Projet IF3:
Générer et Résoudre des sudokus

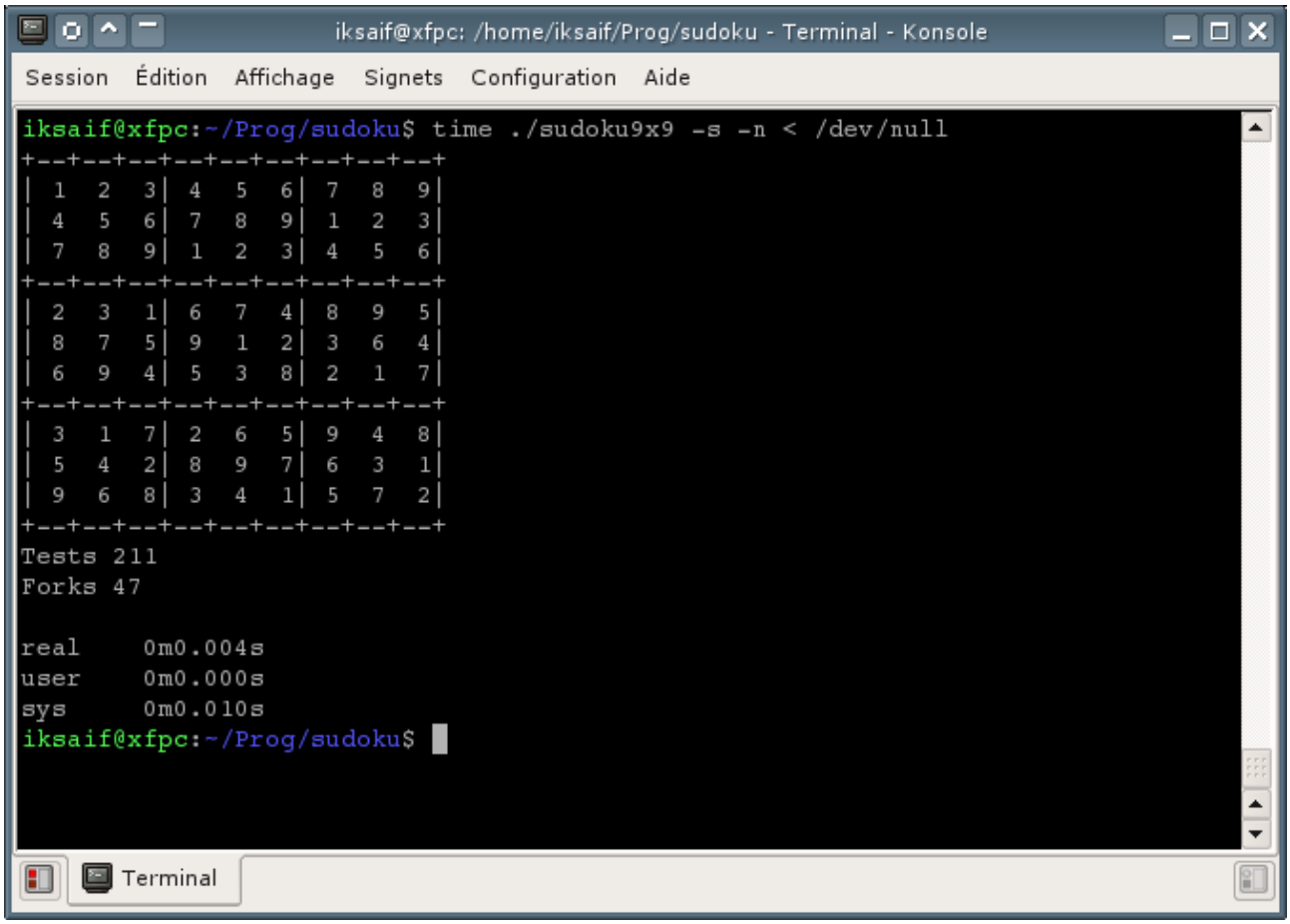


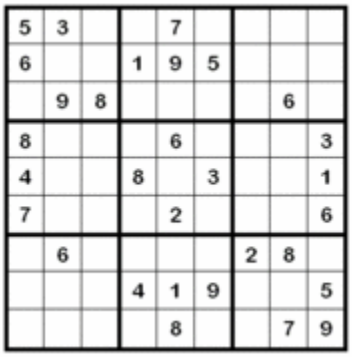
Table des matières

Introduction

1. Choix du Projet

Le but de ce projet est de réaliser un programme permettant de générer et de résoudre rapidement des grilles de Sudoku. L'intérêt étant de trouver une méthode de résolution utilisant le moins possible de ressources ... Le programme pouvant être plus tard couplé à une interface graphique permettant au joueur de résoudre lui même les grilles générées.

