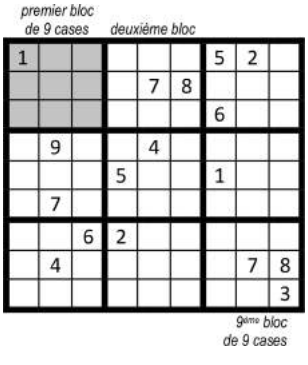
À quoi bon remplir des grilles de Sudoku ?

(Puisque l'ordinateur peut le faire)

Bernard Langer(*)

Introduction

Le jeu de Sudoku classique (il en existe beaucoup de variantes) consiste à compléter une grille carrée de 9 lignes et 9 colonnes divisée en 9 blocs de 9 cases.



Une grille de Sudoku complète doit vérifier un ensemble de règles détaillées ci-dessous.

Règles S :

1. Chaque ligne contient tous les chiffres de 1 à 9
2. Chaque colonne contient tous les entiers de 1 à 9
3. Chaque bloc de 9 cellules contient tous les entiers de 1 à 9

Au départ un certain nombre de chiffres sont dévoilés. A partir de là, il s'agit de compléter la grille en respectant les règles précédentes.
Dernière contrainte : le problème doit admettre une unique solution.

- En 2005 Bertram Felgenhauer et Frazer Jarvis(1) ont prouvé que le nombre de grilles est :

$$91 \times 72^2 \times 2^7 \times 27\,704\,267\,971 \approx 6{,}67 \times 10^{21}.$$

- En 2014 Gary McGuire(2) et son équipe ont démontré que la donnée de 16 chiffres dans une grille est insuffisante et qu'il faut donc dévoiler au moins 17 cases pour obtenir une solution unique.

L'objet de cet article ne concerne pas l'étude mathématique du Sudoku. Il se propose de montrer que certaines méthodes algorithmiques inenvisageables dans le cas d'une recherche manuelle sont pourtant fécondes lorsqu'on a recours à l'ordinateur. L'article est présenté sous forme d'exercices pouvant faire l'objet d'un ou de plusieurs TP.

(*) Bernard.langer@laposte.net
(1) http://www.afjarvis.staff.shef.ac.uk/sudoku/
(2) J.P Delahaye. Le problème du sudoku – Pour la Science – N° 447 – Janvier 2015.

Pour faciliter la mise en œuvre finale nous proposons le « squelette » d'un programme *python*, téléchargeable (ainsi qu'une solution complète) sur le site de l'APMEP à l'adresse : http://www.apmep.fr/Bulletin-515. Celui-ci pourra être ouvert avec *EduPython* par exemple. On trouvera également sur ce site une version plus sophistiquée reposant sur les procédures décrites dans les exercices qui suivent. Ce programme écrit en *JavaScript* s'exécute directement au sein du navigateur Internet et ne nécessite donc aucune installation. Il permet en particulier de contrôler l'unicité d'une solution…

Un algorithme de backtracking

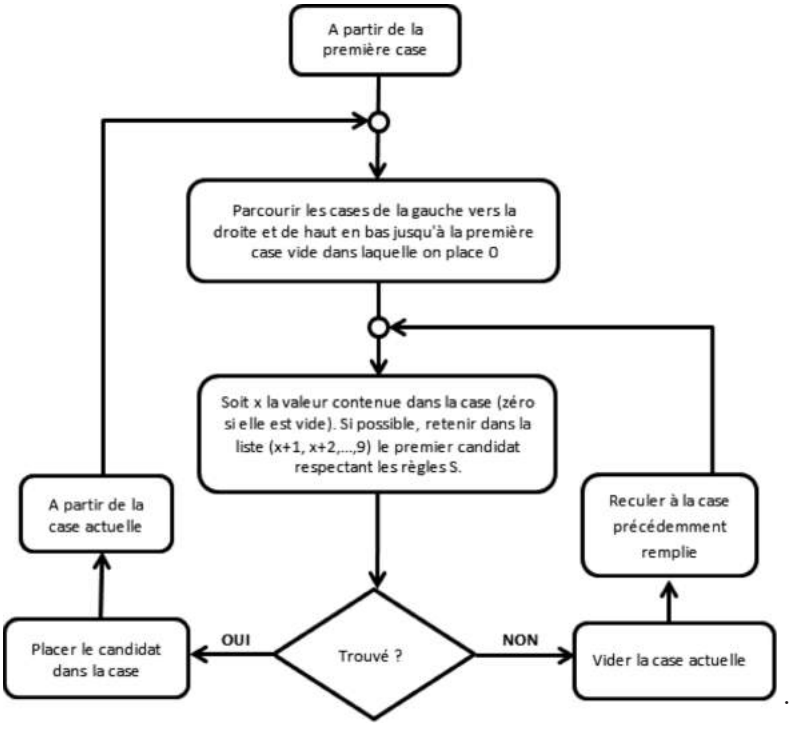
	0	1	2	3	4	5	6	7	8
0	3	5				2	7		
1	8	9		3					
2			2		8				6
3		2		1					4
4			9				6		
5	6					4		7	
6	9				4		1		
7								2	9
8			5	6	9	7		8	3

