Etudiant 1 : Elodie ABGRALL

Etudiant 2 : Julien GRAVE

Niveau : *TC03*

Rapport de NF05 – Automne 2012

Mini projet :

SUDOKU

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

SOMMAIRE

Introduction

1. Mode d’emploi du programme

2. Description détaillée des algorithmes utilisés

2.1. L’algorithme « resolveur »

2.2. L’algorithme « generateur »

Conclusion

Annexes

Références

INTRODUCTION

Ce projet a pour but de créer un jeu de Sudoku avec une interface graphique. Le programme que nous avons réalisé se décompose en deux grandes parties qui correspondent à deux « modes de jeu » pour l'utilisateur. Il peut soit demander à l'ordinateur de lui générer une grille afin que l'utilisateur la résolve, soit proposer une grille à l'ordinateur pour que celui-ci lui affiche la solution.

Pour la réalisation de l'interface graphique, nous avons fait le choix d'utiliser la bibliothèque SDL qui semblait être une bonne solution pour l'élaboration d'un jeu en 2D tel que le jeu du Sudoku.

Ce document se décompose donc en deux grandes parties : une première qui décrit le programme dans son ensemble et qui donne un mode d'emploi du programme, et une deuxième partie qui donne une description plus détaillée des algorithmes « resolveur » et « generateur » qui constituent la part principale du programme.

Vous trouverez en annexe certains passages du code source mentionnés dans le rapport.

1. Mode d’emploi du programme

Ce programme fait donc appel à la bibliothèque SDL pour l’interface graphique. Cette interface permet de rendre le visuel du jeu plus esthétique et donc plus clair également pour l’utilisateur. Lors du lancement du programme, l’utilisateur a le choix entre plusieurs modes de jeu :

- Lancer une nouvelle partie
- Reprendre une partie
- Résoudre une grille qu’il proposera

Si l’utilisateur choisit le mode « Résoudre une grille », il devra remplir une grille vierge à l’écran. Il suffit pour cela que l’utilisateur fasse un clic gauche sur une case de la grille et qu’il rentre le nombre correspondant à cette case en appuyant sur les touches du clavier numérique, et cela pour toutes les cases qui contiennent un chiffre. Ensuite l’utilisateur a juste à cliquer sur « Résoudre » et le programme affiche la grille complètement résolue.

Sinon, l’utilisateur peut choisir le mode « Nouvelle partie » et dans ce cas le programme lui demande de choisir le niveau de difficulté de la grille qu’il souhaite résoudre, avec trois choix possibles : facile, moyen ou difficile. Suivant le niveau de difficulté choisi, le programme va générer une grille que l’utilisateur pourra remplir en utilisant la souris. Il suffit de faire un clic gauche sur la case souhaitée et de rentrer le chiffre que l’on veut écrire pour qu’il s’affiche à l’écran, et si l’on se trompe il suffit de recliquer sur la case à modifier et de rentrer le nouveau chiffre pour que celui-ci soit modifié. Une fois la grille remplie, l’utilisateur clique sur « Valider » et le programme vérifie la réponse donnée par l’utilisateur et lui indique s’il a trouvé la bonne solution ou non.

L’utilisateur peut également continuer la dernière grille qu’il était en train de compléter lorsqu’il a quitté le jeu pour la dernière fois puisque celle-ci s’enregistre automatiquement dans un fichier à la fermeture du programme. Pour cela il a juste à cliquer sur le bouton « Reprendre une partie ».

L’utilisateur a la possibilité d’imprimer les grilles de Sudoku en cliquant sur l’icône imprimante situé en haut à droite de l’écran. Cela fait une capture d’écran et crée un fichier image de la grille que l’utilisateur pourra ensuite imprimer s’il le souhaite.

2. Description détaillée des algorithmes utilisés

Cette partie est consacrée a la description détaillée des deux algorithmes principaux du programme : l'algorithme « resolveur » et l'algorithme « generateur ». La grille de Sudoku est formée d'un tableau tridimensionnel qui nous permet de stocker les possibilités des cases dans ce que l'on appellera sa « profondeur ».