2. Principe du Sudoku



La grille de jeu est un carré de neuf cases de côté, subdivisé en autant de carrés identiques, appelés régions. La règle du jeu est simple : chaque ligne, colonne et région ne doit contenir qu'une seule fois tous les chiffres de un à neuf. Formulée autrement, chacun de ces ensemble doit contenir tous les chiffres de un à neuf.

Les chiffres ne sont utilisés que par convention, les relations arithmétiques entre eux ne servant pas. N'importe quel ensemble de signes distincts : lettre, forme, couleur,... peut être utilisé sans changer les règles du jeu. *Dell Magazine*, le premier à publier des grilles, a utilisé des chiffres dans ses publications. Par contre, *Scramblets*, de *Penny Press*, et *Sudoku Word*, de *Knight Features Syndicate*, utilisent tous les deux des lettres.

L'intérêt du jeu réside dans la simplicité de ses règles, et dans la complexité de ses solutions. Les grilles publiées ont souvent un niveau de difficulté indicatif. L'éditeur peut aussi indiquer un temps de résolution probable. Quoi qu'en général, les grilles contenant le plus de chiffres préremplis soient les plus simples, l'inverse n'est pas systématiquement vrai. La véritable difficulté du puzzle réside plutôt dans la difficulté à trouver la suite exacte de chiffres à ajouter.

Plus d'informations : http://fr.wikipedia.org/wiki/Sudoku

3. Outils Utilisés

Pour réaliser ce projet, j'ai utilisé emacs pour éditer le code, et le compilateur C du GNU (GCC) pour compiler le code, le tout tournant dans un environnement linux (Debian GNU/Linux).

Des outils comme gdb,time ou valgrind ont aussi été utilisés pour le déboguage et pour mesurer les performances du programme.

Analyse du sujet

1. Algorithmes de base

L'algorithme le plus simple permettant de résoudre n'importe qu'elle grille est de faire des essais aux hasard jusqu'à ce que la grille soit remplie. Il est clair qu'en utilisant cette méthode le temps de résolution risque d'être très élevé. Il y a donc plusieurs autres méthodes à appliquer avant d'avoir a utiliser des choix arbitraires.

La première est tout simplement la détermination des possibilités de chaque cases en fonction des cases présentes dans la colonne/ligne/région de cette case. Cette technique permet généralement de résoudre les grilles les plus simples.

La deuxième est basée sur la première, il s'agit de trouver une ligne une colonne, ou une région où un nombre n'est possible qu'à un seul endroit, dans ce cas on peut dire que ce nombre est à l'unique endroit où il est possible.

L'algorithme de résolution fonctionne donc en gros comme ça :

```
Début
	grille <- LireGrille()
	TantQue GrilleNonResolue(grille)
		Si Non TestLogique(grille)
		Alors
			Si Non TestUniqueLigneColonneRegion(grille)
			Alors
				ChoixArbitraire(grille)
			FinSi
		FinSi
	FinTantQue
	AfficherGriller(grille)
Fin
```

Cela permet d'exécuter toujours le test le plus rapide, et n'exécuter les autres que si les tests simples ne fournissent pas de résultat. Cela permet aussi de rajouter des techniques simples avant le choix arbitraire sans avoir a beaucoup modifier le programme.

2. Choix arbitraires

Il arrive donc, lorsqu'aucun test n'apporte de résultat, qu'on ait besoin de faire un choix arbitraire. Le problème de ce choix, est qu'il peut être mauvais, et donc rendre impossible la résolution de la grille. Il faut donc rajouter une sauvegarde des anciens choix faits (et des anciennes grilles), afin de pouvoir les annuler. Maintenant, quand aucun test simple ne fonctionne, on vérifie que la grille est possible, si ce n'est pas le cas, alors on annule le dernier choix, sinon, on fait un nouveau choix arbitraire. Il est possible d'utiliser la récursivité pour faire ce genre de chose, mais j'ai préféré le faire en itératif. Il faut aussi vérifier que la grille de départ est bonne, sinon il sera évidement impossible de la résoudre.

L'algorithme est donc modifié :

```
Début
	grille <- LireGrille()
	Si GrillePossible(grille) Alors
		TantQue GrilleNonResolue(grille)
```

```
                  Si Non TestLogique(grille)
                  Alors
                        Si Non TestUniqueLigneColonneRegion(grille)
                        Alors
                              Si Non GrillePossible(grille)
                              Alors
                                    AnnulerChoix(grille)
                              FinSi
                              ChoixArbitraire(grille);
                        FinSi
                  FinSi
            FinTantQue
            AfficherGrille(grille)
      FinSi
Fin
```

3. Structures de données

Ayant choisis un algorithme itératif plutôt que récursif, j'ai du utiliser une structure de donnée permettant la sauvegarde et l'annulation de certains changements.
J'ai donc du choisir d'utiliser une liste chaînée de structure « sudoku » (décrite plus loin). Un tableau aurait bien entendu pu aussi être utilisé, et sa taille modifiée avec « realloc » lorsqu'il aurait été plein.

