniformément parmi toutes les solutions.

1.1 Contributions

Cet article présente une nouvelle approche permettant de tirer aléatoirement une solution, utilisant un solveur de contraintes comme une boîte noire, et sans ré-écrire le modèle du problème.

Nous proposons un algorithme simple reposant sur l'ajout au problème de contraintes tables générées aléatoirement. Nous avons implémenté cet algorithme en utilisant le solveur Choco-solver[12] et nous le comparons à RandomVarDom sur des plusieurs types de

*Papier doctorant : Mathieu Vavrille[†] est auteur principal.

problèmes. Les expériences montrent que l'approche tire *plus uniformément* que la stratégie RandomVarDom, et atteint l'uniformité sur certain problèmes. Bien que, pour garantir ces propriétés supplémentaires, on s'attende à une dégradation du temps de calcul, nous constatons que ce dernier reste dans un ordre de grandeur comparable.

1.2 Travaux liés

L'approche utilisée dans cet article est fortement inspirée des travaux de Kuldeep Singh Meel [10] en SAT, où les contraintes ajoutées sont des contraintes XOR générées aléatoirement. L'algorithme d'ajout de contraintes, associé à ce choix spécifique de contraintes XOR, permet de contrôler la proximité entre l'échantillonnage effectué, et un tirage uniforme. En comparaison, nous avons fait le choix de contraintes tables, efficaces et déjà existantes dans les solveurs CP, et d'un algorithme de tirage plus économique. En contrepartie, notre méthode ne donne pas de garanties sur l'uniformité du tirage, et nous l'étudions donc expérimentalement.

Le tirage aléatoire de solutions en programmation par contraintes a été étudié d'abord dans [4], puis [6] en utilisant des réseaux bayésiens. Ces approches permettent d'avoir un tirage uniforme (ou de choisir la distribution des solutions), mais ont l'inconvénient majeur d'être exponentielles en la largeur induite du graphe de contraintes, ce qui rend impossible l'utilisation sur de grosses instances, et force l'utilisation d'approximations. D'autres approches, dans [7, 15], garantissent la diversité des solutions, qui est une tâche différente de l'échantillonnage car elle consiste à trouver des solutions éloignées les unes des autres (pour une métrique adaptée). Ces approches utilisent une réécriture du problème, ou une heuristique de recherche visant à explorer les espaces de solutions éloignés des solutions déjà trouvées.

1.3 Organisation de l'article

L'article est organisé de la manière suivante : la section 2 présente les pré-requis théoriques nécessaires à la compréhension de l'article, notamment le test du χ^2 et la stratégie RandomVarDom ; la section 3 présente la nouvelle approche de tirage par tables ; pour finir, la section 4 présente les résultats expérimentaux pour évaluer la qualité de l'aléatoire ainsi que le temps de calcul.

2 Pré-requis théoriques

2.1 Programmation par contraintes

Dans cet article nous sommes intéressés par les problèmes de satisfaction de contraintes. Un problème de satisfaction de contraintes (CSP) $\mathcal{P}$ est un triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ où

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ est un ensemble de variables,
- $\mathcal{D}$ est une fonction associant un domaine à chaque variable,
- $\mathcal{C}$ est un ensemble de contraintes, chaque contrainte $C \in \mathcal{C}$ est constituée :
  - d'un tuple de variables appelé *scope* de la contrainte $scp(C) = (X_{i_1}, \ldots, X_{i_r})$, où r est l'arité de la contrainte
  - d'une relation, i.e. un ensemble d'instanciations

$$rel(C) \subseteq \prod_{k=1}^{r} \mathcal{D}(X_{i_k})$$

Une contrainte est dite satisfaite si chaque variable $X_{i_k} \in scp(C)$ est instanciée à une valeur de son domaine $x_{i_k} \in \mathcal{D}(X_{i_k})$ et que $(x_{i_1}, \ldots, x_{i_r}) \in rel(C)$. Les contraintes peuvent être données en extension (appelées contraintes de table [5]) en donnant explicitement $rel(C)$, ou en intention par une expression dans un langage de plus haut niveau. Par exemple, l'expression $x + y \leq 1$ pour des domaines $\{0, 1\}$ représente $rel(C) = \{(0, 0), (0, 1), (1, 0)\}$.

La résolution d'un CSP est la recherche d'une, une partie ou toutes les solutions, c'est à dire l'affectation d'une valeur à chaque variable, telle que toutes les contraintes sont satisfaites. Les problèmes d'optimisation (COP) sont des CSP auxquels ont été ajoutés une fonction objectif *obj* à minimiser (ou maximiser).

Notation : Soit un problème $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, et une contrainte C, par abus de notation nous notons $\mathcal{P} \wedge C$ le CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{C\} \rangle$. Nous notons $Sols(\mathcal{P})$ l'ensemble des solutions du problème $\mathcal{P}$.

Par la suite, nous ne considérerons que des problèmes de satisfaction, mais les problèmes d'optimisations pourront être traités grâce à la transformation suivante. Soit un COP $(\mathcal{P}, obj)$ à minimiser (resp. maximiser), et soit *opt* la valeur minimum (resp. maximum) de l'objectif. Soit $\epsilon \geq 0$, on transforme alors ce problème en un CSP $\mathcal{P} \wedge (obj \leq opt + \epsilon)$ (resp. $\mathcal{P} \wedge (obj \geq opt - \epsilon)$). Plus l'écart ϵ est élevé, plus les solutions cherchées seront éloignées de la valeur optimale.