2.1. L'algorithme « resolveur »

L'algorithme « resolveur » se trouve en annexe à la page 10. Il se décompose en trois grandes parties.

Tout d'abord il compte le nombre de cases vides restantes dans la grille afin de savoir quand il a terminé de résoudre la grille.
Ensuite il parcourt toutes les cases vides de la grille et recherche les différentes valeurs possibles pour chacune d'entre elles. Les possibilités sont stockées dans la « profondeur » de la case correspondante.
Enfin il fait appel à l'algorithme « resolutionGrille » (en annexe page 11) qui exécute différents raisonnements humains afin de résoudre la grille. En outre, le niveau de difficulté de la grille est fonction des raisonnements employés. Il commence par le niveau 1, si la grille n'est pas totalement résolue alors il passe au niveau 2 qui effectue d'autres tests et ainsi de suite jusqu'à ce que la grille soit complètement résolue. En dernier recours, on emploiera une méthode de « backtracking » afin de résoudre même les grilles les plus difficiles. Cependant si après avoir effectué le test du backtracking au niveau 5, la grille n'est toujours pas résolue, cela signifie que la grille rentrée n'était pas valide.

Les différents tests exécutés dépendent de la difficulté de la grille. Ils sont basés sur les raisonnements que font les joueurs lorsqu'ils essayent de résoudre une grille de Sudoku. Ces tests sont les suivants :

- **« positionUnique » :** c'est une procédure qui détermine si l'une des possibilités de la case à résoudre est l'unique possibilité présente sur sa ligne, sur sa colonne et dans son bloc.
- **« possibiliteUnique » :** c'est une procédure qui teste si la case à résoudre ne contient qu'une seule possibilité.
- **« possibiliteAlignee » :** dans cette procédure, si au sein d'un bloc, les possibilités identiques sont toutes alignées alors on peut supprimer cette possibilité du restant de la ligne.
- **« nakedTuple » :** c'est une procédure qui recherche sur une ligne l'existence de k cases pour lesquelles seulement k valeurs sont possibles (exemple k=2 ou k=3). Dans ce cas, ces k valeurs peuvent être interdites pour l'ensemble des autres cases de la ligne, sauf les k cases repérées.

- **« hiddenTuple » :** c'est une procédure qui recherche sur une ligne l'existence de k valeurs pour lesquelles seulement k cases sont possibles (exemple k=2 ou k=3). Dans ce cas, toutes les autres valeurs sont interdites pour ces k cases.
- **« forcingChains » :** cette procédure permet d'essayer une valeur dans une case et on regarde si ce choix nous mène à une contradiction ou non. Si oui alors on sait que la valeur n'est pas la bonne et on doit revenir en arrière. C'est la méthode du backtracking.

Tous ces tests permettent d'évaluer le niveau de difficulté d'une grille. Les tests « positionUnique » et « possibiliteUnique » sont utilisés à partir du niveau facile, le test « possibiliteAlignee » est utilisé à partir du niveau moyen, les tests « nakedTuple » et « hiddenTuple » sont utilisés à partir du niveau difficile, et enfin le test « forcingChains » est utilisé en dernier recours afin de pouvoir résoudre toutes les grilles sans exceptions.

2.2. L'algorithme « generateur »

L'algorithme « generateur » se trouve en annexe à la page 12. Il se décompose en deux grandes parties.

Tout d'abord, l'algorithme fait appel à la procédure de « backtracking » (en annexe page 12) qui permet de remplir complètement une grille de Sudoku avec les contraintes imposées (un même chiffre ne peut apparaître qu'une seule fois dans une ligne, une colonne et un bloc). Cette procédure consiste à rechercher la première case vide de la grille et à y mettre un chiffre aléatoirement (qui n'est pas déjà présent dans la ligne, colonne ou bloc). Ensuite on passe à la case vide suivante et on la remplie également et ainsi de suite jusqu'à ce que la grille soit complètement remplie. Lorsqu'une case n'a pas de solution, on revient à la case précédente et on modifie sa valeur en mettant une autre possibilité, choisie aléatoirement et différente de la précédente, on repasse à la case suivante et ainsi de suite pour toutes les cases. Ainsi on peut parfois remonter à la case précédente ou à 2, 3 ou 4 cases avant, le but étant de réussir à remplir complètement une grille. Une fois ce travail achevé, on a généré une grille complète que l'on va pouvoir modifier en fonction du niveau désiré.

