Boîtes-noires SAT, MILP, DLX pour le Sudoku
EJCIM 2018

Vincent Delecroix Bruno Grenet Damien Jamet

27 mars 2018

Dans cette feuille de travail on va s'intéresser à différents outils qui vont nous servir de boîtes noires pour la résolution et la génération de sudoku. Ces outils sont

1. les solveurs SAT,
2. les solveurs linéaires (en nombre entiers),
3. les "liens dansants" ou "algorithme X de Knuth".

Ces méthodes sont plus généralement utiles pour résoudre des problèmes combinatoires. Ce sera l'occasion de comparer ces différentes méthodes.

Nous étudions le Sudoku dans une grille $n^2 \times n^2$ dans lequel nous souhaitons placer des chiffres de 1 à n. La version standard du sudoku utilise $n = 3$ mais il sera intéressant de comparer les performances des différentes méthodes avec des n plus grands.

1 Version SAT

Pour l'utilisation du solveur SAT nous renvoyons au Chapitre 5.5.2 du livre "Informatique Mathématique. Une photographie en 2018". Nous utiliserons les variables $x_{i,j,k}$ qui indiquent si le nombre k est présent en position (i, j) (avec $0 \leq i < n^2$, $0 \leq j < n^2$, $1 \leq k \leq n^2$).

1. Écrire des fonctions `sat_encode(i, j, k, n)` et `sat_decode(s, n)` qui permettent de passer de triplets `(i, j, k)` que l'on utilise pour encoder notre problème à un entier de $\{1, \ldots, n^6\}$ que comprennent les solveurs SAT.
2. Écrire une fonction `sat_sudoku_constraints(solver, n)` qui prend en argument une instance `solver` de solveur SAT [1] et un entier `n` et ajoute à `solver` les contraintes du sudoku :
   — il y a exactement un chiffre dans chaque case (i, j),
   — chaque chiffre apparaît exactement une fois dans chaque ligne,
   — chaque chiffre apparaît exactement une fois dans chaque colonne,
   — chaque chiffre apparaît exactement une fois dans chaque sous-grille $n \times n$.
   Notez que les conditions ci-dessus sont redondantes et que plusieurs encodages en clauses sont possibles.
3. Écrire une fonction `sat_sudoku_fill(solver, mat, n)` qui prend en argument une instance `solver` du solveur SAT, une matrice `mat` $n^2 \times n^2$ composée d'éléments de $\{0, 1, \ldots, n\}$ et un entier `n` et ajoute au solveur la condition $x_{i,j,k} = 1$ si $mat[i][j] = k > 0$.
4. Écrire une fonction `sat_sudoku_one_solution(mat, n)` qui retourne une complétion possible de la matrice `mat` si elle existe et `None` sinon.
5. Vérifier que le sudoku 2×2 suivant peut être complété.

```
M0 = [[4, 0, 0, 0],
      [0, 1, 0, 0],
      [0, 0, 2, 0],
      [0, 3, 0, 0]]
```

1. Il faut au préalable installer la libraire `cryptominisat` avec la commande `sage -i cryptominisat`. Une instance de solveur SAT est crée avec la commande `sage: S = SAT()`.

6. Vérifier que le sudoku 3×3 suivant peut être complété.

```
M1 = [[5, 0, 0, 0, 8, 0, 0, 4, 9],
      [0, 0, 0, 5, 0, 0, 0, 3, 0],
      [0, 6, 7, 3, 0, 0, 0, 0, 1],
      [1, 5, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 2, 0, 8, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 1, 8],
      [7, 0, 0, 0, 0, 4, 1, 5, 0],
      [0, 3, 0, 0, 0, 2, 0, 0, 0],
      [4, 9, 0, 0, 5, 0, 0, 0, 3]]
```

7. Vérifier que le sudoku 2×2 suivant ne peut pas être complété.

```
M2 = [[4, 0, 0, 0],
      [2, 1, 0, 0],
      [0, 0, 2, 0],
      [0, 3, 0, 0]]
```

8. (*) modifier le solveur `pysat` [2] pour prendre en compte la contrainte "exactement une variable parmi ce groupe est vraie".
9. Comparer le nombre de contraintes du soduko avec le SAT standard et le SAT modifé considéré dans la question précédente.

2 Version MILP

Pour l'utilisation des solveurs linéaires en nombres entiers nous renvoyons au Chapitre 5.5.1 du livre "Informatique Mathématique. Une photographie en 2018".

8. Reprendre les questions de la partie 1 en utilisant cette fois un solveur linéaire : écrire en particulier une fonction `milp_sudoku_one_solution(mat, n)` qui complète la grille `mat`.