Le problème des sauvegardes étant réglé, il a donc fallut créer une structure « sudoku » permettant de rassembler toutes les informations nécessaires.
Tout d'abord, la grille, qui est un tableau de tableau d'entiers non signés (tableau a double entrée). Pour la résolution et pour faciliter les tests, j'ai choisis d'utiliser des masques binaires pour chaque case/ligne/colonne/région, cela augmente grandement la vitesse du test basique (TestLogique() dans d'algorithme) qui est le plus utilisé, car il suffit de faire un ET logique des masques. L'utilisation d'un masque limite par contre la taille de la grille par le nombre de bit d'un entier non signé, ce qui varie suivant les architectures.

Sont aussi stockés : le dernier choix arbitraire fait ainsi que sa position, le nombre de tests fait (le nombre de forks étant le nombre de choix arbitraire) ainsi que le nombre de cases encore inconnues. Enfin, il y'a un pointeur vers la structure précédente.

Cela donne donc :

```
struct sudoku {
        /* Grille */
        unsigned int grid[COLS][ROWS];


        /*
        ** Masques
        ** Un 1 represente les nombres possibles
        ** m_case = m_rows&m_cols&m_sqr mais on le stock pour pas refaire tout
        ** les calculs quand on a besoin de sa valeur
        ** Et parce qu'il est nécessaire pour les choix arbitraires
        */
        unsigned int m_case[COLS][ROWS];
        unsigned int m_rows[ROWS];
        unsigned int m_cols[COLS];
        unsigned int m_sqr[(COLS/SQR_COLS)*(ROWS/SQR_ROWS)];
```

```
        /* Choix arbitraires */
        unsigned int arb_x;
        unsigned int arb_y;
        unsigned int arb_nb;

        /* Variables diverses */
        unsigned int unknown;

        /* Nombre de tests */
        double tests;

        /* Nombre de tests arbitraires */
        double forks;

        /* Essai précedent */
        struct sudoku *prev;
};
```

4. Fonctions Importantes

- ***Fonction Principale : « main() »***

Cette fonction analyse les arguments donnés par l'utilisateur lors du lancement du programme, et en fonction d'eux appelle les fonctions qui permettent de faire ce que l'utilisateur demande.

Elle prend en argument un entier et un tableau de chaînes de caractères et renvoi un entier.

- ***Initialisation : « init_grid() »***

Cette fonction initialise une grille. Toutes les cases de la grille sont mises à 0, et les masques sont tous à 2^nb_max – 2, nb_max étant le nombre maximum, par exemple 9 dans une grille 9x9. Pour une grille 9x9 on aura donc en binaire : 1111111110.

- ***Analyse de l'entrée standard: « parse_in() »***

Cette fonction permet de remplir la grille avec ce que donne l'utilisateur au programme. Elle recalcule les masques à chaque fois qu'un nombre autre que 0 est mis dans la grille.

- ***Résolution: « resolv() »***

Cette fonction ne fait que choisir et faire les tests en boucle jusqu'à ce que la grille soit résolue. Elle utilise toujours le dernier élément de la liste chaînée.

- ***Test basique: « basic() »***

C'est le test le plus simple, il ne fait que faire un ET logique sur les différents masques pour tenter de déduire quel nombre est dans quelle case. Les grilles les plus simples (celles qu'on peut trouver dans les journaux par exemple), sont généralement résolues uniquement avec ce test.

- ***Un Seul dans une ligne/colonne/région: « single_in_row_col() » et « single_in_box() »***

Ces deux fonctions permettent de vérifier qu'un nombre n'est pas présent qu'une seule fois dans une ligne/colonne/région. Si tel est le cas, alors c'est qu'il est à cette position dans la grille.

- ***Choix Arbitraire: « random_choice() » et « cancel_random_choice() »***

Ces fonctions gèrent les choix arbitraires. Lorsqu'un choix arbitraire est fait, un nouvel élément est rajouté à la liste chaînée, celui-ci est une copie de l'ex-dernier élément. A partir de ce dernier élément, on cherche la case ayant le moins de possibilités, et on change ça valeur en utilisant la plus petite possibilité.

Lors de l'annulation d'un choix, on reprend l'avant dernière grille, et on met le choix arbitraire qui a conduit a une grille fausse comme impossible dans cette grille.