Ensuite l'algorithme fait appel à la fonction « vidage » (en annexe page 13) qui permet de retirer des cases de la grille en fonction du niveau de jeu requis par l'utilisateur. Cette fonction commence par fixer le nombre de cases qu'il restera après le vidage. Cette valeur est comprise entre 30 et 40 pour les niveaux strictement inférieurs à 3 et entre 24 et 35 pour les autres niveaux (la valeur de ces bornes est un choix de notre part). On enlèvera des cases à partir de ce nombre jusqu'à 81. La procédure choisie aléatoirement une case de la grille et modifie sa valeur pour que la case soit vide (la valeur est alors 0). La nouvelle grille est envoyée au résolveur qui renvoie la grille résolue ainsi que le niveau de cette grille. Si le niveau retourné par le résolveur est supérieur au niveau demandé par l'utilisateur, alors on remet la valeur de la case et on en supprime une autre jusqu'à ce que le niveau retourné soit inférieur ou égal au niveau requis. Et on

enlève des cases jusqu'à ce que la boucle soit terminée et qu'il ne reste plus que le nombre de case souhaité.

Enfin, la procédure de vidage retourne au générateur le niveau le plus grand qu'elle a rencontré lors du retrait des cases de la grille. Si celui-ci correspond bien au niveau attendu alors la grille est correcte. Sinon, c'est que toutes les cases se résolvent avec un niveau strictement inférieur à celui demandé, on relance donc à nouveau le processus de backtracking puis de vidage, ainsi de suite jusqu'à obtention du niveau souhaité.

CONCLUSION

L'association des algorithmes utilisés et de la SDL permet une utilisation très simple et rapide du programme par l'utilisateur. De plus ce programme permet différentes options de jeu à l'utilisateur : proposer une grille de n'importe quel niveau ou résoudre une grille avec 3 niveaux différents, mais il peut également sauvegarder ses grilles comme des images pour ensuite les imprimer s'il le souhaite.

Toutefois, ce programme a des limites. En effet, les tests basés sur le raisonnement humain qui sont utilisés permettent uniquement de résoudre des grilles jusqu'au niveau difficile et pas des grilles diaboliques par exemple. Donc le niveau diabolique n'est pas proposé à l'utilisateur dans le mode de jeu « Nouvelle partie ». En revanche on a mis en place la fonction de backtracking ou « forcingChains » qui permet la résolution de toutes les grilles de Sudoku que l'utilisateur pourra proposer, que le niveau soit facile, difficile ou encore diabolique. Donc l'amélioration que l'on pourrait apporter à ce programme consistera à développer d'autres tests basés sur les raisonnements humains tels que le XWing ou le Swordfish afin de résoudre des grilles plus compliquées et ainsi proposer à l'utilisateur un plus large choix de niveaux.

Il y a d'autres points sur lequel le programme pourrait être amélioré. L'utilisateur ne peut enregistrer qu'une seule image de Sudoku à la fois. De plus, une amélioration du design pourrait être apportée à l'interface. Enfin, d'un point de vue pratique, il serait intéressant pour l'utilisateur de pouvoir inscrire dans les cases vides les différentes possibilités qu'il envisage.

