Le Projet
Résolution du Sudoku

Département Biomed — Polytech Marseille — usage interne

Année 2015-16

1 Définition

Une grille de Sudoku est une grille carrée de côté n, où n est le carré d'un entier n_bloc. Elle comporte n lignes et n colonnes sur lesquelles on rajoute un pavage régulier par des blocs carrés de taille $n\_bloc \times n\_bloc$. Ci-contre, le lecteur trouve une grille de Sudoku pour $n = 4$ et $n\_bloc = 2$.

```
************************
*     |     *     |     *
*  1  |  4  *  2  |  3  *
*     |     *     |     *
*-----+-----*-----+-----*
*     |     *     |     *
*  3  |  2  *  4  |  1  *
*     |     *     |     *
************************
*     |     *     |     *
*  4  |  3  *  1  |  2  *
*     |     *     |     *
*-----+-----*-----+-----*
*     |     *     |     *
*  2  |  1  *  3  |  4  *
*     |     *     |     *
************************
```

Toute ligne, toute colonne et tout bloc de la grille comprend n cases. Chaque case de la grille prend une valeur v, entre 1 et n, en respectant la règle suivante. Chaque ligne, chaque colonne et chaque bloc de la grille comporte une et une seule occurrence de chaque valeur v, avec $1 \leq v \leq n$. La grille ci-contre est une grille de Sudoku correctement remplie.

En tant que jeu de réflexion, le Sudoku consiste à considérer une grille partiellement remplie et à la compléter. Ceci peut se faire par la réflexion et/ou par des essais-échecs. Le concepteur de la grille garantit que celle-ci peut être complétée d'une et une seule manière. Le Sudoku a été inventé en 1979 par l'Américain Howard Garns, qui était architecte, et s'inspire du *carré latin* de Leonhard Euler et de jeux similaires.

2 Description globale du projet

Le projet consiste à résoudre par ordinateur des grilles de Sudoku de tailles 4×4, 9×9, 16×16 et 25×25. Il procédera par étapes en implantant progressivement des stratégies de raisonnement de plus en plus sophistiquées. Toutes les stratégies envisagées restent cependant élémentaires, et faciles à implanter. Si le raisonnement à lui seul ne suffit plus, l'ordinateur utilisera le principe du *back-track* pour essayer toutes les possibilités et donc résoudre le problème par ce qu'on appelle *la force brute*. Le lecteur intéressé trouvera à l'adresse `http://www.sudokuwiki.org/sudoku.htm` la liste probablement la plus complète de stratégies de raisonnement applicables au Sudoku. Lorsque ce document présentera les diverses optimisations, il indiquera les noms anglais desdites stratégies d'après ledit site Internet.

Les élèves travailleront en binôme et verront leur enseignant-superviseur quatre fois tout au long du projet. Le rendu consiste en un programme écrit en langage C qui résout le problème proposé. En plus, les élèves présenteront leur projet lors d'une séance de présentation commune à toute la promotion. Cette présentation se fera en anglais, avec un support PowerPoint, et sera une présentation *grand public* de la problématique. Les modalités précises seront communiquées en temps utile.

3 Représentation des données

La représentation des données a clairement une incidence non négligeable sur l'algorithmique et l'aisance avec laquelle on arrivera à décrire le projet. Aussi, nous vous imposons certains choix de représentation des structures. Nous sommes conscients de l'effort que nous vous demandons à intégrer notre modèle de pensée. En retour, nous vous fournissons déjà certaines fonctions qui faciliteront votre programmation. Il n'y aura aucune dérogation à cette règle. Les structures imposées sont fournies sous la forme d'un fichier.

`Max_size_bloc` ainsi que son carré `Max_size` servent aux seules déclarations des structures de données et donnent leur taille maximale. `VALUE` et `COUNT` sont décrites plus loin.

