Projet de programmation

Solveur et générateur de Sudoku

Jonas Lejeune

Master CSI
Université de Bordeaux 1

28 novembre 2012

Table des matières

Introduction

Le sudoku est un jeu en forme de grille inspiré du carré latin et du problème des 30 officiers. Le but est de remplir une grille avec un alphabet de symboles de sorte que chaque symbole n'apparaisse qu'une et une seule fois dans chacune des lignes, colonnes et blocs (appelés sous-grilles) de la grille. Des symboles sont déjà disposés dans la grille afin de pouvoir la résoudre progressivement jusqu'à ce qu'elle soit totalement pleine.

L'engouement pour ce jeu a suscité un intérêt d'analyse mathématique pour ce genre de problèmes. La recherche concernant la complexité des algorithmes ont permis de déterminer que la résolution d'une grille de sudoku était un problème NP-complet.
(sources : Wikipédia, articles Sudoku et Mathématiques du Sudoku)

I - Principe du solveur et mise en œuvre

Le but du projet est de fournir un solveur et un générateur de grilles de sudoku de diverses tailles (les carrés de 1 à 64) en temps résonnable. Le caractère NP-complet de ce problème fait qu'il n'est pas possible (ou tout du moins que l'on ne sait pas si il est possible) d'avoir un algorithme déterministe efficace pour le résoudre. De ce fait, la solution la plus efficace est d'appliquer des contraintes (ou heuristiques) sur la grille afin de réduire au maximum sa complexité puis d'effectuer une recherche exhaustive de solution.

Afin de pouvoir résoudre ce problème, il a été décidé d'utiliser des ensembles préemptifs pour modéliser les cases d'une grille. Un ensemble préemptif est un Type Abstrait de Données qui permet de définir un sous-ensemble de valeurs d'un ensemble fini de valeurs (appellées aussi couleurs). Ces ensembles préemptifs ont des implémentations qui permettent d'effectuer rapidement des opérations ensemblistes dessus (comme l'union, l'intersection, ...). Leur implémentation a été faite à l'aide d'entiers non-signés sur 64 bits (vu comme un vecteur de 64 bits). Chaque bit va représenter la présence (bit mis à 1) ou l'absence (bit mis à 0) d'une couleur (codée par son indice) dans l'ensemble préemptif. On utilise des algorithmes SWAR (SIMD Within A Processor) permettant d'effectuer les opérations ensemblistes quasiment uniquement à l'aide des instructions processeur, et de ce fait les calculs sont effectués en parallèle sur les éléments de l'ensemble.

Chaque case d'une grille de sudoku se voit donc attribuer au début, soit un ensemble préemptif ayant toutes les couleurs possibles si la case est vide, soit un singleton ayant la couleur de la case. On applique ensuite des contraintes sur toutes les sous-grilles pour réduire la cardinalité des ensembles.

Voici un exemple d'une grille de sudoku (à gauche) et sa représentation avec les ensembles préemptifs (à droite) :

1	_	_	**3**	**1**	1234	1234	**3**
_	**2**	_	_	1234	**2**	1324	1324
4	_	**3**	_	**4**	1324	**3**	1234
_	_	_	**4**	1234	1234	1234	**4**

Nous avons implémenté à la base deux heuristiques :

Le cross-hatching consiste pour une sous-grille (ligne, colonne ou bloc) à supprimer dans les cases non-résolues toutes les couleurs des cases déjà résolues (pour assurer leur unicité).

Le lone-number identifie les couleurs apparaissant une et une seule fois dans une sous-grille. De ce fait, la case possédant l'unique couleur aura nécessairement cette valeur (qui ne peut être nulle part ailleurs).