ANNEXES

Annexe 1 : « Résolveur »

```
/// Procédure de résolution d'une grille de Sudoku
void resolveur(int grille[9][9][10], int *niveau)
{
    // On compte le nombre de cases à résoudre
    int compteurDeCases=nombreCasesRestantes(grille);
    bool stop=false;
    // Recherche les possibilités des cases à résoudre de la grille
    recherchePossibilitesGrille(&compteurDeCases, grille, niveau);
    /* Une fois un premier tour complet de la grille effectuée, toutes les
possibilités de chaque case ont été déterminées.
    On passe donc à la résolution de la grille jusqu'à ce que le compteur
de cases restantes soit à zéro.
    Si l'on s'aperçoit que la variable booléenne stop ne varie plus alors
cela signifie que les traitements lié au niveau en cours ne suffise pas, il
faut donc passer au niveau suivant. */
    while((compteurDeCases>0)&&(*niveau!=6))
    {
        // On initialise la variable stop comme "fausse"
        stop=false;
        // On lance la résolution de la grille
        resolutionGrille(&compteurDeCases, grille, niveau, &stop);
        // Si la variable stop est toujours fausse en sortant de
l'algorithme de résolution alors cela signifie qu'aucune des traitements
effectués n'a modifié la grille, on passe donc au niveau suivant
        if(stop==false)
        {
            (*niveau)++;
        }
    }
    if(compteurDeCases>0)
    {
        // Si le compteurs de cases n'est pas nul alors la grille n'est pas
résoluble avec l'algorithme employé
        printf("\nGrille non-r\202soluble !\n");
        affichageGrille(grille); // temporaire
        exit(EXIT_FAILURE);
    }
}
```

Annexe 2 : « Résolution de la grille »

```
/// Procédure qui permet de parcourir toute la grille en quête des solutions de toutes les
cases
void resolutionGrille(int *compteurDeCases, int grille[9][9][10], int *niveau, bool *stop)
{
    int ligne=0, colonne=0;
    // On parcourt toute la grille pour rechercher les solutions des cases
    for(ligne=0; ligne<9; ligne++)
    {
        for(colonne=0; colonne<9; colonne++)
        {
            if(grille[ligne][colonne][0]==0)
            {
                // On effectue des tests concernant la case où l'on se trouve
                testerCase(ligne, colonne, grille, compteurDeCases, niveau, stop);
            }
        }
    }
}

/// On effectue des tests sur la case en fonction du niveau de difficulté de la grille
void testerCase(int ligneCase, int colonneCase, int grille[9][9][10], int *compteurDeCases,
int *niveau, bool *stop)
{
    if(*niveau==1) // Très facile
    {
        positionUnique(ligneCase, colonneCase, grille, compteurDeCases, stop);
    }
    else if(*niveau==2) // Facile
    {
        possibiliteUnique(ligneCase, colonneCase, grille, compteurDeCases, stop);
        positionUnique(ligneCase, colonneCase, grille, compteurDeCases, stop);
    }
    else if(*niveau==3) // Moyen
    {
        possibiliteAlignee(ligneCase, colonneCase, grille, stop);
        positionUnique(ligneCase, colonneCase, grille, compteurDeCases, stop);
        possibiliteUnique(ligneCase, colonneCase, grille, compteurDeCases, stop);
    }
    else if(*niveau==4) // Difficile
    {
        hiddenTuple(ligneCase, colonneCase, grille, stop);
        nakedTuple(ligneCase, colonneCase, grille, stop);
        possibiliteAlignee(ligneCase, colonneCase, grille, stop);
        positionUnique(ligneCase, colonneCase, grille, compteurDeCases, stop);
        possibiliteUnique(ligneCase, colonneCase, grille, compteurDeCases, stop);
    }
    else if(*niveau==5)
    {
        forcingChains(grille, compteurDeCases);
        *stop=true;
    }
}
```