La grille initiale

Considérons la grille ci-dessus. L'idée simple que nous allons mettre en œuvre repose sur un principe appelé *backtracking* (retour sur trace). Il consiste ici à balayer la grille en démarrant en ligne 0, colonne 0, *de la gauche vers la droite* et *du haut vers le bas* en plaçant dans les cases vides rencontrées un entier respectant les règles S… En cas de succès on passe à la case suivante et en cas d'impossibilité on vide la case puis on recule à la case précédemment remplie en y plaçant si possible une autre valeur candidate… avant de poursuivre la marche en avant. En cas de nouvel échec on recule, etc.

On obtient une solution lorsque la dernière case vide est remplie. S'il faut reculer en deçà de la première case vide le problème n'a pas de solution…

En confondant « cases vides » et « cases contenant zéro » et en formalisant davantage les choses on peut décrire le processus de recherche par le schéma suivant :



Pour ne pas alourdir la description nous avons volontairement omis de signaler les conditions d'arrêt :

- le processus s'arrête lorsqu'on ne peut plus reculer ; dans ce cas le problème n'a pas de solution.
- le problème est résolu lorsque la dernière case est remplie.

Exercice 1 : Mise en œuvre sur un exemple

À l'aide du document disponible sur le site http://www.apmep.fr/Bulletin-515, mettre en œuvre *manuellement* l'algorithme décrit ci-dessus à partir de la grille précédente. (Nous vous conseillons cependant de ne pas aller au-delà de la deuxième ligne de la grille…).

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

Les lignes et colonnes sont numérotées de 0 à 8 de la gauche vers la droite et du haut vers le bas

Nous adopterons la notation LiCj pour désigner la case située en ligne i, colonne j.

L'annexe disponible sur le site de l'APMEP permettra au lecteur de s'exercer sans abîmer le bulletin.

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

fig.1

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

fig.2

- La première case vide est *L0C2*. On peut y placer 1 puisque cette valeur ne se trouve ni en ligne 0, ni en colonne 2, ni dans le premier bloc.
- On passe ensuite à la case *L0C3*. Le premier candidat est 4 puisque 1 se trouve en *L0C2*, 2 est en *L0C5* et 3 en *L1C3*.
- 6 est le premier candidat pour *L0C4*.
- 9 est le premier candidat pour *L0C7*
- Et 8 peut se placer en L0C8… Ça se passe plutôt bien !
- Poursuivons en traitant la deuxième ligne. Le démarrage s'effectue comme sur des roulettes : 4 en *L1C2*, 1 en *L1C4*, 5 en *L1C5*, 2 en *L1C6* … jusqu'en *L1C7* (fig. 1) où il n'y plus aucun candidat : 1, 2, 3, 4, 5 figurent déjà en ligne 1 et 6, 7, 8, 9 se trouvent dans le bloc.
- On recule donc d'une case en *L1C6* et on reprend la recherche à partir de 3… aucune possibilité. On efface *L1C6* et on recule en *L1C5*.
- Aucune possibilité à partir de 6, on efface *L1C5* et on recule en *L1C4* où l'on démarre la recherche à partir de 2.

  5 se porte candidat et on reprend la marche en avant…
- etc.
- Après quelques pas de tango le lecteur patient parviendra à compléter la deuxième ligne comme présenté en (fig. 2). Nous lui conseillons vivement de ne pas poursuivre… D'autant plus que même la première valeur, celle placée en *L0C2*, n'est pas correcte…

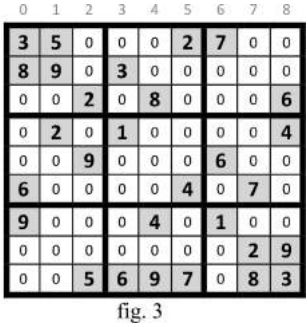
Pourtant cette démarche particulièrement irréaliste pour un être humain va se révéler tout à fait acceptable pour un ordinateur.

