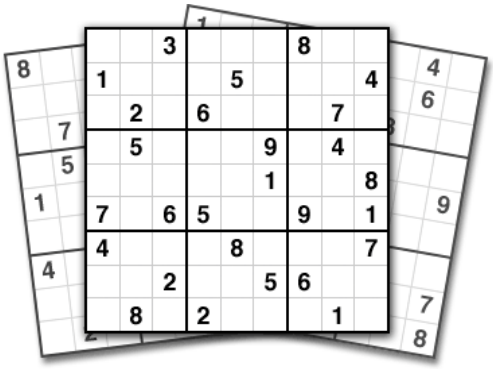

PROJET D'INF431 – SUDOKU

Mehdi KOUHEN et Timothée REBOURS

Sujet proposé par Jean-Pierre Tillich

Ce projet aborde deux points. Tout d'abord, la résolution d'une grille de Sudoku par *backtracking* simple dans un premier temps, et en utilisant l'algorithme X de Knuth par couverture exacte dans un second temps. Puis la génération de grilles de Sudoku de difficultés variables

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

1 Table des matières

2 Introduction

Ce projet a pour objectif de traiter de plusieurs manières la résolution d'une grille de Sudoku générique d'ordre n, c'est-à-dire de taille $n^2 \times n^2$, où n est la taille d'une sous-grille et le nombre de sous-grilles par côté. Le but du sudoku est, à partir d'une grille partiellement remplie, de compléter les autres cases en respectant les contraintes suivantes :

- Chaque ligne et colonne contient exactement une fois tous les nombres de 1 à n^2,
- Chaque bloc carré contient lui aussi exactement une fois tous les nombres de 1 à n^2.

Généralement, on débute d'une grille qui n'a qu'une seule solution possible. À titre d'information, on ne peut avoir une grille à solution unique si elle ne contient pas 17 entrées ou plus au départ. On appelle d'autre part grille minimale une grille ne possédant qu'une solution et qui, si on supprime n'importe quelle case, perd cette unicité.

Voici un exemple de grille minimale, et de sa solution pour $n = 3$:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Pour résoudre le Sudoku, nous étudierons tout d'abord un algorithme de *backtracking* simple qui est de complexité temporelle $n.e^n$.

Nous emploierons ensuite l'algorithme X de Knuth appliqué à la résolution de Sudoku à l'aide de listes doublement chaînées et la technique des *dancing links*.

Ensuite, nous développerons un programme permettant de générer des grilles valides de Sudoku.

Pour finir nous nous emploierons à générer des grilles minimales en combinant les deux programmes précédents.

3 Backtracking simple

Cet algorithme est le plus naïf possible outre le brute-force sans vérification à chaque fois qu'une case est remplie.

Il s'agit d'un appel récursif de la méthode `solve(int[][] matrix, int x, int y)` qui *résout* une case donnée, où matrix est un tableau représentant le Sudoku d'ordre n en cours de résolution, (x,y) sont les coordonnées de la case à remplir.

Tout d'abord on choisit les coordonnées de la case suivante en allant de gauche à droite et de haut en bas. Si on dépasse la taille cela retourne `False`. Si on choisit une case déjà remplie, on la *résout*.

Pour *résoudre* une case vide, on teste successivement tous les entiers entre 1 et n^2 dans cette case en appelant récursivement la méthode `solve` sur la case suivante pour la matrice ainsi modifiée. Si l'une d'entre elle passe tous les tests, on retourne `True`, sinon on remet la valeur de la case à 0 et on retourne `False`.

4 Résolution par un problème de couverture exacte

4.1 Un problème de couverture exacte

Un problème de couverture exacte peut être formalisé de deux manières différentes. La première employée est formalisée par des ensembles :

On considère un ensemble E, un ensemble S constitué de sous-ensembles de E. Le but est de trouver un sous-ensemble S^* de S contenant tous les éléments de E une et une seule fois qui constituerait une *couverture exacte de E* par définition. Cette formalisation est plus qu'inconfortable pour du code informatique, mais permet par la

suite de mieux comprendre la conversion du problème de Sudoku à un problème de couverture exacte.

La seconde manière est formalisée par des matrices binaires. Chaque colonne représente un élément de E. Les lignes correspondent à chaque élément de S. Pour une ligne donnée c'est-à-dire un élément s de S, on met 1 dans les colonnes correspondant aux éléments de E contenus dans s et 0 aux autres.

