Rapport Sudoku

Elliott Blot

7 janvier 2016

Table des matières

Introduction

Une grille de Sudoku est une grille carrée de $n \times n$ cases (avec n, le carrée d'un entier). Elle peut être séparée en plusieurs sous-grilles, de n cases chacune, qui sont les colonnes, les lignes, et les blocs.

Pour que la grille soit résolue, il faut que chaque sous-grille contient une et une seule fois chaque chiffre, lettre ou couleur de la grille (généralement pour des grilles 9×9, on utilise l'ensemble {1,2,3,4,5,6,7,8,9}).

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

FIGURE 1 – exemple de grille résolue de taille 9

Le programme permet la résolution d'une grille de sudoku de taille 1, 4, 9, 16, 25, 36, 49, ou 64. Pour cela on utilise des pset_t pour coder chaque case de la grille. Il s'agit d'un entier codé sur 64 bits, chaque bit représentant une couleur (1 si la couleur est possible pour la case, 0 sinon). On obtient donc, pour chaque case, la liste des couleurs possibles dans celle-ci. La grille est résolue lorsque tous les pset_t de la grille sont des singletons (i.e. le pset_t a seulement un bit égal à 1).

L'algorithme de résolution est le suivant : On applique les heuristiques à chaque sous-grille, jusqu'à ce que la grille soit résolue, ou qu'on ne puisse plus rien changer sur celle-ci. Dans ce dernier cas, on utilise alors le Backtracking. On fait un choix arbitraire sur un des pset_t (dans le programme, on choisit la couleur la plus à gauche du pset_t), où il y a le moins de choix possibles, et on essaye à nouveau les heuristiques. Si on obtient une grille irrésoluble après un choix, alors on reprend la grille avant d'avoir fait ce choix, et on en fait un

autre.

Dans le programme, un choix est implémenté par une structure, avec en paramètre la grille avant le choix, les coordonnées du choix, la couleur choisie, et un pointeur vers le choix précédent.

Dans le programme nous permettons aussi la résolution d'hyper-sudoku(option -H ou –hyperblocks). Il s'agit d'une grille de sudoku classique avec la contrainte de $(\sqrt{n}-1)^2$ sous-grilles supplémentaires.

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

FIGURE 2 – exemple de hypersudoku de taille 9

Les parties en bleu de la grille forment les sous-grilles suplémentaires : les *hyperblocks*.

Le programme permet aussi la génération de grille (option -g ou –generate=) de taille variable, avec une solution unique ou non (option -s ou –strict).

Enfin le programme doit pouvoir estimer de la difficulté d'une grille, en estimant la variabilité des choix (option -r ou –rate).

Chapitre 1

Heuristiques supplémentaires

1.1 Heuristiques générales

Les heuristiques sont appliquées sur chaque sous-grille.

Cross-Hatching

Cette heuristique permet de, si il y a un singleton (une couleur seule dans une case) dans une sous-grille, supprimer celui-ci de toutes les autres cases de la sous-grille. Par exemple, si on a {1234, 2, 123, 14} dans une sous-grille, on remarque que le singleton 2 est présent dans la sous-grille, et donc peut être supprimé : {134, 2, 13, 14}

Lone-Number

Cette heuristique permet de, si une couleur ce trouve uniquement dans une case d'une sous grille, fixer la case à cette couleur (c'est forcement la bonne couleur). Par exemple, si on a {123, 2, 123, 124} dans une sous-grille, on remarque que 4 ne se trouve que dans la dernière case donc c'est la bonne couleur : {123, 2, 123, 4}

N-Possible ou Naked-subset

Cette heuristique permet de, si il y a N fois le même ensemble de couleurs de cardinalité N, supprimer cet ensemble de toutes les autres cases de la sous-grille. Par exemple, si on a {2, 456, 3, 456, 45678, 78, 1, 456, 9} dans une sous-grille, on remarque que l'ensemble 456 apparait 3 fois et est de taille 3, donc on peut le retirer des autres pset_t : {2, 456, 3, 456, 78, 78, 1, 456, 9}

J'aurais aussi souhaité implémenter l'heuristique *Hidden-subset*, mais je n'ai pas réussi à l'inclure dans le programme, ne sachant pas comment l'implémenter à partir des pset_t et des sous-grille.

1.2 Hyper-Sudoku : Hidden-Blocks

Après une recherche sans trouver d'heuristiques j'ai décidé d'implémenter la seule aide à la résolution des hyper-sudokus que j'ai réussi à trouver : les hidden-blocks.

