Résolution du Sudoku

On part d'une grille de Sudoku, partiellement remplie, comme on en trouve dans tous les journaux. Nous allons faire le programme qui va remplir toute la grille et résoudre le problème. Dans ce type de jeu, il existe toujours une solution unique, mais si l'on enlève une ou plusieurs des cases qui étaient remplies au départ, on trouve plusieurs solutions, que le programme donnera aussi.

La méthode de programmation que nous utilisons est récursive, en arborescence, et elle donne une réponse immédiate, même dans les cas des Sudoku les plus difficiles, par exemple ce cas considéré comme le plus difficile :

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

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

la grille initiale (*à gauche*), et la grille remplie (*à droite*) après exécution du programme.

Signalons que cette méthode de résolution est complètement différente de la méthode manuelle. Là où l'ordinateur essaye toutes les possibilités à grande vitesse, le joueur humain remplit les cases au coup par coup, à coup sûr, en ne faisant finalement des essais multiples que dans les cas de Sudoku « diaboliques ». [1]

Conditions initiales

0	1	2	3	4	5	6	7	8
9	10							
18								
27								
36								
45								
54								
63								
72								80

Les cases de la grille sont numérotées de 0 à 80 de gauche à droite et de haut en bas. Puis on remplit les cases initiales, en plaçant les chiffres (entre 1 et 9) correspondants dans un tableau *a*[81] indexé par les numéros des cases, par exemple si la case numéro 0 contient le chiffre 1, on fait *a*[0]=1. Les cases vides restent à 0. A leur tour les lignes sont numérotées de 0 à 8 et les colonnes aussi. Pour chaque numéro *i* d'une case on peut lui associer la ligne *L* et la colonne *C* où se trouve la case. Il suffit de faire *L* = *i* /9 et *C* = *i* % 9.

[1] L'idéal serait de faire un programme à l'image de ce que fait le joueur humain. Si un tel programme est aisé pour traiter les cas dits faciles et même difficiles de Sudoku, cela se complique dès que l'on doit faire plusieurs essais avant de trouver la solution unique.

Pour faciliter les choses, on indexe aussi chaque chiffre placé dans la grille par sa ligne *L* et sa colonne *C*, grâce à un tableau *chiffre*[9][9], en faisant *chiffre*[*L*][*C*]=*a*[*i*]. Enfin, on parcourt la grille pour faire ressortir les cases qui sont à 0, en plaçant les numéros de ces cases dans un tableau *d* [], indexé à partir de 0. Par exemple, *d*[0]=1, *d*[1]=2, etc. Ce tableau a pour longueur utile le nombre des cases vides, soit *nbcasesaremplir* dans le programme. Il reste à remplir ces cases qui sont placées dans le tableau *d*[]. On en déduit la « fonction » *conditionsinitiales*() :

```
void conditionsinitiales(void)
{  int i,L,C,k;       /* les tableaux a[] et chiffre[][] sont déclarés en global, ils sont à 0 */
   a[0]=1;a[5]=7;a[7]=9; a[10]=3; a[13]=2;a[17]=8;a[20]=9;  a[21]=6;a[24]=5;
   a[29]=5;a[30]=3; a[33]=9;a[37]=1;a[40]=8;a[44]=2;  a[45]=6;a[50]=4;
   a[54]=3;a[61]=1;a[64]=4; a[71]=7;a[74]=7;a[78]=3;
   k=0;
   for(i=0;i<81;i++)
   if (a[i]!=0) { L=i/9; C=i%9; chiffre[L][C]=a[i];}
   else {nbcasesaremplir++; d[k++]= i; }
}
```

Pour terminer, on dessine la grille initiale, grâce à la fonction *dessin*() et la sous-fonction *carre*(). Pour cela on colorie chaque case remplie, et l'on place son chiffre à l'intérieur. Avec un programme utilisant le mode graphique *SDL* ainsi que la bibliothèque *TTF* pour mettre du texte dans l'image,[2] cela donne :