Le principe de résolution est d'appliquer ces contraintes sur toutes les sous-grilles jusqu'à ce que la grille ne soit plus modifiée (moment appelé point fixe). À ce moment là, on applique la recherche exhaustive en utlisant un algorithme de backtracking. Cela consiste à prendre une valeur au hasard d'une des cases non-résolues pour réduire la compléxité de la grille puis de réitérer l'algorithme de résolution sur la nouvelle grille. Si cette grille a une solution, alors on a trouvé une solution pour la grille. Dans le cas contraire, on annule le choix et on en essaye un autre. Si à la fin, aucune solution n'a été trouvée, cela veut dire qu'aucune valeur ne peut être appliquée sur cette case et qu'il n'existe donc pas de solutions à la grille.

II - Mise en place d'autres heuristiques

La résolution par heuristiques utilise déjà deux heuristiques qui ont été décrites précédemment. Ces deux heuristiques nous ont été demandées lors des devoirs. Pour réduire d'autant plus la complexité d'une grille, il existe cependant d'autres contraintes applicables sur une grille. J'ai choisi d'implémenter une heuristique supplémentaire appelée N-possible.

La raison de ce choix est que cette heuristique peut être mise en place facilement dans le cas de notre solveur car elle n'agit que sur une sous-grille. D'autres heuristiques ont en effet besoin d'avoir aussi des sous-grilles partageant une même case pour être appliquées, ce qui aurait impliqué des gros changements dans le code du solveur.

L'heuristique N-possible est en fait le cas général de l'heuristique cross-hatching. Son principe est le suivant : pour toutes les cases d'une sous-grille, si un certain ensemble apparait autant de fois que la cardinalité de cet ensemble, alors toutes le scases ayant un ensemble différent ne peuvent avoir les couleurs de cet ensemble. Le cross-hatching est pour un singleton, qui ne doit apparaître qu'une seule fois dans la sous-grille.
En d'autres termes, voici une définition mathématique de cette heuristique :
$\forall case\ c \in subgrid,$
$(\#c = \#\{case\ c' \in subgrid, c = c'\}) \Rightarrow ((\forall c" \in subgrid, c" \neq c) \Rightarrow c" := c)$

Voici un exemple mettant en œuvre l'heuristique N-possible sur une sous-grille de taille 4 :

123	14	14	1234

En comptant les cardinalités de tous les ensembles et leur fréquence d'apparition, on obtient le tableau suivant :

Ensemble	Cardinalité	Fréquence d'apparition
123	3	1
14	2	2
1234	4	1

On observe que l'ensemble 14 à une cardinalité égale à sa fréquence. De ce fait, on peut soustraire les couleurs 1 et 4 à tous les autres ensembles.

123	14	14	1234

On obtient ensuite la sous-grille suivante :

23	14	14	23

On constate en effet que pour cette sous-grille, si on choisit la valeur 2 pour la 2^{eme} case, cela affecte automatiquement la valeur 3 à la 3^{eme} case (car la valeur 2 ne peux plus apparaître dans aucunes des cases). Pour la même raison, on peut enlever la valeur 3 de toutes les cases. Il en résulte que pour toutes les cases qui ne possèdent pas le sous-ensemble $\{2, 3\}$, on leur a soustrait les couleurs 2 et 3. On trouve le même résultat si on avait inversé les choix pour la 2^{eme} case.

III - Problème de génération de grilles

Le but est de pouvoir générer des grilles de la même taille que celles qui peuvent être résolues par le programme. Il doit aussi être possible de demander la génération de grilles ayant une et une seule solution (mode strict pour le programme).

Pour mettre en œuvre la génération de grilles, j'ai choisi de partir d'une grille déjà résolue puis de lui attribuer aléatoirement des cases vides jusqu'à un certain seuil. Une autre possibilité aurait pu consister, au contraire, à partir d'une grille vide puis de la résoudre progressivement jusqu'à un certain seuil de cases remplies.

