Projet : Sudoku

Notions : Fichiers, structures, récursivité, développement logiciel

1 Introduction

Le sudoku est une grille constituée de n^4 cases, présentées sous la forme d'une matrice carrée de n^2 lignes et n^2 colonnes. Cette grille est également découpée en n^2 régions, qui sont des sous-matrices carrées de dimension $n \times n$.

En partant d'une grille partiellement remplie avec des valeurs comprises entre 1 et n^2 (figure 1 à gauche), il faut compléter le sudoku (figure 1 à droite) en respectant une seule règle : dans chaque ligne, chaque colonne, chaque région, les chiffres de 1 à n^2 apparaissent une fois et une seule. L'objectif de ce projet est de réaliser un logiciel capable de lire une grille incomplète de sudoku dans un fichier, d'en trouver la solution, puis d'afficher le résultat à l'écran.

La taille n des sudokus, qui est habituellement égale à 3, sera ici une variable, car nous allons gérer des sudokus de $2^4, 3^4, 4^4$ ou 5^4 cases.

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

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

FIGURE 1 – un sudoku et sa solution ($n = 3$).

2 Représentation du sudoku

Pour représenter le sudoku en mémoire, une matrice est utile. Il est aussi pratique d'avoir la taille (n) de cette matrice. Nous allons donc utiliser une structure comportant un champ `size` pour stocker la taille, et un champ `data` pour les données. Ce champ `data` sera un pointeur vers une matrice allouée dynamiquement, comme vu en cours. Comme la taille des sudokus est limitée à 5 (soit 25 valeurs), les données sont de type `unsigned char`, permettant de stocker des entiers sur 8 bits. Nous définirons donc un type `grid_t` correspondant à cette description.

```
typedef struct {
  int size;
  unsigned char** data; } grid_t;
```

3 Résolution d'un sudoku

La résolution d'un sudoku peut se faire assez facilement à l'aide d'un programme récursif. Il suffit de remplir la première case vide avec un nombre entre 1 et n^2, puis regarder si ce nombre a le droit d'occuper cette place. Trois cas peuvent ainsi se produire :

1. le nombre peut se trouver à cette position : on passe à la case vide suivante par un appel récursif,

2. le nombre ne peut pas se trouver à cette position : on essaye avec un autre nombre compris entre 1 et n^2, qui n'a pas encore été testé,

3. le nombre ne peut pas se trouver à cette position et tous les nombres entre 1 et n^2 ont été testés en vain : on ne peut pas obtenir de solution, si bien qu'un choix précédent doit être modifié.

L'algorithme correspondant, très simple à mettre en œuvre, est donné par l'Algorithme 1

Algorithm 1 int resolution(grid_t grille, int nbvide)

Entrées: grille, la grille de départ à remplir
Entrées: nbvide, le nombre de cases vides

```
1:  Si nbvide==0 Alors
2:    on a trouvé la solution, retourner GAGNE et fin de la fonction
3:  Sinon
4:    Pour tout les lignes i Faire
5:      Pour tout les colonnes j Faire
6:        Si grille[i][j] est VIDE Alors
7:          Pour tout k = 1 à grille.size Faire
8:            Si la position (i, j) est valide pour la valeur k Alors
9:              //On essaye de mettre la valeur k en position i, j
10:             grille.data[i][j] = k;
11:             //Appel de la fonction de résolution en ayant mis la valeur k dans la case i, j
12:             //On fait progresser la solution avec une case vide en moins
13:             Si resolution(grille,vide-1)==GAGNE Alors
14:               //On a trouvé une solution , on quitte la fonction
15:               return GAGNE;
16:             Sinon
17:               //La valeur k ne donne pas de solution
18:               //On remet la case i, j à vide et on revient au problème avec nbvide cases vides
19:               grille.data[i][j] =VIDE;
20:             Fin Si
21:           Fin Si
22:           //On essaye une autre valeur pour k
23:         Fin Pour
24:         //On a testé toutes les valeurs possibles pour la case i, j et aucune ne convient
25:         //On quitte la fonction afin de modifier le choix précédent effectué pour nbvide + 1
26:         return IMPOSSIBLE;
27:       Fin Si
28:     Fin Pour
29:   Fin Pour
30:   //Aucune solution possible si on arrive ici
31:   return IMPOSSIBLE;
32: Fin Si
```

La constante VIDE a pour valeur 0 (voir le format des fichiers des données) et les constantes IMPOSSIBLE et GAGNE sont à définir selon vos désidératas.

