Modélisation Programmation par Contraintes

Kévin ALESSANDRO

Mai 2020

Master 1 Informatique

Université Côte d'Azur

Contenu

1. **Génération de Sudoku**
2. **Niveau de difficulté**
3. **Code**

1. Génération de Sudoku

La première étape de ce projet était d'avoir un programme qui puisse créer un sudoku. C'est-à-dire, une grille de 9x9 cellules (par défaut) avec des cellules pré-affectées et des cellules vierges, qui respectent les règles du jeu et où il n'existe qu'une seule solution, une seule unique valeur par cellule. Pour accomplir cet objectif j'ai accepté la proposition d'algorithme du sujet : de mon point de vue, il est effectivement plus simple de retirer des valeurs à une grille complète que l'inverse.

Accessoirement, le programme accepte des sudokus de taille diverses car la plupart des méthodes de déduction ne changent pas. Je considère le sudoku par défaut de dimension n = 3 (n*n = 9 cases de long), un sudoku de dimension n = 4 aurait 16 cases de long, n = 5 en aurait 25, etc.

D'un point de vue modélisation par contraintes les règles du sudoku se représentent dans mon programme par un AllDiff sur les valeurs des cellules d'une ligne, un autre AllDiff sur les valeurs des cellules d'une colonne, et un dernier AllDiff sur les valeurs des blocs de cellules de taille n*n.

Pour générer des grilles complètes j'ai utilisé Choco Solver. Pour affecter des valeurs aux 81 (n = 3 sudoku) variables du sudoku, j'utilise la technique de recherche par défaut du solveur: un DFS. C'est un comportement déterministe : la première grille complète que Choco génère à partir d'une grille vide sera toujours la même après chaque lancement du programme. Choco génère un grand nombre de grilles (dans une limite de temps – environ 3 seconds. Sur mon ordi, environ 450 grilles sont générées), puis une grille est choisie au hasard en utilisant le *reservoir sampling* : la 1ère grille est choisie avec 1/1 chance. La 2ème grille remplace la première grille avec 1/2 chance. La 3ème grille remplace la dernière grille choisie avec 1/3 chance... la n-ème grille avec 1/n chance. J'utilise cette technique afin de piocher une grille aléatoire parmi les n générées, sans avoir à stocker les n grilles (n grilles contenant chacune 81 valeurs peuvent devenir très lourdes en mémoire)

A partir de ces grilles complètes, j'utilise de l'aléatoire et Choco Solver pour respectivement générer un sudoku et m'assurer qu'une seule solution existe.
Chacune des 81 cases pour un sudoku standard seront traitées, mais l'ordre dans

lesquelles elles le sont influence le sudoku généré.
L'algorithme choisit aléatoirement une cellule. Si cette cellule possède une valeur ET n'a pas encore été traitée :

1) Cette cellule est rendue vierge
2) Le sudoku avec cette modification est résolu par Choco
3) Si il y a plus qu'une solution, alors la cellule reprend sa valeur initiale
4) Cette cellule peu importe le résultat est considérée comme "traitée"

Si la cellule choisie choisie est déjà traitée (vierge ou non) : on choisit la prochaine cellule du sudoku, à partir de celle choisie par l'aléatoire. Si elle est également déjà traitée, on continue jusqu'à trouver une cellule non traitée, et dans la limite des 81 cellules du sudoku.

Cette dernière technique permet de s'assurer que pas plus de $(n*n)^2$ nombres aléatoires seront générés. Si on choisit de relancer l'aléatoire lorsque ce dernier renvoit une cellule traitée, on pourrait avoir des temps de calculs assez irréguliers (surtout quand n grandit)

Aussi, une cellule traitée ne devrait pas être analysée une seconde fois car si la rendre vierge à l'étape de calcul t crée plusieurs solutions, alors la rendre vierge à une étape de calcul t' avec $t' > t$ créera également plusieurs solutions.

Pour m'assurer que les sudokus générés par mon programme n'ont qu'une et unique solution (en dehors des statistiques printées par Choco) j'ai utilisé https://www.sudoku-solutions.com/ ce super site qui donne beaucoup d'infos.