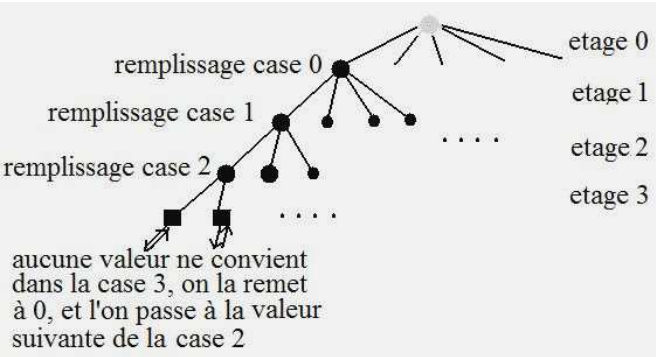
```
void carre(int ligne, int colonne, int n, int couleur)
  { int i,j;       /* dessin d'un carré de longueur pas, dans la ligne et la colonne données */
    for(i=xorig+pas*colonne; i< xorig+pas*(colonne+1);i++)
    for(j=yorig+pas*ligne; j<yorig+pas*(ligne+1);j++)
    if (n!=0)  putpixel(i,j,couleur);    /* coloriage du carré de la case concernée */

     if (n!=0)       /* placement du chiffre n dans le carré */
       { sprintf(chiffr,"%d",n);
         texte=TTF_RenderText_Solid(police,chiffr,couleurnoire);
         position.x=xorig+12+pas*colonne; position.y=yorig+3+pas*ligne;
         SDL_BlitSurface(texte,NULL,screen,&position);
       }
  }

void dessin(void)
{ int i,L,C;
  for(i=0;i<81;i++) {L=i/9;C=i%9; carre(L,C,chiffre[L][C],couleurcase); }
  for(i=0;i<=3;i++) line(xorig,yorig+3*pas*i,xorig+9*pas,yorig+3*pas*i,black);
  for(i=0;i<=3;i++) line(xorig+3*pas*i,yorig,xorig+3*pas*i,yorig+9*pas,black);
}
```

[2] Si l'on n'utilise pas le mode graphique, on peut se contenter d'utiliser des *printf*() pour dessiner la grille (ainsi que la fonction *getchar*() pour faire des pauses dans le programme). Par contre, avec la bibliothèque graphique SDL, il faudra ajouter dans le programme les fonctions *pause*(), *putpixel*(), *getpixel*(), *line*(), que l'on trouve dans la rubrique *book programs* de mes pages web, où se trouve le programme *graphical functions*.

La fonction de recherche des solutions par arborescence

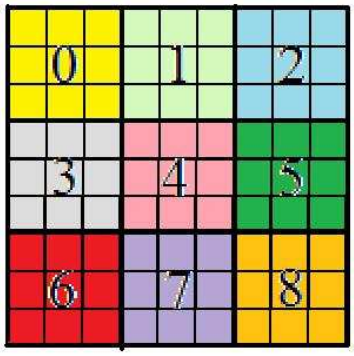
On va prendre successivement les cases vides successives, telles qu'elles ont été enregistrées dans le tableau *d*[]. Le programme principal va appeler la fonction *arbre*(0), où 0 indique que l'on est à l'étage 0 de l'arborescence. Cela exprime que l'on va déterminer quels sont les chiffres que l'on peut mettre dans la case 0 du tableau *d*[]. On va prendre chacune de ces possibilités de la case 0 à tour de rôle, et à chaque fois, la case 0 étant remplie, on rappelle la fonction avec *arbre*(1) pour remplir à son tour la case 1 du tableau *d*[]. Et ainsi de suite. On aura une solution du problème lorsque toutes les cases de la grille seront remplies, c'est-à-dire lorsque l'étage de l'arbre vaut *nbcasesaremplir* (le nombre des cases vides au départ). Cela constitue le test d'arrêt de la fonction *arbre*().

Lorsque l'on est à l'étage *etage* de l'arborescence, la case à remplir est *d*[*etage*]. On cherche alors à connaître tous les chiffres possibles que l'on peut mettre dans cette case, étant entendu que l'on a rempli les cases correspondant à la branche de l'arbre où l'on se trouve, et les chiffres possibles sont placés dans un tableau *b*[]. On remplit la case avec chacun des chiffres de ce tableau *b*[] à tour de rôle, avec une boucle *for*, et à chaque fois on rappelle la fonction *arbre*(*etage*+1). Mais il peut arriver que la descente dans l'arborescence se bloque avec aucun chiffre possible dans une case, et cela avant d'atteindre l'étage du test d'arrêt. Dans ce cas la fonction *arbre*(*etage*+1) ne se rappelle plus, le remplissage partiel effectué jusque là n'est pas valable, et l'on remonte alors vers une boucle *for* précédente où d'autres chiffres vont être essayés. Lorsqu'une des boucles for est terminée à un certain étage de l'arbre, sans donner de solutions, la case correspondant à l'étage est remise à 0, puisque la valeur qu'elle contient n'est pas valable. Même si cette valeur est écrasée par une nouvelle valeur dans une autre branche de l'arborescence, la remise à 0 est indispensable pour éviter d'avoir des résultats faux dans la fonction *possible*() que nous expliquons ci-dessous. Rappelons que dans un programme récursif, la fabrication de l'arborescence se fait en profondeur, jusqu'à un blocage ou à la solution, avec des boucles *for* en attente.



Dans le schéma ci-contre, il y a blocage à l'étage 3 (dans la case *d*[3]) : aucun des deux chiffres possibles n'a de successeur à l'étage suivant. On remet alors la case à 0 et on remonte vers la boucle *for* en attente au-dessus, celle-ci avance vers le deuxième chiffre de l'étage 2, et la descente repart …

Mais comment savoir quels sont les chiffres que l'on peut placer dans une case au cours des étapes du remplissage ? On utilise pour cela la fonction test *possible*(*n*,*L*,*C*) qui dit si oui ou non le chiffre *n* peut être placé dans la case de la ligne *L* et de la colonne *C*. Pour cela on prend d'abord tous les chiffres situés dans la ligne *L*, et si le chiffre *n* se trouve déjà dans cette ligne, la fonction ramène *NON* (0). Puis on fait de même avec les chiffres situés dans la colonne *C*. Enfin on prend tous les chiffres situés dans le carré 3 sur 3 de la case.