4 Format des fichiers de données

Les fichiers contenant les grilles de sudoku sont des fichiers texte contenant chaque ligne du sudoku. Les cases vides sont marquées par un 0 ou un espace. Les valeurs pour les sudokus de taille supérieure à 3 sont des lettres (exemple figure 2) :

- pour les sudokus de taille 2, les valeurs sont les chiffres de 1 à 4,
- pour les sudokus de taille 3, les valeurs sont les chiffres de 1 à 9,
- pour les sudokus de taille 4, les valeurs sont les lettres de l'alphabet A..P,
- pour les sudokus de taille 5, les valeurs sont les lettres de l'alphabet A..Y,
- les cases vides sont marquées indifféremment par un 0 ou un espace, quelle que soit la dimension du sudoku.

```
1 2 3 4
  1 2
0   4 3
4 3 2 1
```

Fichier contenant un sudoku de taille 2

```
0 0 0 9 0 4 7 0 0
0 0 1 0 0 0 4 3 9
0 6 0 0 1 0 0 0 5
0 0 0 0 0 6 0 5 0
6 2 0 0 0 0 0 1 4
0 8 0 5 0 0 0 0 0
9 0 0 0 2 0 0 4 0
2 1 3 0   0 9 0 0
0 0 8 3 0 9 0 0 0
```

Fichier contenant un sudoku de taille 3

FIGURE 2 – Exemples de fichiers sudoku.

5 Optimisation

L'algorithme 1 permet de trouver facilement la solution des sudokus de taille 2 ou 3. Par contre, il devient sans doute trop long pour les sudokus de taille supérieure.

La raison est bien sûr la complexité de cette recherche. Supposons que l'on ait un sudoku de taille 3 avec 45 cases manquantes. Pour la première case vide, nous testons les 9 possibilités, mais seules quelques unes sont valides : 3 ou 4 en moyenne au début de la résolution, si les cases vides sont bien réparties. Une seule conduit à la solution : dans le pire des cas, on teste d'abord les mauvaises possibilités et donc, il va y avoir 3 ou 4 appels récursifs. En appliquant le même raisonnement pour la suite, on obtient aisément que 4^{45} majore le nombre des possibilités testées par le programme. Bien sûr, c'est un majorant très grossier, car de nombreux essais conduisent rapidement à une situation impossible.

9	0	0	8	0	0	0	0	0
0	0	0	0	0	0	5	0	0
0	0	0	0	0	0	0	0	0
0	2	0	0	1	0	0	0	3
0	1	0	0	0	0	0	6	0
0	0	0	4	0	0	0	7	0
7	0	8	6	0	0	0	0	0
0	0	0	0	3	0	1	0	0
4	0	0	0	0	0	2	0	0

FIGURE 3 – Un des sudokus les plus difficiles.

Pour remplir toute une grille, sans tenir compte des contraintes, on a n^2 possibilités pour chacune des n^4 cases, soit $O((n^2)^{n^4})$ possibilités. Si on considère que un quart des cases sont déjà remplies dans la grille initiale, il ne reste plus que $O(n^{2\times 0,75.n^4})$. La complexité globale varie donc en $O(e^{\alpha n^4.\ln(n)})$ avec $(0 < \alpha < 2)$.

En réalité, le nombre de possibilités testées est bien inférieur à cette quantité. Mais les grilles les plus difficiles (par exemple figure 3) demandent quand même 2 à 3 minutes à l'algorithme, qui examine alors 1 305 263 521 possibilités.

5.1 Optimisation de la recherche

Comme dans toutes les méthodes de backtracking, la première optimisation que l'on peut envisager réside dans l'ordre dans lequel on examine les candidats (les cases (i, j) dans l'algorithme 1).

Plutôt que prendre la première case vide, il est plus intéressant, comme un vrai joueur, de considérer la case vide qui est la plus contrainte. C'est la case qui présente le moins de possibilités de jeu. On voit bien que si une case n'offre qu'une seule possibilité, on gagne un facteur n^2 dans la recherche.

Sur une grille comme celle de la figure 3, cette version n'examine plus que 6661 possibilités et résout le sudoku en 42ms.

L'algorithme correspondant est donné par l'Algorithme 2.

Algorithm 2 int resolution(grid_t grille, int nbvide)

Entrées: grille, la grille de départ à remplir
Entrées: nbvide, le nombre de cases vides

```
1:  Si nbvide==0 Alors
2:    on a trouvé la solution, retourner 1 et fin de la fonction
3:  Sinon
4:    Choisir les meilleurs indices i,j : ceux de la case la plus contrainte (moins de possibilités)
5:    Mettre les m valeurs possibles pour cette case dans un tableau choix
6:    //la case i,j peut prendre une des m valeurs stockées dans choix
7:    Pour tout  k = 1 à m Faire
8:      //On essaye de mettre la valeur k en position i,j
9:      grille.data[i][j] = choix[k]
10:     //Appel de la fonction de résolution en ayant mis la valeur choix[k] dans la case i,j
11:     //On fait progresser la solution avec une case vide en moins
12:     Si resolution(grille,vide-1)==GAGNE Alors
13:       //On a trouvé une solution , on quitte la fonction
14:       return GAGNE;
15:     Sinon
16:       //La valeur choix[k] ne donne pas de solution
17:       //On remet la case i,j à vide, on est revenu au problème avec nbvide cases vides
18:       grille.data[i][j] =VIDE;
19:     Fin Si
20:     //On essaye une autre valeur
21:   Fin Pour
22:   //On a testé toutes les valeurs possibles pour la case i,j et aucune ne convient
23:   //On quitte la fonction afin de modifier le choix précédent effectué pour nbvide + 1
24:   return IMPOSSIBLE;
25: Fin Si
```