La variable globale `Size_bloc` donne la taille de la grille de Sudoku que nous voulons résoudre, comprise entre 2 et 5. `Size` est un alias qui simplifie l'écriture et qui autorise par exemple un `for ( ... ; i <= Size ; ... )` bien plus compact que `for ( ... ; i <= Size_bloc * Size_bloc ; ... )`

```
#define VALUE 0

#define COUNT ( Size+1 )

#define Max_size_bloc 5

#define Max_size ( Max_size_bloc * Max_size_bloc )

#define Size ( Size_bloc*Size_bloc )

int Size_bloc ;
```

La variable globale principale est `int Sudoku[ Max_size+1 ][ Max_size+1 ][ Max_size+2 ]`. Elle représente la grille de Sodoku. Nous souhaitons énumérer les lignes de 1 à *Size*, de même pour les colonnes. Nous écrirons donc `Sudoku[ li ][ co ]` avec `li` $\in \{1, \ldots, Size\}$ et `co` $\in \{1, \ldots, Size\}$, sachant fort bien que les parties du tableau pour `li` $= 0$ ou `co` $= 0$ ne sont jamais utilisées. Mais, nous sommes prêts à accepter ce "gaspillage" de mémoire en contrepartie d'une expression bien plus simple.

Pour une case donnée, c'est-à-dire une ligne `li` et une colonne `co` donnée, nous avons en fait un vecteur d'informations, qui est la troisième dimension du tableau `Sudoku`. Ainsi, `Sudoku[ li ][ co ][ 0 ]` donne la valeur de la case en question. Une valeur non nulle signifie que la case en question est déjà déterminée. Dans le cas contraire, la case est dite *vide*. Nous écrirons `Sudoku[ li ][ co ][ VALUE ]`, l'alias `VALUE` ne servant qu'à cela. `Sudoku[ li ][ co ][ VALUE ]` sera souvent vue comme un booléen qui indique si la case a déjà une valeur, ou non.

Dans le cas où `Sudoku[ li ][ co ][ VALUE ]` est nulle, il est intéressant de savoir quelles valeurs la case pourrait prendre. Les éléments `Sudoku[ li ][ co ][ v ]`, avec `v` $\in \{1, \ldots, Size\}$, donnent précisément ces informations sous forme de booléens. Finalement, il sera utile de savoir combien de valeurs une case vide pourrait prendre. Cette information sera mémorisée en `Sudoku[ li ][ co ][ Size + 1 ]`, encore écrite `Sudoku[ li ][ co ][ COUNT ]`. Nous avons donc toujours la relation `Sudoku`[`li`][`co`][`COUNT`] $= \Sigma_{v=1}^{Size}$ `Sudoku`[`li`][`co`][`v`].

Illustrons cette représentation l'aide de deux cases de la grille de la page suivante. La case $(5, 8)$ vaut 9, nous aurons `Sudoku[ 5 ][ 8 ][ VALUE ]` qui vaut `9`. Les valeurs de `Sudoku[ 5 ][ 8 ][ i ]`, pour `i` $\in \{1, \ldots, Size + 1\}$, sont quelconques, soit qu'elles n'ont jamais été initialisées, soit qu'elles contiennent d'anciennes valeurs devenues obsolètes. La case $(4, 7)$ appartient au bloc 6. Elle est vide et nous savons que `Sudoku[ 4 ][ 7 ][ VALUE ]` vaut `0`. Comme l'ensemble des cases de la ligne 4, de la colonne 7 ou du bloc 6 utilisent déjà les valeurs $\{1, 2, 3, 4, 6, 7, 9\}$, il ne reste que l'ensemble de valeurs $\{5, 8\}$ que la case $(4, 7)$ pourrait prendre. Ceci serait représenté par le fait que les valeurs de `Sudoku[ 4 ][ 7 ][ i ]`, pour `i` $\in \{1, \ldots, Size\}$, sont la séquence de valeurs $(0, 0, 0, 0, 1, 0, 0, 1, 0)$. Enfin, `Sudoku[ 4 ][ 7 ][ COUNT ]` vaut `2`, car il reste 2 possibilités pour la case $(4, 7)$.

4 Accès optimisé aux données

Le raisonnement appliqué à une case donnée se fait par rapport au fait qu'elle appartient à une certaine ligne, une certaine colonne ou un certain bloc. Il est donc essentiel que les autres cases de la ligne, la colonne ou du bloc puissent être trouvées rapidement. C'est l'objectif des variables `Lines`, `Columns` et `Blocs`. Ainsi, nous pouvons par exemple écrire `calcul( Lines[ 5 ] )` ou `calcul( Blocs[ 8 ] )` pour appliquer la procédure `calcul` aux cases de la ligne 5 ou celles du bloc 8.

```
int * Lines[ Max_size+1 ][ Max_size+1 ] ;

int * Columns[ Max_size+1 ][ Max_size+1 ] ;

int * Blocs[ Max_size+1 ][ Max_size+1 ] ;
```

Analysons la structure de `Blocs[ i ]`, sachant que cette analyse vaut tout autant pour `Lines` et `Columns`. `Blocs[ i ]` donne les cases de la grille Sudoku qui appartiennent au bloc d'indice `i`. En conséquence, `Blocs[ i ] [ j ]` donne les informations relatives à la case d'indice `j` de ce bloc, c'est-à-dire le vecteur d'informations relatives à une case comme décrit au paragraphe précédent.