La différence entre la génération de grilles à solution unique et celles à multiples solutions est que dans le mode de solution unique, il faut s'assurer à chaque case enlevée que la grille possède toujours une solution unique. De ce fait, la fonction de solveur a été modifiée pour ne plus chercher une solution mais plusieurs. La valeur qu'elle retourne ne sera plus un booléen mais le nombre de solutions trouvées et la grille passée en paramètre sera remplie avec l'une des solutions. Le code qui avait déjà été produit dans les devoirs précédent n'a de ce fait été que très peu modifié (il faut à présent vérifier que le nombre de solutions est strictement positif pour savoir si la grille possède une solution, dans le cas contraire, la grille est inconsistante). Pour la fonction de résolution, la modification s'est faite sur les conditions d'arrêts du backtracking. Alors qu'avant la fonction retournait lorsqu'un des choix était correct, on va à présent tester tous les autres choix.

Pour générer des grilles (quelque soit le mode), une première étape d'initialisation est nécessaire qui fonctionne de cette manière :

- création d'un tableau bidimensionnel checked_cells de la même taille que la grille, initialisé partout à false
- allocation d'une grille remplie avec l'ensemble préemptif maximum
- génération d'une grille entièrement résolue aléatoirement
- calcul du pourcentage de cases à retirer

Le tableau bidimensionnel sert à indiquer les cases qui ont déjà été traitées. Bien qu'il serait possible de s'en passer pour la génération de grilles sans le mode strict, il permet de ne pas avoir de code dupliqué entre les deux modes et éviter ainsi les sources de bugs lors de la modification du code. Pour le mode strict, le tableau est nécessaire car si le retrait d'une case entraîne une augmentation du nombre de solutions, il faut indiquer qu'il ne sert à rien de réessayer plus tard avec cette même case (on aboutirait de nouveau à une grille à multiples solutions, ce qui ferait perdre du temps de calcul).

La génération d'une grille pleine aléatoire a également demandé une modification du solveur. La version originale du solveur appliquait les heuristiques, puis dans le cas où la grille n'était pas résolue, appliquait le backtracking. Le backtracking, quant à lui, choisissait une case de la grille ayant la cardinalité la plus petite et lui affectait la couleur la plus petite de son ensemble préemptif. De ce fait, les heuristiques et le backtracking sont totalement déterministes. La seule modification possible pour faire intervenir de l'aléatoire se fait lors du choix de la couleur à essayer pour le backtracking. Pour cela, on tire un nombre aléatoire (appelons-le n) entre 1 et la cardinalité de la case choisie puis on récupère la couleur située à la n^{eme} position dans cet ensemble. Les choix du backtracking sont rendus alors aléatoires, ce qui conduit à une solution retournée elle aussi aléatoire.

L'étape suivante consiste à calculer le pourcentage de cases à retirer dans la grille. En effet, selon la taille de la grille et le mode, il faudra plus ou moins de temps de calcul pour générer la grille. Ces valeurs ont été choisies arbitrairement afin d'avoir un compromis entre complexité de la grille générée et temps nécessaire à sa génération.

La seconde partie de l'algorithme va consister à retirer des cases aléatoirement dans la grille jusqu'à ne plus pouvoir faire de choix où lorsque l'on satisfait le nombre de cases à retirer. Voici le code :

```
while((removed_cells < remove_limit) && still_choices)
{
    int x = rand() % grid_size;
    int y = rand() % grid_size;
    while(checked_cells[x][y])
    {
        x = (x + 1) % grid_size;
        if(x == 0)
            y = (y + 1) % grid_size;
    }

    if(strict)
    {
        pset_t** result_tmp = grid_copy(result);
        result_tmp[x][y] = pset_full(grid_size);

        if(grid_solver(result_tmp) == 1)
        {
            result[x][y] = pset_full(grid_size);
            ++removed_cells;
        }
        grid_free(result_tmp);
    }
    else
    {
        result[x][y] = pset_full(grid_size);
        ++removed_cells;
    }
    checked_cells[x][y] = true;
```

```
    for(int i = 0 ; i < grid_size ; ++i)
        for(int j = 0 ; j < grid_size ; ++j)
            still_choices = still_choices
                || !(checked_cells[i][j]);
}

return result;
```