Les carrés 3x3 sont numérotés ainsi, de 0 à 8. Pour connaître le numéro *nc* du carré où se trouve la case *L*, *C*, on fait *nc*=3*(*L*/3)+*C*/3 (Vérifiez-le). Puis on parcourt les 9 cases de ce carré, numérotées par la variable *pdc* allant de 0 à 8. On connaît la ligne et la colonne de chacune de ces neuf cases en faisant :

ligne=3*(*nc*/3)+*pdc*/3 et *colonne*=3*(*nc*%3)+*pdc*%3.

Si l'on tombe sur un chiffre égal à *n*, là encore la fonction *possible*() ramène NON.

Mais si le chiffre *n* n'est ni dans la ligne, ni dans la colonne, ni dans le carré 3x3, la fonction ramène *OUI* (soit 1). On en déduit le programme de cette fonction *possible*()

```
int possible(int n,int L, int C)
{  int i, nc,pdc,ligne,colonne;
   for(i=0;i<9;i++) if (chiffre[L][i]==n) return 0;
   for(i=0;i<9;i++) if (chiffre[i][C]==n) return 0;
   nc=3*(L/3)+C/3;
   for(pdc=0; pdc<9; pdc++)
          { ligne=3*(nc/3)+pdc/3; colonne=3*(nc%3)+pdc%3;
            if (chiffre[ligne][colonne]==n) return 0;
          }
   return 1;
}

void arbre(int etage)
 {  int casearemplir,ligne,colonne,kk,k,nombre,b[9];
    if (etage==nbcasesaremplir)
      { nbsolutions++; dessin(); /* SDL_Flip(screen); pause(); si plusieurs solutions */
      }
   else
      {  casearemplir=d[etage]; ligne=casearemplir/9; colonne=casearemplir%9;
        k=0;
        for(nombre=1;nombre<=9;nombre++)
        if (possible(nombre,ligne,colonne)==1) b[k++]=nombre;
        for(kk=0;kk<k;kk++)
              {chiffre[ligne][colonne]=b[kk];
               arbre(etage+1);
              }
        chiffre[ligne][colonne]=0;
      }
 }
```

Finalement, le programme principal se contente d'appeler les fonctions précédentes. On la trouvera ci-dessous avec les déclarations associées.

```
#include <SDL/SDL.h>
#include <SDL/SDL_ttf.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#define xorig 20
```

```
#define yorig 20
#define pas 40

void conditionsinitiales(void);
void arbre(int etage);
int possible(int n,int L, int C);
void carre(int ligne, int colonne, int n, int couleur);
void dessin(void);
void pause(void);
void putpixel(int xe, int ye, Uint32 couleur);
void line(int x0,int y0, int x1,int y1, Uint32 c);
SDL_Surface * screen;
SDL_Surface *texte;  SDL_Rect position;TTF_Font *police=NULL;char chiffr[2000];
SDL_Color couleurnoire={0,0,0};
Uint32 white,black,couleurcase;
int a[81],chiffre[10][10],nbcasesaremplir,d[81],nbsolutions;

int main(int argc, char ** argv)
{   SDL_Init(SDL_INIT_VIDEO);
    screen=SDL_SetVideoMode(800,800,32, SDL_HWSURFACE|SDL_DOUBLEBUF);
    white=SDL_MapRGB(screen->format,255,255,255);
    couleurcase=SDL_MapRGB(screen->format,255,200,200);
    black=SDL_MapRGB(screen->format,0,0,0);
    SDL_FillRect(screen,0,color[0]);
   TTF_Init();  police=TTF_OpenFont("times.ttf",30);

  conditionsinitiales(); dessin();SDL_Flip(screen); pause();
  arbre(0);

   sprintf(chiffr,"Nombre de solutions : %d",nbsolutions);
   texte=TTF_RenderText_Solid(police,chiffr,couleurnoire);
   position.x=30; position.y=500; SDL_BlitSurface(texte,NULL,screen,&position);
   SDL_Flip(screen); pause(); return 0;
}
```

On a donné le résultat de l'exécution du programme sur un exemple, au début de ce texte, et la solution est unique. Par contre si l'on enlève le remplissage initial de la case numéro 78 (*a*[78]=3), on trouve dans ce cas 90 solutions. On en déduit comment fabriquer une grille de Sudoku : on commence par placer quelques chiffres dans des cases, en évitant toute incompatibilité, et l'on cherche toutes les solutions. Puis on rajoute un par un des chiffres dans des cases, jusqu'à ce que l'on trouve une solution unique.