`Blocs` est une structure tri-dimensionnelle et `Blocs[ i ][ j ][ VALUE ]` donne la *valeur* de la j^e case du i^e bloc. La variable `Sudoku` est une variable tri-dimensionnelle déclarée sous la forme `int Sudoku[ ][ ][ ]` car toutes les cases mémoire sont contiguës. `Blocs` est également tri-dimensionnelle mais est constituée d'un tableau bi-dimensionnel (`Blocs[ i ][ j ]`) de vecteurs "piqués" dans la variable `Sudoku`. Ceci explique sa déclaration sous la forme `int * Blocs[ ][ ]`. Ceci étant, les deux variables s'utilisent de la même façon. Le code de construction des variables `Lines`, `Columns` et `Blocs` vous est fourni.

5 Calcul des valeurs possibles

La première analyse de toute grille de Sudoku consiste à déterminer quelles sont les valeurs possibles pour les cases vides. Le programme commence bien-sûr par cette même analyse. Pour `( li , co )` vide, nous allons donc déterminer les valeurs `v` $\in \{1, \ldots, Size\}$ que la case peut prendre pour remplir le tableau `Sudoku[ li ][ co ][ v ]` en conséquence. Calculer les valeurs possibles revient à les supposer initialement toutes possibles et à enlever par la suite celles qui sont déjà prises par une autre case de la même ligne, la même colonne ou du même bloc. Nous ajusterons bien-sûr `Sudoku[ li ][ co ][ COUNT ]` comme requis.

			◯	8		6	4
	4		1	9	7		
			3			2	
2			9				
4			♡	7		9	6
	8			5	◇	1	3
	4		△		6		
6		□	7	▽	2	4	

La grille ci-dessus vous est fournie dans le fichier `grids.c`, avec quelques entrées en moins, sous le nom `Grid_nine_1`. Le calcul des valeurs possibles montre que la case □ peut prendre l'ensemble des valeurs $\{1, 3, 5, 9\}$. Donc, `Sudoku[ 9 ][ 3 ][ 1 ]` vaut `1`, `Sudoku[ 9 ][ 3 ][ 2 ]` vaut `0`, etc. En conséquence, `Sudoku[ 9 ][ 3 ][ COUNT ]` vaudra 4. La case vide ◇ ne peut prendre que la seule valeur `2`.

6 La fonction de back-track

C'est la fonction principale et sa signature sera `int back_track ( int squares_filled )`. Le résultat rendu est un booléen qui dit, si oui ou non, l'appel qui se termine a trouvé une solution. L'exploration continue tant que ce booléen est faux. Dès qu'il devient vrai, les différentes instances de calcul en cours sont abandonnées (ce n'est pas la peine de chercher une éventuelle seconde solution) et nous ne modifions plus le tableau `Sudoku` pour ne pas perdre la solution trouvée.

Le paramètre de `back_track` indique le nombre de cases qui ont déjà une valeur. Dès que `squares_filled` est égal à `Size * Size`, la grille est complète et il suffit de rendre le booléen 1. Dans le cas contraire, il reste des cases vides. La première chose consiste alors à recalculer, pour toutes les cases vides, les valeurs qu'elles pourraient prendre. Ensuite, nous effectuerons les calculs indiqués ci-dessous.

Nous cherchons la case `( li , co )` vide qui offre le moins de possibilités, c'est-à-dire la case dont la valeur `Sudoku[ li ][ co ][ COUNT ]` est minimale. En cas de cases minimales ex-aequo, nous en prenons une au hasard. Le code pourrait être `count_min = count_of_square_min_possible( & li , & co ) ;` qui rend le nombre de possibilités sous forme de résultat de l'appel et qui positionne, via des pointeurs, les valeurs de `li` et `co` pour indiquer les coordonnées de la case en question.