2.2 Test du chi-deux

Évaluer le caractère aléatoire d'un système est une tâche compliquée, du fait que les processus aléatoires

peuvent prendre des valeurs étonnantes sans pour autant être biaisés (par exemple, une pièce tombant 10 fois de suite sur pile). Le test du chi-deux (ou χ^2) permet de confronter un résultat d'expérience aléatoire a une distribution de probabilité attendue. Il provient d'un résultat de convergence d'une loi vers la loi du χ^2 énoncé dans [11] et rappelé ici. Soit Y une variable aléatoire prenant la valeur k avec probabilité p_k pour $1 \leq k \leq d$. Soient $Y_1, \ldots, Y_n$ des variables aléatoires indépendantes de même loi que Y. Soit $N_n^{(k)}$ le nombre de variables $Y_i, 1 \leq i \leq n$ égales à k.

Théorème 1 ([11]). *Quand n tend vers l'infini, la fonction de répartition de la variable aléatoire*

$$Z_n = \sum_{k=1}^{d} \frac{\left(N_n^{(k)} - n \cdot p_k\right)^2}{n \cdot p_k}$$

tend vers la fonction de répartition de la loi du χ^2 à $(d-1)$ degrés de liberté (notée χ^2_{d-1}).

Le test du χ^2 revient à tirer des valeurs en faisant l'hypothèse qu'elles suivent la loi de Y, de calculer la valeur expérimentale z_n^{exp}, puis calculer la probabilité (appelée valeur-p)

$$\mathbb{P}(Z_n \geq z_n^{exp}) \approx \mathbb{P}(\chi^2_{d-1} \geq z_n^{exp})$$

Si cette probabilité est faible, cela signifie qu'il est improbable d'avoir un résultat plus extrême que celui obtenu, ce qui signifie que l'hypothèse selon laquelle les valeurs tirées suivent la même loi que Y peut être rejetée avec confiance.

2.3 Stratégie de recherche aléatoire

L'algorithme de recherche de solution d'un problème de contraintes alterne :

- une recherche en profondeur où l'espace de recherche est réduit en ajoutant une contrainte (appelée décision), par exemple, une affectation d'une variable à une valeur de son domaine ;
- une phase de propagation qui vérifie la satisfiabilité des contraintes du CSP.

Une stratégie de recherche naturelle pour avoir une recherche aléatoire de solutions est la stratégie RandomVarDom, qui consiste à choisir aléatoirement (uniformément) une variable X parmi toutes les variables non instanciées, puis une valeur $x \in \mathcal{D}(X)$, et appliquer la décision $X = x$. Cette stratégie a l'avantage d'être très facilement implémentée dans n'importe quel solveur de contraintes. En revanche elle empêche d'utiliser une autre stratégie d'exploration plus efficace, et elle n'offre pas de garanties sur la répartition des solutions tirées, notamment la répartition des solutions peut ne pas être uniforme. Par exemple sur le problème $\mathcal{P} = \langle\{X, X\}, \{X \to \{0,1\}, Y \to \{0,1\}\}, \{X + Y > 0\}\rangle$, soit s la solution renvoyée par un solveur configuré pour faire une recherche aléatoire, alors,

$$\mathbb{P}(s = \{X \mapsto 0, Y \mapsto 1\}) = \frac{3}{8}$$
$$\mathbb{P}(s = \{X \mapsto 1, Y \mapsto 0\}) = \frac{3}{8}$$
$$\mathbb{P}(s = \{X \mapsto 1, Y \mapsto 1\}) = \frac{1}{4}$$

Les implications sur le temps de calcul pour trouver des solutions sur différents problèmes sont évoquées dans la section 4.

```
1 Fonction TableAleatoire(P, v, p)
      Données : Un CSP
                P = ⟨{X_1, ..., X_n}, D, C⟩, v > 0,
                0 < p < 1
      Résultat : Une contrainte table aléatoire
2     T ← {};
3     i_1, ..., i_v ← TireIndices(P, v);
4     pour chaque (x_{i_1}, ..., x_{i_v}) ∈ ∏_{k=1}^{v} D(X_{i_k})
       faire
5         si Random() < p alors
6             T.add((x_{i_1}, ..., x_{i_v}));
7     retourner Table([X_{i_1}, ..., X_{i_v}], T);
```

Algorithme 1 : Algorithme de tirage de contrainte table aléatoire

3 Nouvelle approche d'échantillonage

Nous présentons ici une nouvelle approche pour tirer des solutions. Cette approche se base sur l'ajout de contraintes tables générées aléatoirement pour réduire le nombre de solutions du problème, solutions qui seront alors énumérées.

3.1 Contraintes tables aléatoires

L'algorithme de tirage aléatoire de tables est présenté dans l'algorithme 1. On suppose accessibles des fonctions Random() qui renvoie un nombre flottant aléatoire entre 0 et 1, TireIndices$(\mathcal{P}, v)$ qui tire v indices $i_1, \ldots, i_v$ tels que $|\mathcal{D}(X_{i_k})| \neq 1, 1 \leq k \leq v$ et Table(X, T) qui crée une contrainte table C telle que $scp(C) = X$ et $rel(C) = T$. Les deux paramètres de l'algorithme sont : v le nombre de variables dans la table, et p la probabilité d'ajouter un tuple dans la table. L'algorithme choisit aléatoirement v variables parmi les variables dont le domaine n'est pas un singleton, puis parcourt toutes les instanciations de ces v

variables, en ajoutant chacune à la table avec probabilité p. Le théorème qui suit montre qu'ajouter une table réduit le nombre de solutions du problème d'un facteur p en moyenne.