Prenons un ensemble $E = \{1,2,3,4\}$. Un sous-ensemble $S = \{A = \{1,2\}, B = \{1,3\}, C = \{2,4\}\}$. La matrice binaire correspondante est :

$$\begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

Pour cet exemple, il existe une et une seule solution, triviale, au problème de couverture exacte:

$$\begin{array}{c} \\ B \\ C \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

4.2 L'algorithme X de Knuth

L'algorithme X de Knuth est un algorithme récursif non déterministe permettant de trouver efficacement des solutions à ce problème NP-complet qu'est celui de la couverture exacte.

On peut l'écrire en pseudo-code de la manière suivante pour une matrice A donnée, et on récupère la solution dans S :

```
1  Tant que la matrice A n'est pas vide faire
2      Choisir la première colonne C contenant un minimum de 1;
// Déterministe
3      Choisir une ligne L telle que A[L,C] = 1; //Non
déterministe
4      On ajoute la ligne L à la solution partielle S;
5      Pour chaque colonne J telle que A[L,J] = 1 faire
6            Pour chaque ligne I telle que A[I,J] = 1 faire
```

```
7                   Supprimer la ligne I de la matrice A;
8             Fin
9             Supprimer la colonne J de la matrice A;
10      Fin
11  Fin
12  Afficher S // S étant la solution ainsi construite
```

4.3 La méthode des *dancing links*

4.3.1 Idée de base

Cette méthode est la solution d'implémentation de l'algorithme X proposée par Donald Knuth. Elle est basée sur des listes doublement chaînées circulaires pour lesquelles on fait un constat simple :

```
1  // Ces opérations suppriment le nœud x :
2  x.left.right = x.right ;
3  x.right.left = x.left ;
4  // Celles-ci le restaure à sa position initiale :
5  x.left.right = x ;
6  x.right.left = x ;
```

Ainsi, on ne supprime pas vraiment un nœud, on ne fait que le masquer, et c'est ce principe de base qui permettra par la suite de travailler sur la matrice sans la modifier.

4.3.2 En-têtes de colonne

Une liste doublement chaînée circulaire a une topologie équivalente à celle d'un tore. On dispose pour chaque colonne de cette liste un en-tête comprenant à tout instant le nombre de nœud dans cette colonne afin de diminuer la complexité temporelle lorsque l'on doit récupérer le nombre de nœuds dans une colonne. En plus de tous ces

en-têtes de colonnes, on dispose d'un en-tête de matrice appelée *head* qui contient le lien à droite vers le premier en-tête de colonne de la matrice.

4.3.3 Masquer et démasquer une colonne

On dispose de fonctions `coverCol(int c)` et `uncoverCol(int c)` qui permettent de masquer et démasquer une colonne. Leur fonctionnement est simple : on commence par masquer les nœuds ayant un lien avec la colonne que l'on veut masquer, puis on masque la colonne elle-même. Pour la démasquer on fait simplement l'opération inverse compte-tenu du fait que un nœud garde les informations sur ses voisins après avoir été masqué.