Si d'aventure `count_min` est nul, nous avons trouvé une case vide pour laquelle il ne reste aucun choix possible. C'est à l'évidence un échec de l'énumération et nous rendons immédiatement le booléen `0`. Cette situation est due à un choix antérieur où le `back_track` a sélectionné au hasard une valeur pour une case qui s'avère maintenant être un mauvais choix. Le booléen retourné avertit d'ailleurs le *fautif* du fait qu'aucune solution n'a été trouvée et celui-ci essayera de choisir une autre possibilité.

A défaut d'être dans le cas précédent, il existe une case vide `( li , co )` pour laquelle il y a `count_min` possibilités, avec $1 \leq$ `count_min` $\leq$ *Size*. Le back-track va à tour de rôle essayer toutes ces possibilités jusqu'à, soit trouver une solution, soit constater que toutes conduisent à un échec.

Ceci signifie que, dans le cas où `Sudoku[ li ][ co ][ v ]` est vrai, nous donnons la valeur `v` à la case en question (`Sudoku[ li ][ co ][ VALUE ] = v ;`) et nous rappelons le back-track sous la forme du code `resultat = back_track( squares_filled+1 )`. Si le booléen `resultat` récupéré vaut `1`, le choix que nous venons d'effectuer est le bon et nous nous contentons de transmettre le booléen positif. Si au contraire `resultat` vaut `0`, le dernier choix est mauvais et il faut l'annuler (`Sudoku[ li ][ co ][ VALUE ] = 0 ;`) pour envisager le choix suivant.

Notons qu'il est crucial pour des raisons d'efficacité de considérer les cases vides en commençant par celles qui offrent le moins de possibilités, de sorte à traiter d'abord les cases qui ne peuvent prendre qu'une seule valeur avant de lancer éventuellement un back-track pour les autres cases.

7 Amélioration du raisonnement

Notre programme élémentaire est suffisant pour résoudre la plupart des grilles de taille 4×4, 9×9 ou encore 16×16, mais capitule devant les grilles 25×25 (http://www.top-sudoku.com/), même simples. En effet, le choix aveugle sous forme `back_track` est trop mis à contribution et les temps de calcul s'envolent.

Nous sommes donc contraints d'améliorer le raisonnement. La fonction `back_track` commencera donc par le calcul naïf des possibilités offertes aux cases vides avant d'enchaîner avec une ou plusieurs étapes de raisonnement en vue de restreindre ces possibilités. Les optimisations se feront en deux étapes.

7.1 Une valeur proposée une seule fois

La stratégie peut être illustrée grâce à la case ◯ de la grille de la page 3. Cette case peut prendre l'ensemble de valeurs $\{2, 5, 7\}$. Par contre, si nous considérons la case comme étant membre de sa colonne, nous voyons qu'elle est la seule à pouvoir proposer la valeur 7. Les autres cases vides de la colonne 5 ne proposent en effet jamais la valeur 7. L'ensemble des propositions faites par ◯ peut donc être réduit au singleton $\{7\}$. Cette stratégie est connue sous le nom de *last remaining cell*.

Comme ce raisonnement nous permet de diminuer l'ensemble des valeurs possibles pour certaines cases, il se peut que le raisonnement ou d'autres raisonnements s'appliquent ensuite ailleurs. Nous pouvons donc éventuellement observer une cascade d'optimisations.

Pour tenir compte de ce phénomène, chaque application d'un raisonnement rend un booléen qui dit si, oui ou non, les valeurs possibles d'au moins une case vide ont pu être réduites. Tant que ce booléen reste vrai, nous itérons les tentatives d'application du ou des raisonnements. Ceci donne lieu à la fonction `optimise` qui vous est imposée.

La mise en œuvre de cette stratégie est contrôlée par le booléen `Optimise_one` et consiste à chercher s'il existe, oui ou non, une case sur la ligne, la colonne ou le bloc qui soit la seule à proposer une certaine valeur. Si tel est le cas, les possibilités de la case en question peuvent être réduites à cette seule proposition.