Le choix de la case à retirer se fait en deux étapes. La première consiste à choisir aléatoirement une case quelconque dans la grille. Cependant, il se peut que la case ait déjà été retirée ou que son retrait entraîne la multiplication du nombre de solutions. De ce fait, on vérifique ce choix n'a jamais été testé. Si c'est le cas, la solution naïve serait de choisir à nouveaux des coordonnées aléatoires jusqu'à ce que l'on tombe sur un choix de case valide. Cependant, il y a une probabilité (certe, très faible) de ne tirer que des cases déjà traitées et donc de rentrer dans une boucle infinie. La solution qui a été choisie à la place permet de garantir qu'un tel cas n'arrive pas. En effet, à partir des coordonnées aléatoires, tant que la case correspondante a été vérifiée, on se déplace à la case voisine (celle directement à droite ou la première de la ligne suivante dans le cas des cases aux extrémités droites de la grille). De ce fait, on parcourt totalement la grille et on est obligé de tomber sur un candidat valide car à chaque tour de boucle, on s'assure qu'il reste des choix.

C'est à partir de ce moment qu'une distinction se fait entre les deux modes de génération.

Pour le mode non-strict, il suffit de retirer la couleur de la case. La grille conserve sa consistance et il se peut que ce retrait ait conduit à la multiplication des solutions mais on ne s'en préoccupe pas dans ce mode. On incrémente ensuite le nombre de cases enlevées et on indique que la case a été vérifiée. On peut constater aussi que ce mode de génération est borné sur le nombre d'itérations effectuées.

Pour le mode strict, il faut vérifier que la suppression de la couleur de la case n'augmente pas le nombre de solutions. Pour cela, on créé une copie de la grille actuelle sur laquelle on retire la case choisie, puis on tente de résoudre la grille. Si le nombre de solutions est supérieur à 1, ce choix n'était pas bon et on n'applique pas de modifications. On indique tout de même que la case a été vérifiée afin de ne pas retenter ce choix.

Si le nombre de solutions est exactement 1, alors le choix de case est valide et on applique le changement à la grille en cours de génération (on ne lui attribue pas la valeur de la grille temporaire car cette grille a été résolue et la case est donc un singleton). On incrémente ensuite le nombre de cases retirées et on indique que la case a été vérifiée.

Pour les deux modes, on itère ce processus jusqu'à satisfaire nos contraintes puis on retourne la grille générée comme grille finale. Il faut ensuite afficher la grille au format qui est lu par le parseur de grilles.

Remarque La fonction d'affichage a demandé un case particulier pour les grilles de taille 1 car son fonctionnement de base est le suivant :

- si la case est un singleton, on affiche cette couleur
- sinon, on affiche le caractère '_'

Le problèmes est que le seul singleton d'une grille de taille 1 est aussi l'ensemble contenant toutes les couleurs possibles. De ce fait, une grille de taille 1 générée (qui est forcement la grille vide) était affichée comme une grille résolue. Du fait de l'unicité de la génération, un cas particulier est fait pour l'affichage lors de la génération d'une grille de taille 1 et affiche la grille « en dur »(la grille est tout de même générée, seul l'affichage possède cette exception).

IV - Implémentation du code

1. Implémentation et optimisations du code

L'implémentation du sudoku a été quasiment totalement détaillée dans les parties précédentes et suivent le cahier des charges imposé lors des différents devoirs sur ce projet. Le choix majeur qui a été fait pour l'implémentation a été d'utiliser une fonction récursive pour l'algorithme de backtracking. L'alternative aurait été de le faire de façon itérative avec une structure de pile pour parcourir les choix effectués. Ma décision d'implémenter le backtracking de façon récursive vient du fait que l'algorithme est par nature récursif et qu'il est alors plus intuitif de le coder de cette façon. De plus, la pile d'appel de fonction fait office de pile des choix et on évite donc d'avoir à implémenter sa propre structure, facteur possible de bugs supplémentaires. Cependant, le désavantage de l'utilisation d'une fonction récursive est que le calcul peut être plus long que si il était itératif (création d'un contexte de fonction, copie des paramètres) et qu'il est possible d'avoir un dépassement de pile sur de très grandes tailles de grilles (plus grandes que celles traitées dans ce sujet).