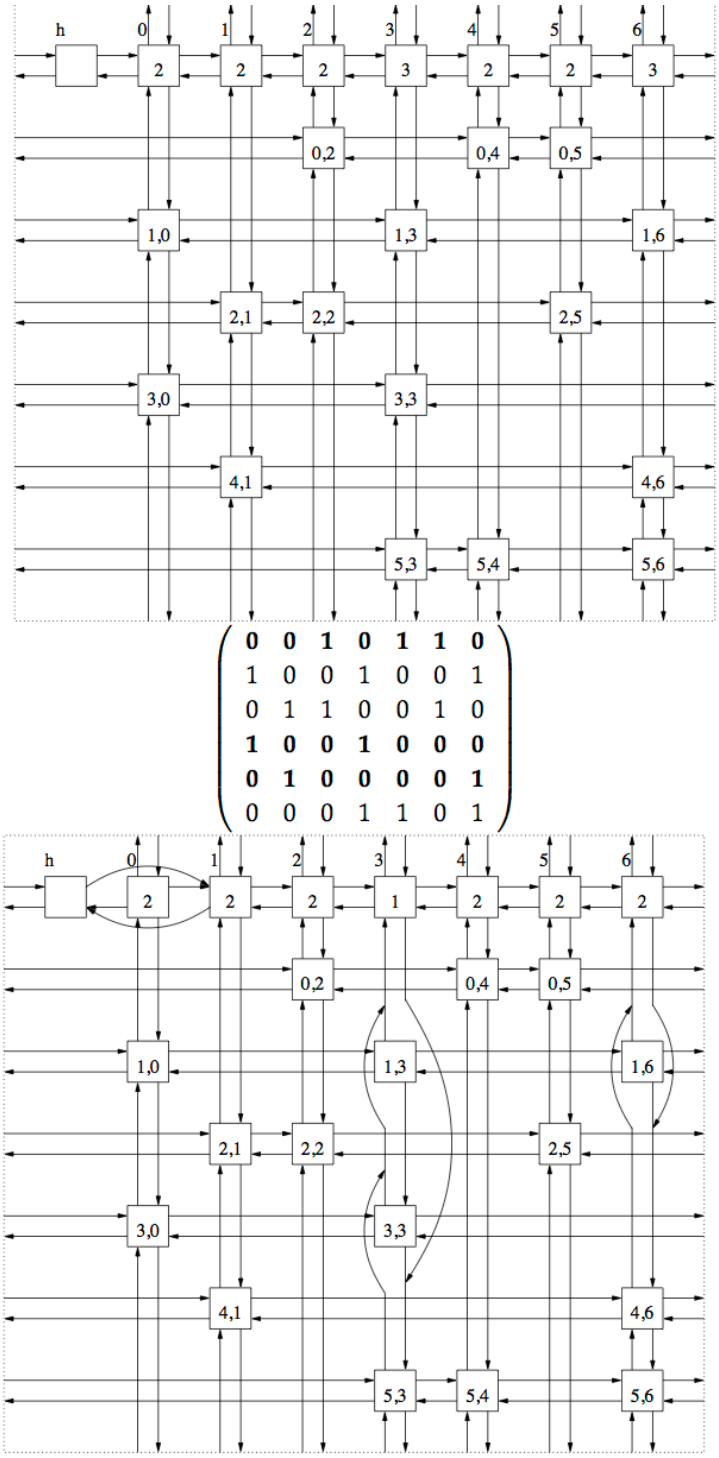

Figure 1 : Illustration de la structure de données et des colonnes masquées

4.3.4 Intérêt

Cette méthode permet d'implémenter l'algorithme X avec une grande efficacité, en diminuant beaucoup sa complexité. En effet, grâce aux entêtes de colonne stockant le

nombre d'éléments, la recherche de colonne ayant le plus petit nombre de 1 et alors de complexité O(n) plutôt que $O(n^2)$.

De plus, la recherche de 1 en ligne ou en colonne est alors de complexité O(1) plutôt que O(n).

5 Génération d'une grille

On veut générer dans un premier temps une grille pleine, pour ensuite enlever aléatoirement des éléments jusqu'à obtenir un Sudoku qui n'est plus minimal. On sait alors que la grille précédente l'était.

On commence par générer une grille vide. On la « résout » ensuite selon une légère variante de la méthode de *backtracking* simple, qui consiste à ne plus choisir les valeurs a tester dans l'ordre, mais aléatoirement.

Une fois que l'on a une grille de Sudoku pleine, on enlève des éléments petit à petit en vérifiant le nombre de solutions (voir la fonction `solveAll()` qui produit une `ArrayList<Stack<Integer>>` dont chaque élément constitue une solution du Sudoku, et est exhaustive). Une fois qu'il dépasse 1, on trouve immédiatement une grille minimale.

6 Utilisation du programme

Le programme est articulé en trois classes : `Cell`, `Grid` et `sudoku`.

`Cell` correspond à la structure de données du nœud dans les *dancing links*, ainsi que les méthodes de génération et de masquage associées.

`Grid` correspond à la structure de matrice binaire ainsi que toutes les méthodes de masquage et de résolution avec l'algorithme X de Knuth associées.

`sudoku` implémente la génération de grilles, le backtracking simple, la vérification de la validité d'une grille, la conversion des matrices binaires aux grilles et la détermination des contraintes.