5.2 Optimisation des structures

La ligne 8 de l'algorithme 2 teste la validité d'un chiffre dans une case. Pour réaliser cette fonction, il faut chercher si la valeur que l'on met dans cette case respecte les règles d'unicité dans une ligne, dans une colonne et dans une région.

Chacune de ces règles parcourt donc la grille pour trouver les valeurs déjà placées dans cette grille et comme la fonction sera exécutée pour tous les essais, il est particulièrement rentable de l'optimiser. Or, quand une valeur est mise dans une case (i, j), elle n'est plus modifiée tant qu'il n'y a pas d'échec de cette tentative de solution et la ligne i, la colonne j et la région (i, j) contiennent donc déjà cette valeur. Si on conserve cette information, la recherche n'a plus besoin de parcourir la ligne, la colonne et la région, mais juste retrouver cette information.

Une solution consiste à utiliser 3 tableaux [1] de n^2 lignes et n^2 colonnes dont les valeurs seront binaires

1. Ces tableaux seront bien sûr alloués dynamiquement.

(0 ou 1). L'élément (i, j) d'un tableau vaut 1 si la valeur j est présente dans la ligne (resp. colonne ou région) i, 0 sinon. On gagne ainsi n^2 comparaisons pour chaque règle d'unicité testée.

Avec cette version, une grille comme celle de la figure 3, est résolue en 15ms. Le gain de temps dépend directement du nombre de possibilités testées. Il est plus important sur des grilles de taille plus grande. Sur la grille contenue dans le fichier `sud4_4.dat`, 255539 possibilités sont testées en 9,1s pour la version sans stockage, contre 0,8s pour la version avec stockage.

6 Le projet

Le projet clôture le premier semestre. Il met en œuvre l'ensemble des concepts que vous avez vus tout au long du semestre : tableau, fichier, structure, allocation dynamique. Il ajoute une dimension : travailler en équipe, en utilisant bien sûr le gestionnaire de version git.

Ce projet se réalise donc en binôme, que vous devez former avant le début du projet. Lisez le sujet et préparez vos questions pour la première séance qui est encadrée par un enseignant. Vous gérez ensuite votre projet à 2 en autonomie et posez vos questions, de préférence sur l'outil `riot.ensimag.fr` aux enseignants. Vous déposerez sur `gitlab.ensimag.fr` votre code. Les 2 séances prévues dans l'emploi du temps ne sont pas suffisantes pour réaliser ce projet, qui nécessite du travail supplémentaire. Il n'y a pas de rapport à rendre, uniquement votre code.

Ce projet permet de revoir la plupart des notions que vous avez abordées pendant le semestre. Une partie de l'examen écrit sera inspirée de ce projet.

6.1 Travail à réaliser

Vous devez donc réaliser un programme qui effectue les actions suivantes :

1. lire un fichier contenant une grille incomplète, dont la dimension peut être 4,9,16 ou 25,
2. afficher la grille incomplète,
3. résoudre le sudoku,
4. afficher la solution,
5. vérifier que les règles d'unicité sont respectées par la solution trouvée,
6. sauver cette solution dans un fichier.

Faites la première version et testez la, avant d'essayer les optimisations proposées.

6.2 Fichiers de données

Nous fournissons quelques fichiers contenant des sudokus, plus ou moins faciles :

1. Sudoku de taille 2 : très faciles, utiles pour la mise au point des programmes.
   `sud2_1.dat, sud2_2.dat, sud2_3.dat`
2. Sudoku de taille 3 : vous trouverez beaucoup d'autres grilles sur le www.
   - faciles ou très faciles : `sud3_0.dat, sud3_1.dat, sud3_2.dat, sud3_3.dat`
   - moyen : `sud3_4.dat`
   - difficile : `sud3_5.dat`
   - très difficile : `sud3_6.dat`
3. Sudoku de taille 4 :
   - moyen : `sud4_1.dat`
   - difficile : `sud4_2.dat`
   - très difficile : `sud4_3.dat`
4. Sudoku de taille 5 : difficile : `sud5_1.dat`

6.3 Gestion de projet

Lors de la gestion de projet, il est important de définir des jalons afin d'en suivre l'avancement. Ces jalons permettent de définir le travail que chaque membre de l'équipe doit réaliser et pour quelle date.