En pratique, les grilles trouées générées par mon programme ont environ entre 20 et 25 cellules remplies. Mon programme calcule une vingtaine (valeur fixée dans le code) de différentes combinaisons de trous, puis retourne la grille avec le moins d'indices (cellules pré-remplies) possibles.

2. Niveau de difficulté

La deuxième partie du sujet concerne l'évaluation du niveau de difficulté d'un sudoku. Le travail que j'ai interprété à partir du sujet était de créer des modèles qui jugent la difficulté d'un sudoku ; et non la génération de sudokus de difficulté variable.

Une première heuristique assez banale est de dire que plus une grille possède d'indices, plus la probabilité d'utiliser des techniques de déductions "avancées" est faible.

Par manque de temps lié à des soucis techniques, je n'ai pas pu réaliser un programme qui évalue la difficulté d'un sudoku, même si il y a quelque artifacts dans le code. Ceci dit mon intuition était d'évaluer, à chaque étape de calcul, le niveau de déduction logique nécéssaire pour procéder à une prochaine étape.

Je considèrais comme "très facile" un sudoku qui peut être résolu uniquement avec la déduction la plus naturelle : si il n'y a qu'un candidat possible pour une cellule, la valeur de cette cellule est ce candidat ("naked single"). En Choco, on traduirait ça par une recherche des variables à plus petit domaine, et on vérifierait qu'à chaque étape, la variable choisie, grâce à l'arc-consistance qui réduit son domaine, a bien un domaine de 1 valeur. Si une étape n'utilise pas le naked single, le sudoku est "au moins" "facile". En d'autres mots, à n'importe quelle étape, il ne doit jamais y avoir QUE des cellules avec 2 candidats ou plus.

Je considérais comme "facile" un sudoku qui utilise en plus les hidden single : une cellule avec plus d'un candidat dont l'un d'entre eux n'apparaissant pas dans la même ligne, colonne, ou bloc, a pour valeur ce candidat. Je vois toutefois difficilement comment orienter la stratégie de recherche de Choco pendant sa résolution pour appliquer cette technique de déduction. A ma connaissance, Choco Solver peut choisir ses variables selon des principes généraux (plus petit domaine, activité, plus grandes valeurs en premier...) les stratégies avancées du sudoku me semblent hors de sa portée.

Le swordfish, X/Y/V wings, double/triple pairs... sont réservés pour des sudokus moyens à difficiles, et enfin le backtracking qui n'est pas une déduction mais une recherche de force brute que ironiquement une force de calcul sait faire le mieux, serait réservés aux sudokus les plus complexes.

3. Code

Je présente ici mon programme et quelque explications techniques. Le projet entier est sur https://github.com/StardustMotion/sudoku_chocolat

3.1 Execution

Je conseille d'ouvrir le projet avec un IDE (Intellij, Eclipse...) et d'exécuter

Main.java dans le dossier main. Il y a du temps de calcul, quelque secondes. Le programme a fini quand il retourne "Process finished with exit code 0".

La première grille affichée est une grille générée aléatoirement. Puis, suivent des stats concernant le calcul qui a permit de choisir une grille aléatoire. Puis, le programme affiche une combinaison de trous pour la grille qui a été choisie.

3.2 Codage

Toutes les idées sont évoquées dans les parties 1 et 2, je donne juste quelque explications sur mon code ici.

3.2.1 MySudoku.java

Cette classe possède des constructeurs, un utilisé pour génerer des grilles, un pour les résoudre. Dans mon code une cellule vide est considérée comme ayant la valeur 0, inutilisée par le sudoku. Le solveur restart après avoir trouvé une grille – ça permet de varier la recherche et donc les grilles.

Pour définir la contrainte AllDiff sur un bloc : j'opère sur les blocs de gauche à droite et haut en bas. J'ai encodé la ligne et la colonne d'une information sur une valeur. Avec un nombre [0,80] pour un sudoku standard on peut obtenir la colonne et la ligne avec modulo et division euclidienne.

Je fais une copie profonde des objets de type IntVar[][] utilisés par Choco en int[][] pour mieux opérer dessus.

3.2.2 Main.java

Pour choisir une cellule à rendre vierge dans la fonction peckAHole, je navigue sur le sudoku là aussi avec une valeur [0,80] où je peux obtenir l'index (i,j) encodé sur ces 81 valeurs.

Merci de m'avoir lu en entier!