Théorème 2. *Soit $\mathcal{P}$ un CSP, et T une contrainte table générée aléatoirement avec probabilité p. Alors*

$$\mathbb{E}\left(|Sols(\mathcal{P} \wedge T)|\right) = p|Sols(\mathcal{P})|$$

Démonstration. Pour $\sigma \in Sols(\mathcal{P})$, soit γ_σ une variable aléatoire valant 1 si et seulement si $\sigma \in Sols(\mathcal{P} \wedge T)$. Pour connaître $\mathbb{P}(\gamma_\sigma = 1)$ il suffit de savoir avec quelle probabilité σ satisfait T. Soient $X_{i_1}, \ldots, X_{i_v}$ les variables choisies dans la table T. Chaque instanciation de ces variables a été ajoutée dans la table avec probabilité p, notamment l'instanciation $(\sigma(X_{i_1}), \ldots, \sigma(X_{i_v}))$. Cela signifie que σ satisfait la contrainte table T avec probabilité p. On a donc $p = \mathbb{P}(\gamma_\sigma = 1) = \mathbb{E}(\gamma_\sigma)$. Cela nous donne

$$\begin{aligned}
\mathbb{E}\left(|Sols(\mathcal{P} \wedge T)|\right) &= \mathbb{E}\left(\sum_{\sigma \in Sols(\mathcal{P})} \gamma_\sigma\right) \\
&= \sum_{\sigma \in Sols(\mathcal{P})} \mathbb{E}(\gamma_\sigma) \\
&= \sum_{\sigma \in Sols(\mathcal{P})} p \\
&= p|Sols(\mathcal{P})|
\end{aligned}$$

□

3.2 Algorithme d'échantillonage

Avant de présenter l'algorithme d'échantillonage, il faut présenter les fonctions auxilliaires qui sont utilisées. La première fonction est ElementAleatoire(S) qui renvoie un élément aléatoire pris uniformément dans S. La seconde fonction est TrouveSolutions($\mathcal{P}, s$), qui énumère des solutions jusqu'à en avoir trouvé s, puis les renvoie. Ce qu'il faut remarquer sur cette fonction est que si elle renvoie s solutions, alors $|Sols(\mathcal{P})| \geq s$, et si elle renvoie strictement moins de s solutions, alors toutes les solutions ont été trouvées. La recherche en profondeur faite dans les solveurs de contraintes facilite l'implémentation de cette fonction.

L'algorithme d'échantillonage de solutions fonctionne de la manière suivante : des contraintes tables sont ajoutées au problème pour réduire le nombre de solutions, et quand il y a moins de solutions qu'une valeur pivot préalablement choisie, une solution est renvoyée aléatoirement et uniformément parmi les solutions restantes. L'algorithme est présenté en détail dans l'algorithme 2. Une valeur pivot κ est choisie pour l'énumération successive des solutions des problèmes intermédiaires,

```
1  Fonction TirageParTables(P, κ, v, p)
       Données : Un CSP P, κ ≥ 2, v > 0,
                 0 < p < 1
       Résultat : Une solution du problème P
2      S ← TrouveSolutions(P, κ);
3      si |S| = 0 alors
4          retourner "Pas de solution";
5      tant que |S| = 0 ∨ |S| = κ faire
6          t ← TableAleatoire(P, v, p);
7          S ← TrouveSolutions(P ∧ t, κ);
8          si |S| ≠ 0 alors
9              P ← P ∧ t;
10     retourner ElementAleatoire(S);
```

Algorithme 2 : Algorithme de tirage par ajout de tables

ainsi que le nombre de variables par tables v, et la probabilité d'ajouter un tuple dans la table p.

L'algorithme commence par énumérer κ solutions du problème initial. Il s'arrête immédiatement si le problème n'a pas de solutions, ou qu'il y a moins de κ solutions. Si le problème a plus de κ solutions, à chaque étape une nouvelle contrainte table est générée aléatoirement et si le problème avec cette contrainte a toujours des solutions, la contrainte est ajoutée au problème définitivement. L'algorithme s'arrête quand le problème est satisfiable et qu'il y a moins de κ solutions, et une solution est choisie aléatoirement et uniformément parmi les solutions du problème.

3.3 Ajout dichotomique des tables

Il est possible d'améliorer l'algorithme en modifiant le nombre de tables ajoutées à chaque étape. Au début de la recherche une table a une faible probabilité de rendre le problème inconsistant, donc il est plus judicieux pour réduire le nombre de résolutions d'ajouter plusieurs tables d'un coup au problème. Cet algorithme est inspiré de la recherche dichotomique dans un espace non borné : commencer par trouver i tel que la solution est entre 2^i et $2^{i+1} - 1$, puis faire une recherche dichotomique habituelle entre 2^i et $2^{i+1} - 1$.

L'algorithme d'ajout dichotomique de tables est présenté dans l'algorithme 3 et vise à remplacer les lignes 6 à 9 de l'algorithme 2. Soit n le nombre de tables ajoutées à l'étape précédente, on choisit $nbTables = 1$ si $n = 0$, ou $nbTables = 2n$ sinon, puis $nbTables$ sont générées et stockées dans la liste $\mathcal{T}$. L'algorithme énumère ensuite κ solutions au problème auquel on a ajouté toutes les contraintes de $\mathcal{T}$, et si il n'y a pas de solution il supprime la moitié des contraintes présentes dans $\mathcal{T}$. La procédure s'arrête quand le problème est satisfiable, ou que $|\mathcal{T}| = 0$.