Exercice 2 : Représentation des données

Plusieurs structures de données peuvent être utilisées pour construire les divers algorithmes (on trouvera une alternative en *Annexe 1*). Néanmoins le recours aux *listes* s'est révélé particulièrement fructueux.
D'un point de vue informatique, une liste est une *suite finie* sur laquelle sont définies un certain nombre d'opérations telle par exemple l'*adjonction en queue* que nous noterons $adjq(l,x)$, qui consiste à ajouter l'élément x en queue de la liste l. Dans la suite nous noterons $l[i]$ l'élément de rang i de la liste l.

Nous représenterons la grille de Sudoku sous forme d'une liste de 81 éléments correspondants aux différentes cases de la grille. Cette liste est obtenue en rangeant à la queue leu leu les éléments du tableau parcouru de la gauche vers la droite et du haut vers le bas, les cases vides étant remplacées par des zéros. Le premier élément,

de rang 0, correspond à la case *L0C0* et le 81e à la case *L8C8*.
Dans toute la suite, *grille* désignera la grille de Sudoku en cours de traitement. Nous supposerons cette grille accessible par tous les algorithmes qui suivent.



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

fig. 3

rang	0	1	2	3	4	5	6	7	8	9	10	11	12	
grille	3	5	0	0	0	2	7	0	0	8	9	0	3	...

fig. 4

L'élément situé en *L1C3* de la grille occupe le rang 12 dans la liste. Nous adopterons les notations suivantes :

- *l* : numéro de la ligne de l'élément de rang *p* de la liste.
- *c* : numéro de la colonne de l'élément de rang *p* de la liste.
- *lc* : numéro de la ligne du coin supérieur gauche du bloc contenant la case de rang *p* de la liste.
- *cc* : numéro de la colonne du coin supérieur gauche du bloc contenant la case de rang *p* de la liste.

Q1 : Déterminer le rang *p* dans la liste *grille* de l'élément situé en ligne *l* colonne *c* du tableau en fonction de *l* et *c*.
Q2 : Déterminer *l* et *c* en fonction de *p*.
Q3 : Déterminer *lc* et *cc* en fonction de *l* et de *c*.

Rép : En notant *quotient*(*a*,*b*) le quotient de la division entière de *a* par *b* et *reste*(*a*,*b*) le reste de cette division :
Q1 : $p = 9 \times l + c$.
Q2 : $l = quotient(p,9)$; $c = reste(p,9)$.
Q3 : $lc = 3 \times quotient(l,3)$; $cc = 3 \times quotient(c,3)$.

Exercice 3 : Chaînages

Afin de faciliter la circulation (avancer ou reculer) parmi les emplacements libres de la liste grille en évitant les cases initialement remplies nous mettons en place un *chaînage* entre les éléments *nuls* de cette liste. Pour cela, nous utiliserons deux autres listes notées *prec* et *suiv* (pour précédant et suivant) qui à chaque élément de *grille* associent le rang de l'élément précédent ainsi que le rang de l'élément suivant en « sautant » les éléments initialement dévoilés. Par exemple dans la grille *fig.4*, *suiv*[4] = 7 et *prec*[11] = 8. Nous conviendrons que la variable *prem* désigne le rang de la première case vide (dans notre exemple, *prem* = 2). Puisque le précédent du premier n'existe pas, nous conviendrons que *prec*[*prem*] = –1. De manière analogue, nous décidons que le suivant de la dernière case vide a pour rang –2. Ces conventions permettent de différencier les terminaisons possibles de l'algorithme : lorsque la valeur de la case courante est (–1) il n'y a pas de solution ; si cette valeur est (–2) une solution a été trouvée.
La grille de Sudoku est donc codée par l'entier *prem* et les listes *grille*, *suiv* et *prec*.

Dans toute la suite nous supposerons que ces variables ont une portée globale c'est-à-dire qu'elles sont accessibles par tous les algorithmes partiels.

prem=2

rang	*prec*	*grille*	*suiv*
0		**3**	
1		**5**	
2	-1	**0**	3
3	2	**0**	4
4	3	**0**	7
5		**2**	
6		**7**	
7	4	**0**	8
8	7	**0**	11
9		**8**	
10		**9**	
11	8	**0**	13
…	…	**…**	…
73	72	**0**	78
74		**5**	
75		**6**	
76		**9**	
77		**7**	
78	73	**0**	-2
79		**8**	
80		**3**	

.