Il ne s'agit pas réellement d'une heuristiques, mais l'implémentation de nouvelles sous-grilles (hidden-blocks), qui sont présent dans la grille à cause des contraintes sur les *hyper-blocks*.Cela permet de résoudre les hypersudokus plus rapidement, en appliquant les heuristiques communes à toutes les grilles sur ces hidden-blocks, et en permettant de repérer plus rapidement si la grille est consistante, ou pas, lors du backtraking, si un hidden-blocks est inconsistant(i.e. les règles du sudoku ne sont pas respectées dans la sous-grille).

Il y a $2(\sqrt{n}-1)+1$ hidden-blocks dans la grille. On peut distinguer 3 types d'hidden blocks : *les hidden-blocks en lignes, les hidden-blocks en colonnes*, et enfin *le hidden-block diagonal*.

Les *hidden-blocks en lignes* sont les cases de la grille qui sont sur la même ligne qu'une case d'un hyperblock, mais qui ne sont pas sur la même colonne. Il y a un *hidden-blocks en lignes* par lignes d'hyperblocks,donc $\sqrt{n}-1$ *hidden-blocks en ligne*.

Les *hidden-blocks en colonnes* sont les cases de la grille qui sont sur la même colonne qu'une case d'un hyperblock, mais qui ne sont pas sur la même ligne. Il y a un *hidden-blocks en colonnes* par colonnes d'hyperblocks donc $\sqrt{n}-1$ *hidden-blocks en colonnes*.

le *hidden-blocks diagonal* est l'ensemble des cases de la grille qui ne sont, ni sur la même ligne qu'une case d'un hyperblocks, ni sur la même colonne. On peut aussi les voir comme l'ensemble des cases qui ne sont ni dans un hyperblock, ni dans un autre *hidden-block*. Le *hidden-block diagonal* est unique dans la grille.

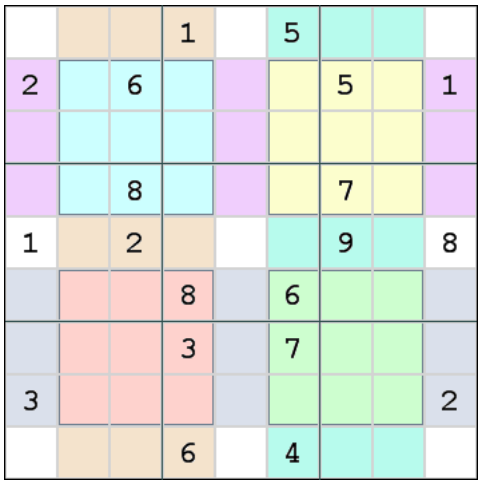

FIGURE 1.1 – les parties colorées représentent les hyper-blocks et hidden-blocks

Dans la figure ci-dessus on a donc les quatre hyperblocks, de couleur bleu clair, rouge, jaune et vert, puis cinq *hidden-blocks*. Les deux *hidden-blocks en ligne* sont en rose et gris, et les deux *hidden-blocks en colonnes* sont en orange et bleu turquoise. les cases blanches restantes forment ensuite le dernier *hidden-block* : le *hidden-block en diagonal*.

Les hidden-blocks ne sont parcouru que si l'option -H (ou –hyperblocks)- est utilisée.

sources :
`http://sudopedia.enjoysudoku.com/Hypersudoku.html`

Chapitre 2

Génération de grilles

2.1 Génération de la solution

Pour commencer, on crée un bloc de la taille correspondante, contenant chaque couleur une seule fois, qu'on mélange de manière aléatoire afin d'obtenir la base de notre grille (seed). Les autres blocs sont construits en fonction de la seed. Dans le programme, je place la seed en première ligne de la grille, et je construis les autres lignes en fonction de celle-ci. Chaque ligne correspond au décalé de $\sqrt{n}$ cases à droite de la ligne précédente, et de $\sqrt{n}+1$ si le numero de la ligne et un multiple de $\sqrt{n}$,c'est à dire si on change de blocs (les lignes sont numérotées de 0 à $n-1$).

Par exemple si la seed est :

1	2	3	4	5	6	7	8	9

Alors la grille produite est :

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

Une fois la grille completée, je fais des permutations de lignes et de colonnes (seulement si leurs numéros sont dans le même bloc, donc leurs numéros doivent être dans le même ensemble $[k\sqrt{n}, (k+1)\sqrt{n}]$, avec k un entier entre 0 et $\sqrt{n}-1$, pour garder la grille consistante) afin de la rendre plus aléatoire.