- ***Vérifications: « check_grid() »***

Cette fonction permet de vérifier qu'une grille est valide, donc qu'elle respecte bien les règles d'un sudoku.

- ***Affichage: « show_grid() », « show_nice_grid() »***

Ces deux fonctions permettent d'afficher une grille (terminée ou non). La première est destinée à être lue par un programme, la dernière par un humain.

Guide d'utilisation

Compilation

UNIX

Sous n'importe quel UNIX, sous GNU/Linux en particulier, il suffit de lancer le script bash « compile.sh » ou de compiler à la main avec gcc :

```
gcc $CFLAGS -std=gnu99 -O2 -o sudoku9x9 sudoku.c -DSIZE9x9
gcc $CFLAGS -std=gnu99 -O2 -o sudoku16x16 sudoku.c -DSIZE16x16
gcc $CFLAGS -std=gnu99 -O2 -o sudoku25x25 sudoku.c -DSIZE25x25
```

Windows

Pour compiler le programme avec Dev Cpp (testé uniquement avec la dernière version de DevCpp) il faut régler les options de compilations dans « Options du Projet » et spécifier la taille de la grille : « -DSIZE9x9 », « -DSIZE16x16 » ou « -DSIZE32x32 ».

Note: *il est possible d'optimiser le programme en utilisant les paramètres de compilation suivants « -std=gnu99 -Wall -pedantic-errors -march=athlon-xp -O2 -pipe -fforce-addr -fomit-frame-pointer -funroll-loops -frerun-loop-opt -frerun-cse-after-loop -falign-functions=4 » (en remplacent athlon-xp par la bonne valeur bien entendu !) cela double la taille du binaire, mais permet de gagner 100ms sur la résolution d'une grille de 25x25 !*

Utilisation

Le programme s'utilise uniquement en mode console. Sous UNIX, je pense qu'il est inutile d'expliquer comment faire. Sous Windows, il suffit de faire « Démarrer » -> « Exécuter » et de taper cmd.

Pour la suite, sudoku9x9 peut être remplacé par sudoku16x16, sudoku25x25 ou sudoku.exe (sous Windows) ....

Les options peuvent être connues avec l'option « -h » :

```
$ ./sudoku9x9 -h
Usage: ./sudoku9x9 [-n] [-s|-m diff_lvl] [-v verbose_lvl]
        -n              Afficher la grille avec les décorations
        -s              Résoudre (Par defaut)
        -c diff_lvl     Créer une grille de difficulté diff_lvl {1,..,5}
        -v verbose_lvl  Niveau de Verbose: {1,2,3,4}
```

Le programme a besoin d'une grille de départ, qui peut être vide. Cette grille est lue sur l'entrée standard.

Exemple de grille (9x9):

```
 9  7  15
  1 356
5 74 2 3
 5  8   6
7  1  389
3   261
 78  3 4
  5 4 2 7
4 2 6 89
```

Les chiffres peuvent aussi être à la suite (sans saut de ligne). La grille n'a pas besoin d'être complète. Les espaces peuvent être remplacés par des 0 dans le cas d'une grille 9x9 et par des ` dans le cas de grilles 16x16 et 25x25.

Il n'est pas forcement très pratique d'utiliser le clavier pour taper les grilles, il est donc conseillé d'utiliser des fichiers, on peut par exemple faire :

```
Résous une grille a partir de rien
./sudoku9x9 -s < /dev/null


Résous une grille a partir du fichier grille
cat grille | ./sudoku9x9 -s -n


Génère une grille de difficulté 5 à partir de rien puis résous cette grille :
./sudoku9x9 -m 5 < /dev/null | ./sudoku9x9 -s -n
```

Note: *Pour windows, on peut remplacer /dev/null par un fichier vide.*

Conclusion

Les temps sont assez bon, 5ms pour générer une grille de difficulté 5 et de taille 9x9 puis la résoudre, 80ms pour une grille de taille 16x16 et 1sec pour une grille de taille 25x25 (certaines grilles de 25x25 peuvent mettre beaucoup plus de temps ..).

A titre de comparaison, le programme « ksudoku » met à peu prêt 20secondes pour générer une grille de 25x25. Le programme « kaiketsu » met lui plusieurs secondes à résoudre une grille de 9x9.

De plus toutes les grilles valides peuvent être résolues par le programme.

La réalisation de ce programme m'a permit de trouver une occasion d'utiliser « getopt » (qui permet de parser facilement les lignes de commandes). J'ai aussi du utiliser les opérateurs binaires de manière assez importante ce qui m'a permit d'en apprendre plus sur leur fonctionnement.