3 Les liens dansants

Le problème que résout l'algorithme X est le problème de *couverture exacte*. On se donne un ensemble de sous-ensembles A_0, A_1, ..., A_{m-1} d'un ensemble $E = \{0, 1, \ldots, n-1\}$ et on cherche l'ensemble des $\{A_{i_1}, \ldots, A_{i_k}\}$ formant une partition de E (c'est-à-dire que chaque élément de E apparaît une et une seule fois parmi les A_{i_j}). La structure de données utilise une matrice M faite de 0 et de 1 dont les colonnes sont indexées par les éléments de E et les lignes par les ensembles A_0, ..., A_{m-1} :

$$M_{ij} = \begin{cases} 1 & \text{si } j \in A_i \\ 0 & \text{sinon.} \end{cases}$$

Dans cette feuille de travail nous ne cherchons pas à comprendre le fonctionnement de cet algorithme [3]. Nous allons simplement l'utiliser via les fonctions `OneExactCover` et `AllExactCovers`. Ci-dessous nous prenons l'exemple de $A_0 = \{0, 1, 4\}$, $A_1 = \{2, 3, 4\}$, $A_2 = \{1, 2, 3\}$, $A_3 = \{0, 3\}$, $A_4 = \{1, 2\}$, $A_5 = \{0, 1\}$ et $A_6 = \{2, 3\}$ et $E = \{0, 1, 2, 3, 4\}$.

```
sage: M = matrix([[1,1,0,0,1],
....:             [0,0,1,1,1],
....:             [0,1,1,1,0],
....:             [1,0,0,1,0],
....:             [0,1,1,0,0],
```

2. https ://bitbucket.org/lorensi/pysat
3. On pourra pour cela consulter la page `https://en.wikipedia.org/wiki/Dancing_Links` ou l'article original de D. Knuth `https://arxiv.org/abs/cs/0011047`.

```
....:              [1,1,0,0,0],
....:              [0,0,1,1,0]])
sage: OneExactCover(M) # renvoie une liste de lignes formant une partition
[(1, 1, 0, 0, 1), (0, 0, 1, 1, 0)]
```

La fonction `AllExactCovers` retourne un itérateur sur les solutions.

```
sage: for s in AllExactCovers(M):
....:     print(s)
[(1, 1, 0, 0, 1), (0, 0, 1, 1, 0)]
[(0, 0, 1, 1, 1), (1, 1, 0, 0, 0)]
```

Pour encoder le problème du sudoku, on va utiliser une matrice de couverture avec n^6 lignes et $4n^4$ colonnes. Chaque ligne correspond au choix d'un chiffre k en position (i, j) (similaire à la variable $x_{i,j,k}$ de 1). Chaque colonne encode une contrainte :

— un chiffre dans chaque case (i, j),
— chaque chiffre apparaît dans chaque ligne,
— chaque chiffre apparaît dans chaque colonne,
— chaque chiffre apparaît dans chaque sous-grille $n \times n$.

9. Écrire une fonction `dlx_sudoku_one_solution(mat, n)` qui complète la grille de sudoku `mat` via `OneExactCover`.

4 Compter les solutions

Nous nous intéressant maintenant à savoir de combien de façon une grille peut être complétée.

10. Écrire une fonction `sat_has_unique_solution(mat, n)` qui détermine si la matrice `mat` peut-être complétée de manière unique en sudoku (on pourra utiliser un code de retour avec 3 alternatives : `-1` pour pas de complétion, `0` pour plusieurs valeurs de retour
11. Écrire une fonction `sat_count(mat, n)` qui compte le nombre de complétions valides de la grille `mat` en utilisant un solveur SAT (indice : vous pouvez ajouter successivement des clauses qui consistent à interdire une solution donnée).
12. Écrire une fonction `dlx_count(mat, n)` avec les mêmes spécifications mais utilisant `AllExactCovers` de la partie 3.
13. Vérifier que les deux fonctions donnent les mêmes résultats sur les matrices `M0`, `M1` et `M2`.
14. Combien y a-t-il de complétions possibles de la matrice suivante ?

```
M3 = [[5, 0, 0, 0, 8, 0, 0, 4, 9],
      [0, 0, 0, 5, 0, 0, 0, 3, 0],
      [0, 6, 7, 3, 0, 0, 0, 0, 1],
      [0, 5, 0, 0, 0, 0, 0, 0, 0],
      [0, 0, 0, 2, 0, 8, 0, 0, 0],
      [0, 0, 0, 0, 0, 0, 0, 1, 8],
      [7, 0, 0, 0, 0, 4, 1, 0, 0],
      [0, 3, 0, 0, 0, 2, 0, 0, 0],
      [4, 9, 0, 0, 5, 0, 0, 0, 3]]
\item (*) Ajouter une fonction au solveur \texttt{pysat}\footnote{https://bitbucket.org/lor
compter les solutions.
```

5 Générer des grilles

10. Écrire des fonctions `sat_random_sudoku(n)` et `dlx_random_sudoku(n)` qui fabriquent une grille aléatoire de sudoku (avec une unique solution).
11. Fabriquer des grilles avec $n = 6$ et comparer les temps de résolution des différents solveurs.

6 Comparaison de performances

En utilisant *commande magique* `%time`, on peut connaître le temps d'exécution d'une commande [4].

```
sage: %time sat_encode(1, 2, 3, 4)
CPU times: user 77 µs, sys: 7 µs, total: 84 µs
Wall time: 69.9 µs
802
```

12. Comparer l'efficacité des fonctions `sat_syudoku_one_solution`, et `milp_sudoku_one_solution` et `dlx_sudoku_one_solution`.

4. Se reporter à la page `http://doc.sagemath.org/html/en/thematic_tutorials/profiling.html` pour plus d'informations sur le profilage de code en SageMath.