1 **Fonction**
AjoutDichotomique$(\mathcal{P}, nbTables, \kappa, v, p)$
Données : Un CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, $nbTables > 0$, $\kappa \geq 2$, $v > 0$, $0 < p < 1$
Résultat : $\mathcal{P}$ augmenté des nouvelles contraintes table, et le nombre de tables ajoutées
2 $\mathcal{T} \leftarrow$ tableau de taille $nbTables$;
3 **pour** $i = 0$ *to* $nbTables - 1$ **faire**
4 $\mathcal{T}[i] \leftarrow$ TableAleatoire$(\mathcal{P}, v, p)$;
5 $S \leftarrow$ TrouveSolutions$(\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa)$;
6 **tant que** $|S| = 0 \wedge |\mathcal{T}| > 0$ **faire**
7 $\mathcal{T} \leftarrow \mathcal{T}[0 : |\mathcal{T}|/2[$;
8 $S \leftarrow$ TrouveSolutions$(\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa)$;
9 **retourner** $\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, |\mathcal{T}|$;

Algorithme 3 : Algorithme dichotomique d'ajout de tables

3.4 Discussion

Qualité de la division des tables : Dans la preuve du théorème 2, les variables aléatoires $(\gamma_\sigma)_{\sigma \in Sols(\mathcal{P})}$ ne sont pas indépendantes. Par exemple, soient σ_1 et σ_2 deux solutions du problème qui ne diffèrent que sur une seule variable X, alors

$$\mathbb{P}(\gamma_{\sigma_2} = 1 | \gamma_{\sigma_1} = 1) = p \cdot \mathbb{P}(X \in scp(T)) + 1 \cdot \mathbb{P}(X \notin scp(T)) \quad (1)$$

Cela signifie que si la table ne porte pas sur toutes les variables, alors elle ne sépare pas bien les regroupements de solutions qui prennent la même valeur sur plusieurs variables. Cette notion d'indépendance est centrale dans les approches de Kuldeep Meel [10] pour prouver l'uniformité du tirage. Ici le tirage n'est pas uniforme, mais est quand même une amélioration par rapport à la stratégie RandomVarDom.

La formule 1 sur la non indépendance nous montre aussi que plus la table contient de variables, plus les variables aléatoires γ_σ sont indépendantes. Des tables contenant toutes les variables du problème rendraient indépendantes les variables aléatoires γ_σ, et donneraient ainsi une garantie théorique, mais elles seraient impossible à générer en pratique.

Influence des paramètres Trois paramètres doivent être choisis pour exécuter l'algorithme. Nous pouvons déjà estimer quel est l'impact de chaque paramètre sur le temps de calcul, et sur la qualité de l'aléatoire.

— Le pivot détermine combien de solutions au maximum seront énumérées à chaque étape. Avoir un pivot élevé nécessite plus de résolution car à chaque étape il faut trouver plus de solutions.
— Comme vu précédemment, augmenter le nombre de variables doit améliorer l'aléatoire, mais va aussi augmenter exponentiellement le nombre de tuples dans les tables ce qui aura un impact négatif sur le temps de calcul.
— Réduire la probabilité d'ajouter un tuple à une table doit améliorer le temps de calcul, car les tables sont alors plus petites donc la propagation de ces tables sera plus rapide, et le nombre de tables ajoutées sera aussi plus faible car le problème sera plus vite réduit.

Ces hypothèses sont vérifiées dans la section 4 expérimentalement.

4 Expériences

Cette section présente les expériences faites pour tester notre nouvelle approche. Le premier but est d'évaluer le comportement aléatoire de l'approche, et ensuite d'évaluer le temps de calcul, en se comparant à la stratégie RandomVarDom. Le code est disponible en ligne [1], avec les scripts pour générer les figures présentées dans cet article.

4.1 Implémentation

L'implémentation a été faite en Java 11 en utilisant le solveur de contraintes `choco-solver` version 4.10.6 [12]. Il est possible de créer le modèle directement en utilisant `choco-solver`, ou alors en donnant en paramètre un fichier au format FlatZinc. La stratégie de recherche utilisée est celle par défaut du solveur (`dom/Wdeg` [3] et `lastConflict` [9]). Si le fichier FlatZinc déclare une stratégie, elle est utilisée.

Une amélioration a été apportée en ajoutant une étape de propagation du solveur avant la génération d'une table (avant la ligne 6 de l'algorithme 2). Cela permet lors de la génération des tables de ne pas avoir à énumérer des tuples qui auraient été immédiatement éliminés par la propagation.

Par la suite, l'algorithme utilisé est le tirage par tables amélioré avec l'ajout dichotomique des tables.

Gestion de l'aléatoire Le générateur de nombres aléatoires utilisé est celui présent de base en Java : `java.util.Random`. Ce générateur utilise une formule de congruence linaire pour modifier une graine sur 48 bits donnée en entrée. La documentation de Java pointe vers [8], section 3.2.1 pour plus d'information. Ce générateur d'aléatoire a des défauts (notamment

1. https://github.com/MathieuVavrille/tableSampling

une periode de 2^{48}), mais pour l'utilisation qui est faite ici il est suffisant (comme montré dans [2]).

L'implémentation utilise une unique instance du générateur aléatoire qui est passé en argument à toutes les fonctions ayant besoin de générer des nombres aléatoires. Cela permet d'éviter des effets de non indépendance qui peuvent advenir à cause d'une mauvaise création de graines aléatoires.

4.2 Les problèmes utilisés

L'approche mise en place est indépendante des contraintes des modèles résolus, cela nous a permis de l'appliquer sur quatre problèmes différents, dont trois issus de problèmes réels. Nous présentons ici les modèles et leurs caractéristiques.

N-reines Le premier problème est celui des N-reines, qui consiste à placer N reines sur un plateau d'échec de taille $N \times N$ de sorte à ce qu'aucune n'en attaque une autre (les reines attaquent les pièces dans les huit directions et aussi loin que possible). Nous avons choisi la modélisation classique avec N variables qui ont un domaine $[1, N]$, une contrainte `all_different`, et des contraintes binaires d'inégalité (pour les attaques en diagonale). Le nombre de solutions du problème est connu pour les premières valeurs de N [16].