La fonction main se situe dans Sudoku et a deux modes, génération et résolution :

- pour la génération, les arguments à passer à `sudoku` sont :
  `args[] = ["g",n]` avec n l'ordre du Sudoku souhaité.
- Pour la résolution, les arguments à passer à `sudoku` sont :
  `args[] = ["b" ou "x",n,xyc,…,xyc]` avec `"b"` pour résolution par backtracking simple, `"x"` pour résolution par dancing links, n l'ordre du Sudoku à résoudre, et pour ajouter un chiffre c aux coordonnées `(x,y)` de la grille on doit donner comme argument "`xyc`" .

7 Traces d'exécution

Trace d'exécution d'une résolution avec le backtracking simple :

```
$ java sudoku b 3 014 023 037 069 088 125 143 211 263 306 342
357 404 427 461 483 535 544 589 622 673 745 764 805 824 851
862 876
 ----------------------
|   4 3 | 7     | 9   8 |
|     5 |   3   |       |
|   1   |       | 3     |
 ----------------------
| 6     |   2 7 |       |
| 4   7 |       | 1   3 |
|       | 5 4   |     9 |
 ----------------------
|     2 |       |   3   |
|       |   5   | 4     |
| 5   4 |     1 | 2 6   |
 ----------------------

Solving...

 ----------------------
```

```
| 2 4 3 | 7 1 6 | 9 5 8 |
| 9 8 5 | 2 3 4 | 7 1 6 |
| 7 1 6 | 8 9 5 | 3 4 2 |
 -----------------------
| 6 3 9 | 1 2 7 | 5 8 4 |
| 4 5 7 | 9 6 8 | 1 2 3 |
| 8 2 1 | 5 4 3 | 6 7 9 |
 -----------------------
| 1 6 2 | 4 7 9 | 8 3 5 |
| 3 7 8 | 6 5 2 | 4 9 1 |
| 5 9 4 | 3 8 1 | 2 6 7 |
 -----------------------
Execution time : 76.083ms
```

Trace d'exécution d'une résolution avec l'algorithme X :

```
$ java sudoku x 3 014 023 037 069 088 125 143 211 263 306 342
357 404 427 461 483 535 544 589 622 673 745 764 805 824 851
862 876
 -----------------------
|   4 3 | 7     | 9   8 |
|     5 |   3   |       |
|   1   |       | 3     |
 -----------------------
| 6     |   2 7 |       |
| 4   7 |       | 1   3 |
|       | 5 4   |     9 |
 -----------------------
|     2 |       |   3   |
|       |   5   | 4     |
| 5   4 |     1 | 2 6   |
 -----------------------

Solving...
```

```
 ----------------------
| 2 4 3 | 7 1 6 | 9 5 8 |
| 9 8 5 | 2 3 4 | 7 1 6 |
| 7 1 6 | 8 9 5 | 3 4 2 |
 ----------------------
| 6 3 9 | 1 2 7 | 5 8 4 |
| 4 5 7 | 9 6 8 | 1 2 3 |
| 8 2 1 | 5 4 3 | 6 7 9 |
 ----------------------
| 1 6 2 | 4 7 9 | 8 3 5 |
| 3 7 8 | 6 5 2 | 4 9 1 |
| 5 9 4 | 3 8 1 | 2 6 7 |
 ----------------------
Execution time : 7.085ms
```

Trace d'exécution d'une génération de grille minimale :

```
$ java sudoku g 3
 ----------------------
|       | 2   8 |       |
|     8 |   6   |   4 7 |
| 3 6   | 4 5 7 |   2   |
 ----------------------
|   7 1 |       | 6   5 |
| 5 8 2 |   9 1 | 4 7 3 |
|   9 3 | 5     |       |
 ----------------------
|     4 |   2   | 9   6 |
|       | 9     | 7   4 |
| 9   6 | 7 4 5 | 8     |
 ----------------------
```

```
Solution :


 ----------------------
| 1 4 7 | 2 3 8 | 5 6 9 |
| 2 5 8 | 1 6 9 | 3 4 7 |
| 3 6 9 | 4 5 7 | 1 2 8 |
 ----------------------
| 4 7 1 | 3 8 2 | 6 9 5 |
| 5 8 2 | 6 9 1 | 4 7 3 |
| 6 9 3 | 5 7 4 | 2 8 1 |
 ----------------------
| 7 1 4 | 8 2 3 | 9 5 6 |
| 8 2 5 | 9 1 6 | 7 3 4 |
| 9 3 6 | 7 4 5 | 8 1 2 |
 ----------------------
```