La grille obtenue servira de solution à la grille générée par la suite.

2.2 Suppression de couleurs aléatoirement

Afin de garder une symétrie par rapport au centre de la grille, à chaque fois que la case (i, j) est vidée, on vide aussi la case $(n-1-i, n-1-j)$ (avec i et j de 0 à $n-1$).

Dans le programme, je choisis de manière aléatoire un couple (i, j), et je vide les cases associées de la grille (aprés avoir sauvegardé les pset_t dans des variables). Si l'option -s a été utilisée (option permettant que la grille n'est qu'une seule solution), alors on vérifie si le programme est capable de résoudre la grille, et si la solution trouvée et égale à la solution générée dans la partie précédente. Dans ce cas on recommence l'opération avec la nouvelle grille. Sinon on replace les pset_t dans leur case et on essaye avec un couple différent. On fait l'opération un certain nombre de fois, pour obtenir la grille finale.

Afin que la génération avec solution unique soit plus rapide pour des grandes tailles (49 ou 64), j'ai fais en sorte que la résolution se fasse en moins de n^2 étapes (uniquement si on essaye de génère une grille), même si cela réduit la difficulté de celle-ci (surtout pour les grilles de grande taille).

Cette méthode de génération permet de générer des grilles très rapidement (si on à pas utilisé l'option -s), même pour une grille de grande taille, car elle ne dépend pas du solveur (qui est lent pour mon cas). Cependant elle ne permet pas la génération d'hypersudoku, et la génération de grille à solution unique reste lente.

source :
`http://www.mathspace.com/comap/Training_Materials/Team2975_ProblemB.pdf`

Chapitre 3

Evaluation de la difficulté des grilles

Après lecture de l'article (*"The Chaos Within Sudoku"*) j'ai compris que la difficulté d'une grille pouvait être calculée a partir de la transformation du problème de sudoku a celui de k-SAT, en utilisant le nombre de variables et de clauses du problème k-SAT. Cependant je n'ai pas réussi à comprendre comment obtenir ces valeurs, et donc en faire un algorithme pouvant être utilisé par le programme.

j'ai donc essayé de calculer la difficulté d'une grille différemment, sur une échelle de 1 à 10, à partir de la variabilité des choix pour chaque case. Pour cela, je commence par compter le nombre de choix possibles sur toute la grille, après le premier passage des heuristiques, et juste avant de faire le premier choix (si la grille n'a pas été résolue). On obtient une valeur notée M.On calcule ensuite le nombre de cases où on peut faire un choix(celles où il y a plusieurs couleurs possibles), multiplié par la taille maximum du pset_t (qui est égal à la taille de la grille) noté N. On a donc $0 \leq \frac{M}{N} \leq 1$.

Je multiplie ensuite ce nombre obtenu par 10, ce qui permet de le placer sur une échelle de 1 a 10. Enfin, si la difficulté est égale à 0 alors je décide de la passer à 1 (cela signifie que aucun ou très peu de choix sont à faire pendant la résolution, et donc que la difficulté est minimale).

Avec cette méthode, une grille est considérée difficile si les choix ont, en moyenne, une plus faible probabilité d'être les bons, et me permet de définir la difficulté de la grille en fonction de sa taille (chaque taille de grille à son échelon de difficulté entre 1 et 10, autrement dit le programme peut considérer une grille de taille 4 plus difficile qu'une grille de taille 64).

Conclusion

Je suis satisfait de ce projet, j'ai réussi à faire un programme qui fonctionne même s'il n'est pas optimal.

J'ai eu de la difficulté à trouver la bonne complexité des algorithmes, pour accélérer le programme, et permettre de résoudre les grilles en un temps acceptable (les grilles classiques de taille 9 était résolu en plusieurs minutes au début, et je n'était pas assez patient pour tester les grilles de taille 16) .

Ce projet m'a permit de m'améliorer avec le langage C, principalement au niveau des structures et de la gestion de mémoire, ainsi que beaucoup d'autres chose, ayant peu d'expérience avec ce langage avant ce projet. J'ai appris à utiliser les debuggers comme *gdb* ou *Valgrind*, principalement pour vérifier s'il n y a pas de fuites mémoires dans le programme. J'ai du être plus rigoureux au niveau de la syntaxe, lors de la réalisation de mon programme, mes précédents projets sont chaotique par rapport à celui-ci.

Le projet m'a aussi permis de me familiariser avec le langage LaTeX, que je n'avais jamais eu l'occasion d'essayer avant ce projet.