Voici quelques jalons ou étapes dont vous pourriez vous inspirer pour travailler :

1. Jalon 1 : définir les structures de données et les entêtes de fonctions,
2. Jalon 2 : allocation dynamique et libération mémoire des grilles, fonction de lecture dans un fichier et d'affichage à l'écran,
3. Jalon 3 : résolution récursive simple, vérification des règles,
4. Jalon 4 : sauvegarde dans un fichier,
5. Jalon 5 : optimisations proposées.

6.4 Rendu du projet

Le code sera déposé sur le dépôt gitlab que nous vous avons créé. Il comportera les fichiers sources `*.c` et d'entête `*.h`, ainsi qu'un fichier `Makefile` permettant de compiler le programme. Il comportera aussi un fichier README indiquant ce qui fonctionne et ce qui ne fonctionne pas dans ce projet.

Tout plagiat, qu'il soit interne ou externe à Phelma, est bien sûr interdit et sera sanctionné.

6.5 Travailler en binôme

Quelques conseils pour travailler en binôme

— Commencez par analyser le problème posé, définissez les structures de données dont vous avez besoin et/ou complétez celles que nous vous suggérons, puis définissez les fonctions, leur rôle et leur prototype ensemble. Ensuite, répartissez vous les fonctions à écrire en indiquant dans quels fichiers elles se trouvent.

— Travaillez de manière incrémentale : écrivez une première fonction, compilez puis testez cette fonction avec un programme principal `main` spécifique à cette fonction. Déposez le fichier sur `gitlab.ensimag.fr`. Passez ensuite à une autre fonction.

— Utilisez les outils de debug pour traiter les cas de `segmentation fault`, dûs à des accès mémoire inadéquats : `gdb`, `ddb` et `valgrind` dont vous trouverez une initiation sur le site : http://tdinfo.phelma.grenoble-inp.fr/1AS1/site.

— Evitez de travailler à 2 sur les mêmes fichiers. Bien que cela soit possible, cela peut générer des conflits au moment de déposer votre code sur git, conflits qu'il faut ensuite gérer correctement. Cela demande un peu d'habitude avec `git`.

— Déposer régulièrement votre code sur gitlab, et faites aussi régulièrement un `git pull` pour récupérer le travail de votre binôme.

7 Facultatif pour mieux tester : 9 millions de sudokus à résoudre en un temps minimum

Pour aller plus loin, petit concours : écrire un code pour résoudre (et vérifier) les 9 millions de sudokus de taille 3 du fichier `sudoku.zip`, dans le répertoire `/users/prog1a/C/librairie/projetS12022` des serveurs de l'école et sur le site `tdinfo.phelma.grenoble-inp.fr`.

Commencez par désarchiver les données, selon la commande `unzip sudoku.zip`

Le format de ce fichier est différent des précédents : il contient sur chaque ligne un sudoku de taille 3, une virgule puis la solution de ce sudoku. La première ligne est ainsi celle-ci :

```
301086504046521070500000001400800002080347900009050038004090200008734090007208103,

371986524846521379592473861463819752285347916719652438634195287128734695957268143
```

Il faut faire un programme qui lit un sudoku dans ce fichier, le résout, lit la solution dans le fichier pour la comparer à celle que vous avez trouvée. Il affiche une erreur si les solutions ne sont pas identiques.

Mesurer le temps de chaque résolution, sans compter les temps de lecture des sudokus, des solutions et des comparaisons, cumuler ce temps et l'afficher.

Le temps réalisé sur les machines de l'école en salle M160 (avec la méthode indiquée ci dessous) est de 342 secondes pour résoudre les 9 millions de sudokus, avec un maximum de 0,01s. Le temps total d'exécution du programme (avec les lectures et comparaisons) mesuré par la commande linux `time` est de 6 minutes et 35 secondes.

Pour mesurer le temps pris par un appel de fonction, vous pouvez vous inspirer du code suivant :

```
// Ne pas oublier d'inclure time.h au début pour la fonction clock()
#include <time.h>

main() {
  int cl;
  cl=clock();
    // Appel de la fonction f() dont on veut mesurer le temps d'exécution
  f();
  cl = clock()-cl;
  printf("Temps mesure en secondes par la fonction f : %lf\n",cl/(double)CLOCKS_PER_SEC);
}
```

8 Références

Armando B. Matos : The most difficult Sudoku puzzles are quickly solved by a straightforward depth-first search algorithm, LIACC, https ://www.dcc.fc.up.pt/ acm/sudoku.pdf

Gary McGuire, Bastian Tugemann, Gilles Civario, There is no 16-Clue Sudoku : solving the sudoku minimum number of clues problem, http ://arxiv.org/abs/1201.0749