Configuration des Renault Méganes Il s'agit d'un problème de configuration des Renault Mégane introduit dans [1], et déjà utilisé dans [7] pour la recherche de solutions diverses. Il y a 101 variables, avec des domaines contenant jusqu'à 43 valeurs, et les 113 contraintes sont modélisées par des contraintes tables, dont la plupart non-binaires. Ce problème est peu contraint, et a ainsi plus de $1.4 \cdot 10^{12}$ solutions.

On call rostering Ce problème modélise le système de "gardes", notamment utilisé par les personnels de santé. Le problème est disponible dans les benchmarks de MiniZinc [2], et contient différents types de contraintes, comme des contraintes linéaires, des contraintes globales `count`, des valeurs absolues, des implications, et des contraintes tables. Plusieurs jeux de données sont disponibles, mais seul le plus petit (`4s-10d.dzn`) a été utilisé ici. Il s'agit d'un problème d'optimisation (de minimisation), donc il a fallu le transformer en problème de satisfaction en bornant l'objectif. Ce problème a pour valeur optimale 1 :

— Il y a 136 solutions en fixand $obj \leq 1$
— Il y a 2099 solution en fixant $obj \leq 2$
— Il y a plus de 10000 solutions en fixant $obj \leq 3$

2. https://github.com/MiniZinc/minizinc-benchmarks/tree/master/on-call-rostering

La génération aléatoire de solutions permet d'utiliser le solveur comme un outil d'aide à la décision pour la personne créant les plannings (lui donner plusieurs plannings à comparer), et apporte une forme d'équité entre les personnels. En effet, des méthodes de recherche orientées peuvent favoriser certaines personnes, au détriment d'autres.

Feature Models Les *Feature Models* sont des problèmes de gestion de logiciels, et visent à aider à la prise de décision quant à l'ordre d'implémentation des fonctionnalités d'un logiciel. Le problème est spécifié au format MiniZinc dans l'article [13], en utilisant les données de [14]. Il s'agit encore ici d'un problème d'optimisation (de maximisation), la valeur optimale étant 20222 la résolution va ajouter la contrainte $obj \geq 17738$ ce qui en fait un problème de satisfaction à 95 solutions.

4.3 Qualité de l'aléatoire

Les premières expériences conduites visent à évaluer la qualité de l'aléatoire, c'est à dire, savoir si les solutions sont tirées aléatoirement et uniformément. Les résultats qui suivent montrent que les solutions ne sont pas tirées uniformément, mais que l'approche par tables est *plus uniforme* que la stratégie RandomVarDom.

4.3.1 Évaluation de l'uniformité

Pour avoir une mesure quantitative de l'uniformité du tirage des solutions, nous avons utilisé le test du χ^2. Connaissant le nombre de solutions $nbSols$ d'un problème (et en numérotant ces solutions), $nbSample$ tirages sont fait et le nombre d'occurences $nbOcc_i$ de chaque solution $i \in \{1, \ldots, nbSols\}$ est compté. Nous calculons la valeur de la variable

$$z_{exp} = \sum_{k=1}^{nbSols} \frac{(nbOcc_k - nbSamples/nbSols)^2}{nSamples/nbSols}$$

et ensuite la valeur-p du test [3] (c'est à dire, la probabilité que la loi du χ^2 prenne une valeur plus extrême que z_{exp}). Cette valeur-p nous donne une évaluation quantitative de la qualité de l'aléatoire. Plus particulièrement, un très grand nombre de tirages sont faits, et l'évolution de la valeur-p en fonction du nombre de tirage est affichée. Dans notre cas, comme le tirage n'est pas uniforme, la valeur-p va tendre vers 0 quand le nombre de tirages augmente, mais nous remarquerons que le tirage par tables a une valeur-p qui tend moins vite vers 0 que le tirage en utilisant RandomVarDom.

3. Nous utilisons librairie "Apache Commons Mathematics Library" (https://commons.apache.org/proper/commons-math/) pour les calculs de probabilités

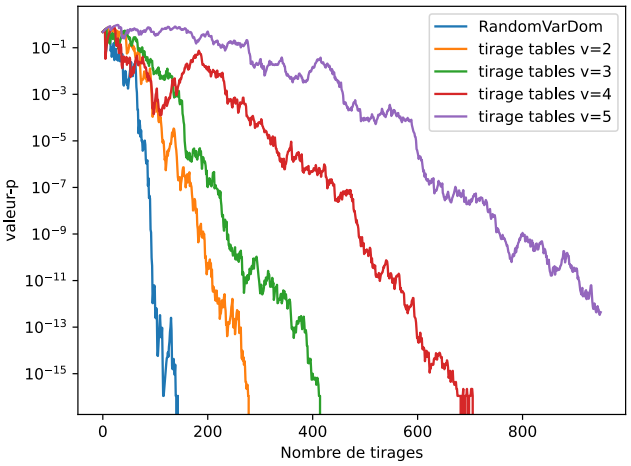

FIGURE 1 – Évolution de la valeur-p sur le problème Feature Models, avec différents nombre de variables dans les tables où $\kappa = 2$ et $p = 1/2$