Q : Écrire un algorithme, noté *InitGrille* qui construit les listes *prec* et *suiv* à partir de la liste grille.

Rép :

InitGrille

i, j : entiers : compteurs de boucles.

prec, suiv, grille : listes : définies plus haut

```
Début InitGrille
prec=[ ]
suiv=[ ]
pour i de 0 à 80 répéter
    prec = adjq(prec, 0)
    suiv = adjq(suiv, 0)
i=0
tant que (grille[i] ≠ 0)répéter
    i=i+1
prem=i
prec[i]= -1
tant que (i < 81)
    j=i+1
    tant que (grille[j] ≠ 0 et j < 81)répéter
        j=j+1
    si (j = 81) suiv[i] = −2
        sinon suiv[i] = j
              prec[j] = i
    i=j
fin InitGrille
```

Rappelons que l'opération *adjq*(*l*,*x*) consiste à ajouter l'élément *x* en *queue* de la liste *l*.

Exercice 4 : Les règles S

Dans cet exercice, il s'agit de construire une fonction booléenne qui retourne *vrai* lorsque la valeur *v* placée en rang *p* de la liste *grille* respecte les règles S et *faux* dans le cas contraire.
En d'autres termes, il faut exiger que la valeur *v* ne figure ni dans la ligne ni dans la colonne et ni dans le bloc la contenant.

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

L'idée est la suivante : pour traiter la case située en ligne *l*, colonne *c* on construit une liste auxiliaire nommée *listAux* contenant toutes les valeurs de la ligne *l*, toutes les valeurs de la colonne *c* et toutes les valeurs du bloc. Pour être candidat, l'entier *n* ($1 \leq n \leq 9$) ne devra pas figurer dans *listAux*.

Intéressons-nous par exemple à la case *L2C1*. *listAux* contient tous les éléments de la ligne 2, tous les éléments de la colonne 1 et tous les éléments du premier bloc nous obtenons :

listAux = [0,0,2,0,8,0,0,0,6,5,9,0,2,0,0,0,0,0,3,5,0,8,9,0,0,0,2]

Un candidat pour la case *L2C1* ne devant pas figurer dans *listAux*, les candidats sont : 1, 4, 7.

Q1 : Écrire une fonction *ListeAuxiliaire*(*p*) qui retourne la liste *listAux* définie ci-dessus.
Q2 : Écrire une fonction *ChercheCandidat*(*c*) qui retourne le premier candidat possible pour la case de rang *c* de la liste grille autre que la valeur actuelle ou bien 0 en cas d'impossibilité.

Rép :

Q1 :
Fonction *ListeAuxiliaire*(*p*)

l, *c*, *lc*, cc : entiers définis dans l'exercice 2.
p : entier : rang de l'élément
pp : entier : rang du coin supérieur gauche du bloc contenant l'élément de rang *p*
listAux : liste : la liste auxiliaire

Attention : pour éviter la succession de trois boucles, nous les avons regroupées en une seule. Dans l'exemple précédent est générée sous la forme :

```
Début ListeAuxiliaire(p)
l = quotient(p, 9)
c = reste(p, 9)
lc = 3 × quotient(l, 3)
cc = 3 × quotient(c, 3)
pp = 9 × lc + cc
listeAux = [ ]
Pour i de 0 à 8 répéter
    k1 = 9 × i + c
    k2 = 9 × l + i
    k3 = pp + 9 × quotient(i, 3) + reste(i, 3)
    listAux = adjq(listAux, grille[k1])
    listAux = adjq(listAux, grille[k2])
    listAux = adjq(listAux, grille[k3])
résultat = listAux
fin ListeAuxiliaire
```

listAux = [5,0,3,9,0,5,0,2,0,2,0,8,0,8,9,0,0,0,0,0,0,0,0,0,0,6,2]
Seul l'ordre des éléments diffère.

Q2 :

Fonction *ChercheCandidat*(*c*)

c : *entier* : rang de la case

v : *entier* : valeur courante

ok : *booléen* : vrai dès qu'on a trouvé un candidat

```
début ChercheCandidat(c)
listAux = ListeAuxiliaire(c)
v = grille[c]
ok = faux
tant que (non ok) et (v < 9)
    v = v + 1
    ok = (v ∉ listAux)
si (non ok) alors v = 0
résultat = v
Fin ChercheCandidat
```

Exercice 5 : Sudoku

En supposant que la grille initiale est donnée (la saisie n'est pas à faire) et est cohérente (les règles S sont vérifiées), construire l'algorithme final qui résout le problème de Sudoku.