L'optimisation majeure qui a été faite se trouve dans le solveur de grilles. Le solveur peut en effet avoir 2 cas d'utilisation :

- résoudre une grille donnée par l'utilisateur
- résoudre une grille en cours de génération lorsque l'on demande une solution stricte

Pour le premier cas, il suffit de retourner la première solution trouvée (c'est ce qui se faisait avant l'implémentation du générateur). Le second cas en revanche nécessite théoriquement de balayer toutes les possibilités du backtracking pour s'assurer de l'unicité de la solution. À la base, le solveur avait été modifié pour ne plus retourner losqu'une solution avait été trouvée mais simplement d'ajouter la solution à une liste chainée contenant toutes les solutions. Il suffisait alors de regarder la taille de la liste pour connaître le nombre de solutions. Mais de ce fait, le temps de calcul est largement augmenté.

L'optimisation consiste à dire que si l'on se trouve dans le mode de génération stricte (les modes sont définis dans des variables booléennes globales), alors il suffit d'arrêter le backtracking si une grille à plus de 1 solution (lorsque 2 solutions sont donc trouvées). Pour cela, la fonction ne va plus retourner un booléen indiquant si la grille est résolue mais le nombre de solutions trouvées. On va additionner ces valeurs lors des choix de couleurs du backtracking et si le nombre de solutions dépasse 1, il n'est plus nécessaire de continuer le calcul.

Cette optimisaiton permet de minimiser le temps de calcul uniquement sur les grilles à multiples solutions. Si la grille n'a qu'une solution, il faut tout de même vérifier tous les autres choix.

2. Correction, robustesse et performance du code

Les outils qui ont été utilisés pour s'assurer que le code était suffisament robuste ont été principalement GDB et Valgrind. GDB a été utilisé principalement lors des phases de développement pour vérifier le bon fonctionnement de fonctions lorsque de petites parties de programmes étaient bugguées. L'outil Valgrind a surtout été utilisé lors de la fin du développement des devoirs lors de la vérification des différents jeux de tests fournis, afin de s'assurer qu'aucunes fuites mémoires n'étaient présentes sur ces tests. Des tests ont également été fait sur la résolution d'autres grilles (notamment celles générées par le programme) et également lors de la génération de grilles pour les différents modes de fonctionnement et les différentes tailles.

L'outil principale utilisé pour l'analyse des performances du programme a été la commande time, qui, couplé à GDB a permis de voir où le programme passait le plus de temps (bien que cette approche pouvait être faite à l'aide de gprof, il y a eu une étape où le programme était dans une boucle infinie et gprof à besoin de la terminaison du programme pour générer son fichier d'analyse). Cet outil a aussi permis de déterminer les valeurs plus ou moins arbitraires du pourcentage de cases à retirer lors de la génération de grilles (le pourcentage reste tout de même quelque peu aléatoire pour des tailles pas trop grandes).

Conclusion

Ce projet a permis de s'intéresser à un problème populaire qui est la résolutio de grilles de sudoku. On a ainsi pu avoir une approche mathématique du problème et s'intéresser aux méthodes possibles pour y répondre tout en sachant qu'il s'agit d'un problème difficile. De plus, le projet m'a permis d'utiliser des outils de gestion de projet auxquels je n'étais pas nécessairement habitués (SVN, valgrind, ...) sur un projet de taille moyenne. Il a aussi été interessant de voir différents aspects de la programmation (de la création d'un parseur à des techniques fines pour gérer les ensembles préemptifs).