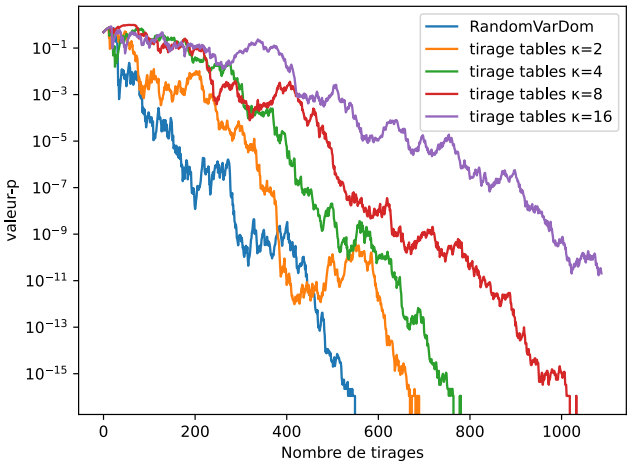

FIGURE 2 – Évolution de la valeur-p sur le problème On Call Rostering avec $obj \leq 1$, avec différents pivots et où $v = 4$ et $p = 1/8$

Pour effectuer ce test, il faut connaître le nombre de solutions $nbSols$, et pouvoir tirer plusieurs fois $nbSols$ solutions, donc l'évaluation de l'aléatoire ne peut se faire que sur des petits problèmes. Les problèmes qui ont été utilisés : 9-reines (352 solutions), Feature Models avec la contrainte $obj \geq 17738$ (95 solutions), et On Call Rostering avec les contraintes $obj \leq 1$ et $obj \leq 2$ (136 et 2099 solutions).

Nous voulons aussi évaluer l'impact de l'évolution d'un paramètre (nombre de variables, pivot, ou probabilité) de l'algorithme de tirage par tables sur la qualité de l'aléatoire. Pour cela, dans les figures qui suivent sont affichés l'évolution de la valeur-p pour la stratégie RANDOMVARDOM, ainsi que pour différents paramètres du tirage par tables, en faisant varier un paramètre à la fois. La légende donne les paramètres associés à chaque exécution (v pour le nombre de variables, κ pour le pivot et p pour la probabilité).

Remarque. *Les figures montrent la valeur-p sur une échelle logarithmique, car elle tend très vite vers 0. De plus, comme les calculs sont fait en utilisant une représentation en machine des nombres flottants, une valeur de valeur-p inférieure à* 10^{-16} *sera considérée comme nulle.*

4.3.2 Impact du nombre de variables

Nous commençons par faire varier le nombre de variable dans les tables, la figure 1 montre l'évolution de la valeur-p, où des tirages ont été fait avec différentes valeurs pour v, et où on a fixé $\kappa = 2$ et $p = 1/2$. On constate premièrement une amélioration par rapport à la stratégie RANDOMVARDOM, car la valeur-p décroît beaucoup plus vite. Ensuite, notre intuition quand à l'utilisation de tables avec plus de variables est vérifiée, car quand les tables portent sur plus de variables, la valeur-p décroît plus lentement.

4.3.3 Impact du pivot

Nous voulons ensuite évaluer l'impact du pivot sur la qualité de l'aléatoire, la figure 2 montre l'évolution de la valeur-p, avec différentes valeurs de κ, en fixant $v = 4$ et $p = 1/8$, sur le problème On Call Rostering, avec la contrainte $obj \leq 1$. Quand le pivot augmente, la valeur-p décroît moins vite. Plus le pivot est grand, moins de tables seront ajoutées au problème, le cas extrême étant le cas où le pivot est plus grand que le nombre de solutions et dans ce cas l'algorithme revient à énumérer toutes les solutions, et en piocher une aléatoirement (ce n'est pas le cas ici, car il y a 136 solutions, et le pivot vaut au maximum 16).

4.3.4 Impact de la probabilité

Nous avons fini par évaluer l'impact de la probabilité d'ajouter un tuple dans les tables générées. La figure 3 montre l'évolution de la valeur-p sur le problème On Call Rostering, où la contrainte $obj \leq 2$ a été ajoutée, en fixant les paramètres $v = 5$ et $\kappa = 16$ et en faisant varier p. Encore une fois, le tirage par tables montre une amélioration par rapport à la stratégie RANDOMVARDOM, mais il n'y a pas d'influence claire de la probabilité d'ajouter un tuple dans la table sur la qualité de l'aléatoire.

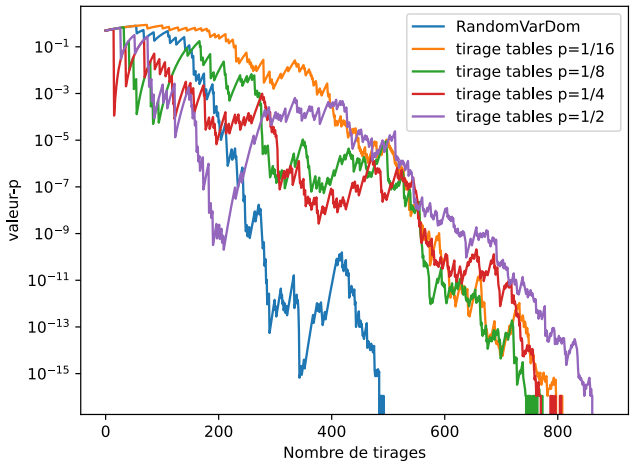

FIGURE 3 – Évolution de la valeur-p sur le problem On Call Rostering avec $obj \leq 2$, avec différentes probabilités et où $\kappa = 16$ et $v = 5$