Annexe 3 : « Génération par backtracking puis vidage »

```
/// Générateur de grille de sudoku
void generateur(int grille[9][9][10], int *niveauRequis)
{
    srand(time(NULL));
    int niveauTeste=0;
    do
    {
        int ligne=0, colonne=0;
        // On initialise la grille
        for(ligne=0; ligne<9; ligne++)
        {
            for(colonne=0; colonne<9; colonne++)
            {
                grille[ligne][colonne][0]=0;
            }
        }
        // On envoi la grille au backtracking qui va la remplir en entier de manière aléatoire
        backtracking(grille, 0);
        niveauTeste=(*niveauRequis);
        // On retire des cases à la grille
        vidage(grille, &niveauTeste);
    }
    while(niveauTeste!=(*niveauRequis)); // Tant que la grille n'a pas le niveau désiré on
recommence
}

/// Fonction qui remplie une grille case par case.
bool backtracking(int grille[9][9][10], int position)
{
    // Si la grille est complètement remplie alors on a terminé et on retourne true
    if(position==81)
    {
        return true;
    }

    int i=position/9, j=position%9;
    // Si la case est déjà remplie on passe à la suivante
    if(grille[i][j][0]!=0)
    {
        return backtracking(grille, position+1);
    }

    int k=0, *possibilites=initSansDoublons(1,10);
    // On test les possibilités de manière aléatoire et sans doublons
    melangeur(possibilites, 9);
    // On teste toutes les possibilités que peut prendre une case
    for(k=0; k<9; k++)
    {

if(absentSurLigne(possibilites[k],grille,i)&&absentSurColonne(possibilites[k],grille,j)&&absen
tSurBloc(possibilites[k],grille,i,j))
        {
            grille[i][j][0]=possibilites[k];
            // Si le tableau se remplie en entier toutes les cases sont justes et on retourne
true
            if (backtracking(grille, position+1))
            {
                return true;
            }
        }
    }

    free(possibilites);
    // Si une case ne peut prendre aucune possibilités alors on retourne false et on passe à
la possibilité suivante de la case précédente
    grille[i][j][0]=0;
    return false;
}
```

```c
/// Procédure qui permet de retirer des cases à la grille en fonction du niveau requis
void vidage(int grille[9][9][10], int *niveauRequis)
{
    int nombreDeCases=0, indice2=0, *position=NULL, indice=0, niveauMax=0, grille2[9][9][10] =
{{{0}}}, sauvegarde[9][9][10] = {{{0}}}, niveauTeste=0, compteur=0, ligne=0, colonne=0;
    // On choisit le nombre de cases de départ de manière aléatoire en fonction du niveau
désiré
    if(*niveauRequis<3)
    {
        nombreDeCases=chiffreAleatoire(30,40);
    }
    else
    {
        nombreDeCases=chiffreAleatoire(24,35);
    }
    // On crée un tableau avec les positions des cases de manière aléatoire
    position = malloc(81*sizeof(int));
    while(indice<81)
    {
        position[indice]=chiffreAleatoire(0,81);
        indice2 = 0;
        while(indice2!=indice&&position[indice]!=position[indice2])
        {
            indice2++;
        }
        if(indice==indice2)
        {
            indice++;
        }
    }
    indice=0;
    // On retire le nombre de cases désirées de la grille
    for(compteur=nombreDeCases; compteur<81; compteur++)
    {
        copierTableau(grille, sauvegarde);
        do
        {
            copierTableau(sauvegarde, grille);
            ligne = position[indice]/9;
            colonne = position[indice]%9;
            grille[ligne][colonne][0]=0;
            copierTableau(grille, grille2);
            niveauTeste=1;
            resolveur(grille2, &niveauTeste);
            indice++;
        }
        while(niveauTeste>(*niveauRequis)&&indice<81);
        /* Tant qu'on a pas une case qui se résout avec un niveau de difficulté inférieur ou
égal à celui désiré
         alors on passe à la case suivante du tableau des positions jusqu'à la fin de ce
dernier */
        // On enregistre le niveau maximum rencontré lors de la résolution de la grille
        if(niveauTeste>niveauMax)
        {
            niveauMax=niveauTeste;
        }
        // Si on a fini de parcourir toutes les cases on arrête de retirer des cases
        if(indice==81)
        {
            break;
        }
    }
    free(position);
    *niveauRequis=niveauMax;
}
```

Références

http://www.emn.fr/z-info/jussien/publications/laburthe-JFPC06.pdf

http://www.siteduzero.com/tutoriel-3-14189-apprenez-a-programmer-en-c.html

http://www.siteduzero.com/tutoriel-3-360004-le-backtracking-par-l-exemple-resoudre-un-sudoku.html