7.2 Deux cases proposent les deux mêmes valeurs et que celles-là

Lors qu'une case vide ne propose qu'une seule valeur, il est clair qu'elle doit prendre cette valeur. Ce raisonnement s'étend à deux cases. Admettons que, sur une ligne, une colonne ou dans un bloc, les cases c_1 et c_2 aient les mêmes ensembles de valeurs proposées $\{a, b\}$ et qu'elles ne proposent que ces seules deux valeurs. Nous ne savons alors pas encore laquelle des cases va prendre laquelle des deux valeurs, par contre, il est sûr que les autres cases de cette ligne, cette colonne ou ce bloc, ne peuvent pas prendre ni la valeur a, ni la valeur b. Ce raisonnement est aussi connu sous les noms de *naked pair* ou encore *conjugate pair*.

Toujours pour la colonne 5 de la grille page 3, les cases $\triangle$ et $\triangledown$ proposent l'ensemble de valeurs $\{5, 8\}$. Celles-ci sont donc interdites pour les autres cases de la colonne 5. En particulier, la case $\heartsuit$ voit ses possibilités réduites de $\{2, 8\}$ à l'ensemble singleton $\{2\}$ et se retrouve donc déterminée de manière unique.

Le code relatif à cette optimisation est un peu plus complexe, car il s'agit de comparer toutes les cases qui font exactement deux propositions de valeurs pour voir si celles-ci sont identiques. Si tel est le cas, il faut interdire lesdites valeurs pour toutes les autres cases.

7.3 Synthèse

Globalement, les deux raisonnements proposés permettent déjà de limiter considérablement les ensemble de valeurs que les cases vides peuvent prendre. Le tableau ci-dessous indique pour les cases de la colonne 5, les valeurs qu'elles peuvent prendre, avant et après les optimisations.

	$\bigcirc$	$\heartsuit$	$\triangle$	$\triangledown$
Avant	$\{2, 5, 7\}$	$\{2, 8\}$	$\{5, 8\}$	$\{5, 8\}$
Après	$\{7\}$	$\{2\}$	$\{5, 8\}$	$\{5, 8\}$

Les stratégies proposées sont d'ailleurs déjà assez puissantes pour résoudre la grille 25×25 appelée `Grid_twenty_five_1` sans avoir besoin du tout de recourir à la force brute par l'essai systématique d'un ensemble de possibilités. Le seul raisonnement est suffisant.

De nombreuses améliorations pourraient être apportées à ce projet pour améliorer l'efficacité du programme. Bien qu'intéressantes, elles n'apporteraient pas grand chose au niveau de la démarche générale tout en se situant clairement à l'extérieur de l'effort que l'on peut demander aux élèves pour un projet de semestre de première année.

8 Modalités de mise en œuvre du projet

L'énoncé du projet consiste dans le présent document, accompagné d'un fichier de code partiel qui donne les structures de données et fonctions qui vous sont imposées, ainsi que d'une séance de présentation orale organisée par les enseignants et son fichier PowerPoint.

Il faudra définir la fonction `void print_Sudoku ( void )`, puis la fonction `void fill_possibilities ( void )` qui calcule les valeurs possibles pour les cases, sans optimisation, ensuite la fonction `int optimise_possibilities ( int * squares[ ] )` et finalement la fonction d'exploration combinatoire `int back_track ( int squares_filled )`. Les types de ces fonctions ne peuvent en aucun cas être modifiées.

Les entrevues avec les enseignants doivent être préparées, car elles ont une incidence notable sur la note de projet. Elles consistent en des rendez-vous de 15 à 20 minutes pour évaluer ensemble la progression du travail et répondre aux questions qui se posent à vous. Les enseignants répondront à vos questions , mais n'écriront, ni ne corrigeront le code à votre place.

Concernant la programmation, l'accent est mis sur l'algorithmique du projet et les fioritures de mises en page ou de dialogue avec l'utilisateur sont secondaires. Le programme devra savoir lire les grilles telles que fournies dans `grids.c` et les afficher, partielles ou complètes, au format de la grille de la première page. deux booléens contrôlent l'activation ou non des raisonnements.

La notation tient compte pour 10 points du programme final rendu et de l'engagement du binôme pendant les entrevues avec les enseignants. La date de rendu du programme sera annoncée et ne souffrira d'aucun report.

Les autres 10 points notent la présentation *grand public* à la fin du projet, en tenant compte de la démarche et de la construction du discours, mais aussi de la qualité du support PowerPoint. La présentation sera fera en anglais. Sa durée sera de 10 minutes qui seront suivies d'une brève session de questions-réponses, toujours en anglais.

Bon projet !