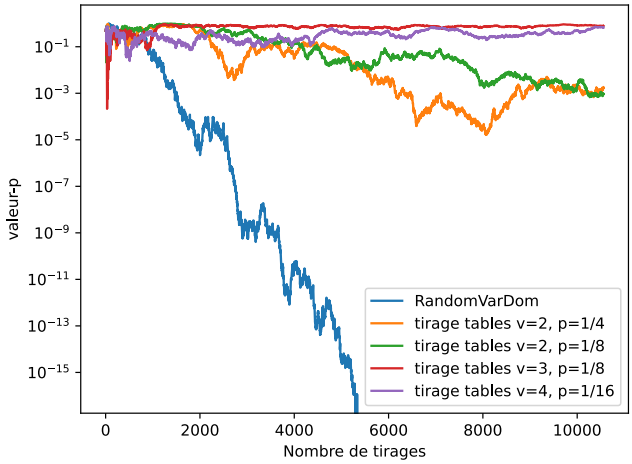

FIGURE 4 – Évolution de la valeur-p sur le problème des 9-reines, avec différents paramètres pour v et p pour le tirage par tables, avec $\kappa = 8$

4.3.5 Qualité de l'aléatoire obtenu

On constate que sur le problème des N-reines, la qualité de l'échantillonnage est particulièrement bonne. La figure 4 montre l'évolution de la valeur-p sur ce problème, avec différents paramètres pour le tirage par tables. On constate que la valeur-p ne tend pas vers 0 avec le tirage par tables, alors qu'elle tend toujours vers 0 avec la stratégie RANDOMVARDOM. Nous supposons que les solutions sont assez bien réparties dans l'espace de recherche, ce qui expliquerait ces bonnes performances.

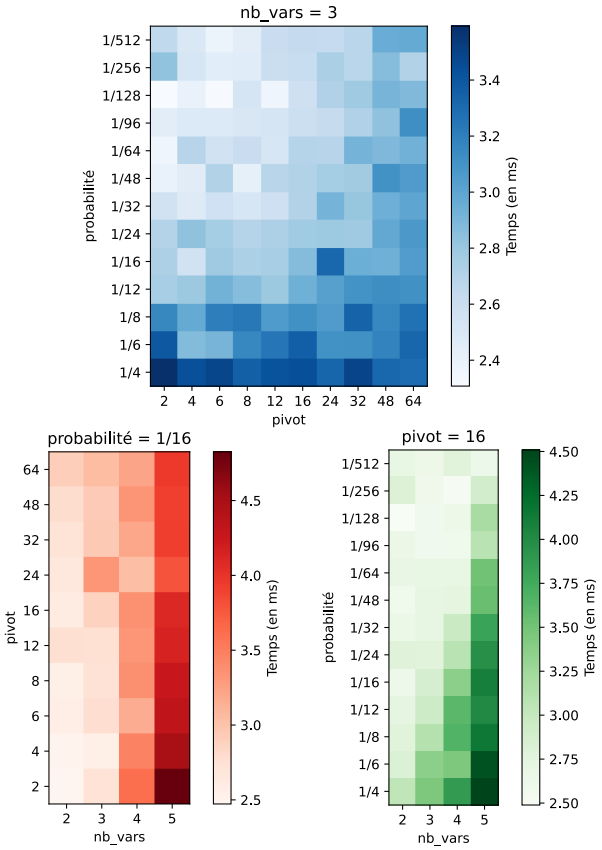

FIGURE 5 – Cartes de chaleur du temps pour tirer une solution, en fixant différent paramètres, sur le problème d'On Call Rostering, avec la contrainte $obj \leq 3$

4.4 Temps d'exécution

L'évaluation du temps d'exécution se fait en deux parties. Le choix des paramètres est important dans le temps d'exécution, donc il faut trouver quels paramètres ont un impact positif sur le temps de calcul. Par la suite, nous utiliserons des problèmes avec plus de 10000 solutions, et 50 tirages seront fait pour faire une moyenne du temps pour tirer une solution avec les différentes méthodes.

4.4.1 Impact des paramètres

Pour montrer l'impact des paramètres sur le temps pour tirer une solution, nous fixons un des paramètres, et faisons varier les deux autres pour ainsi avoir des cartes de chaleur du temps en fonction de différents paramètres. La figure 5 montre les 3 cartes de chaleur obtenues en fixant le nombre de variables, le pivot ou la probabilité. Les hypothèses faites en section 3.4 sont vérifiées :

a. diminuer le nombre de variables dans les tables améliore le temps de calcul

b. augmenter le pivot augmente le temps de calcul

c. diminuer la probabilité d'ajouter un tuple aux tables améliore le temps de calcul

Comme nous l'avons vu précédemment, augmenter le pivot améliore la qualité de l'aléatoire. Nous avons donc ici un compromis à faire entre le temps de calcul (réduire le pivot ou le nombre de variables) et la qualité de l'aléatoire (augmenter le pivot ou le nombre de variables).

Nous avons aussi vu que la probabilité n'avait pas d'impact significatif sur la qualité de l'aléatoire, donc il est possible de choisir des probabilités très faibles pour que les tables soient plus petites, et ainsi avoir une propagation plus rapide, ainsi que moins de tables ajoutées lors de l'algorithme.

4.4.2 Comparaison par rapport à RandomVarDom

Pour finir, il faut comparer les temps à la stratégie RandomVarDom. La table 1 montre pour huit ensembles de paramètres différents le temps pour tirer une solution avec le tirage par tables, contre le temps en utilisant la stratégie RandomVarDom, la dernière colonne donnant le ratio $T_{tables}/T_{RandomVarDom}$. Nous remarquons que notre approche est plus lente que la stratégie RandomVarDom sur les problèmes de configuration de la Renault Mégane, ainsi que sur les 12-reines. En revanche sur le problème On Call Rostering (où l'objectif a été borné par 3, le problème a ainsi plus de 10000 solutions), les performances sont meilleures.

La raison pour laquelle le tirage par tables fonctionne particulièrement bien sur On Call Rostering est que ce problème a très peu de solutions, et surtout très peu de valeurs rendent le problème satisfiable pour chaque variable. Les problèmes de configuration de la Renault Mégane et des 12-reines ont très peu de valeurs qui sont interdites, ce qui fait que peu importe le choix fait par la stratégie d'exploration, les retours en arrière sont très rares. En revanche sur le problème On Call Rostering, beaucoup de valeurs initialement dans les domaines ne débouchent pas sur une solution. Cela signifie que la stratégie RandomVarDom va faire des choix de couples (variable,valeur) qui ont une forte probabilité de ne pas donner de solution, ainsi obligeant un retour en arrière dans l'arbre de décision. Une stratégie optimisée pour la résolution de problème, même simple va fortement améliorer le temps de calcul, et notre approche de tirage par tables permet de laisser le solveur choisir la stratégie de recherche en fonction de la structure du problème.

Problème	RandomVarDom	Tirage par tables				Ratio
		v	κ	p	Temps	
Megane	32 ms	2	16	1/16	55 ms	1.7
				1/32	59 ms	1.8
			32	1/16	89 ms	2.8
				1/32	86 ms	2.7
		3	16	1/16	74 ms	2.3
				1/32	67 ms	2.1
			32	1/16	83 ms	2.6
				1/32	65 ms	2.0
On Call Rostering	38 ms	2	16	1/16	14 ms	0.36
				1/32	14 ms	0.38
			32	1/16	15 ms	0.4
				1/32	18 ms	0.46
		3	16	1/16	18 ms	0.47
				1/32	15 ms	0.4
			32	1/16	19 ms	0.5
				1/32	17 ms	0.44
12-reines	2 ms	2	16	1/16	4 ms	2.4
				1/32	4 ms	2.3
			32	1/16	7 ms	4.1
				1/32	7 ms	3.9
		3	16	1/16	10 ms	5.6
				1/32	8 ms	4.3
			32	1/16	12 ms	6.9
				1/32	11 ms	6.0

Table 1 – Comparaison du temps pour tirer une solution entre RandomVarDom et le tirage par tables. Les temps sont les moyenne des temps pour 50 tirages

5 Conclusion

Nous avons présenté ici une nouvelle approche pour tirer des solutions aléatoirement en ajoutant au problème des contraintes tables tirées aléatoirement jusqu'à ce que le problème soit assez petit pour pouvoir être énuméré. Bien qu'il n'y ait pas de garantie d'uniformité, l'approche montre expérimentalement une amélioration de la qualité de l'aléatoire par rapport à la stratégie de recherche RandomVarDom. De plus, le fait que la stratégie de recherche ne soit pas fixée permet de garder un temps de calcul du même ordre de grandeur qu'avec la stratégie RandomVarDom.

L'algorithme de tirage étant indépendant des contraintes utilisées pour diviser l'espace, il serait intéressant d'étudier si d'autres contraintes peuvent être générées aléatoirement et permettre un tirage plus uniforme des solutions, sans devenir trop coûteuses. Une analyse plus approfondie de la structure des problèmes ou des solutions permettrait aussi de choisir des paramètres plus appropriés, tant pour avoir un plus faible temps de calcul que pour avoir un tirage plus uniforme. Il serait aussi intéressant d'étudier plus particulièrement les problèmes d'optimisation, et les notions de tirage dans ces problèmes.

Références

[1] Jérôme Amilhastre, Hélene Fargier et Pierre Marquis : Consistency restoration and explanations in dynamic csps—application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[2] Eric Bach : Realistic analysis of some randomized algorithms. *In Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 453–461, 1987.

[3] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre et Lakhdar Sais : Boosting systematic search by weighting constraints. *In ECAI*, volume 16, page 146, 2004.

[4] Rina Dechter, Kalev Kask, Eyal Bin, Roy Emek *et al.* : Generating random solutions for constraint satisfaction problems. *In AAAI/IAAI*, pages 15–21, 2002.

[5] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin et Pierre Schaus : Compact-table : efficiently filtering table constraints with reversible sparse bit-sets. *In International Conference on Principles and Practice of Constraint Programming*, pages 207–223. Springer, 2016.

[6] Vibhav Gogate et Rina Dechter : A new algorithm for sampling csp solutions uniformly at random. *In International Conference on Principles and Practice of Constraint Programming*, pages 711–715. Springer, 2006.

[7] Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan et Toby Walsh : Finding diverse and similar solutions in constraint programming. *In AAAI*, volume 5, pages 372–377, 2005.

[8] Donald E Knuth : *Art of computer programming, volume 2 : Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[9] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary et Vincent Vidal : Last conflict based reasoning. *In* Gerhard Brewka, Silvia Coradeschi, Anna Perini et Paolo Traverso, éditeurs : *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 de *Frontiers in Artificial Intelligence and Applications*, pages 133–137. IOS Press, 2006.

[10] Kuldeep S Meel : Constrained counting and sampling : bridging the gap between theory and practice. *arXiv preprint arXiv :1806.02239*, 2018.

[11] Karl Pearson : X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[12] Charles Prud'homme, Jean-Guillaume Fages et Xavier Lorca : *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[13] Björn Regnell et Krzysztof Kuchcinski : Exploring software product management decision problems with constraint solving-opportunities for prioritization and release planning. *In 2011 Fifth International Workshop on Software Product Management (IWSPM)*, pages 47–56. IEEE, 2011.

[14] Günther Ruhe et Moshood Omolade Saliu : The art and science of software release planning. *IEEE software*, 22(6):47–53, 2005.

[15] Yevgeny Schreiber : Value-ordering heuristics : Search performance vs. solution diversity. *In International Conference on Principles and Practice of Constraint Programming*, pages 429–444. Springer, 2010.

[16] Neil James Alexander Sloane : https://oeis.org/A000170.