Rép :

Sudoku

c : *entier* : rang courant dans la liste grille.
suiv, *prec*, *grille* : *listes* : définies plus haut

```
Début Sudoku
c = prem
tant que c ≥ 0 répéter
    grille[c] = ChercheCandidat(c)
    si (grille[c] = 0) alors c = prec[c]
                       sinon c = suiv[c]
si (c = −1) alors afficher "Cette grille n' a pas de solution"
            sinon afficher la grille
fin Sudoku
```

La page Sudoku.html disponible sur le site de l'APMEP repose sur les algorithmes précédents mais comporte quelques extensions qui peuvent se révéler utiles :

- Saisie des données.
- Contrôle de cohérence.
- Affichage sophistiqué.
- Contrôle de l'unicité de la solution.

Pour ne pas alourdir cet article, les algorithmes correspondants à ces extensions ne sont pas explicités mais ils peuvent faire l'objet d'exercices complémentaires (Un complément disponible sur le site de l'APMEP donne quelques pistes.)

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

Voici une grille particulièrement retorse destinée à éprouver les algorithmes précédents.

69 175 316 valeurs ont été « essayées » par l'ordinateur avant de parvenir à la solution ce qui pourrait constituer une réponse au titre de cet article.

Temps d'exécution du programme

Internet explorer version 11	Google Chrome version 40	Mozilla FireFox version 35
1 min 24 s	**2 min 38 s**	**4 min 47 s**

Curieusement lorsqu'on remplace la grille précédente par sa symétrique selon l'axe vertical, le temps d'exécution chute de manière spectaculaire en passant à quelques millisecondes ! L'explication est simple… La première ligne de la solution est : 9-8-7-6-5-4-3-2-1. Or la première valeur testée en *L0C0* étant 1, il faudra quelques

millions de retours arrière avant de trouver 9… Alors que pour la grille symétrique, la première ligne de la solution est 1-2-3-4-5-6-7-8-9 c'est-à-dire que les 9 premières valeurs testées sont les bonnes, ce qui divise le nombre de retours arrières par 1600 !

Remarque : Il peut arriver que le programme s'arrête sans être arrivé à terme. Dans ce cas il faut cliquer sur un bouton « Attendre » ou « Autoriser le contenu bloqué ». Ce comportement est dû au moteur *JavaScript* du navigateur utilisé. Par ailleurs on peut constater que les temps d'exécution varient du simple au triple.

Sitographie **:**

http://fr.wikipedia.org/wiki/Sudoku
http://fr.wikipedia.org/wiki/Mathématiques_du_Sudoku
http://en.wikipedia.org/wiki/Backtracking

ANNEXE 1 : Quelle structure de données ?

Choix de la structure de données

Les algorithmes précédents utilisent la structure de liste. Ce choix sans doute arbitraire m'a paru être particulièrement bien adapté à la réflexion initiale : (avancer ou reculer d'une cellule en « sautant » celles déjà remplies).

La notion de liste est largement utilisée et est fondamentale en informatique. Le langage Lisp, par exemple, repose entièrement sur cette structure de données.

La première idée venant à l'esprit lorsqu'il s'agit de travailler sur une grille de Sudoku est bien sûr d'avoir recours à un tableau d'entiers de 9 lignes et 9 colonnes. Un tel tableau permet de manipuler individuellement les diverses cellules de la grille en écrivant par exemple $G[3,6]$ pour désigner la cellule se trouvant à la ligne de rang 3 et colonne de rang 6.
Le parcours de ce tableau de la gauche vers la droite et du haut vers le bas se faisant simplement grâce à deux boucles imbriquées :

```
pour l de 1 à 9 répéter
    pour c de 1 à 9 répéter
        traitement
```

Néanmoins, le traitement devient plus délicat lorsqu'il s'agit d'avancer ou de reculer d'une case tout en « évitant » les cases initialement dévoilées. Cela reste bien sûr possible. On pourrait par exemple utiliser un second tableau *figé* (noté *GS* dans l'encadré ci-dessous) ne contenant que les valeurs de départ et zéro dans les autres cellules. En supposant que la cellule courante soit $G[l,c]$ avancer à la cellule suivante se traduirait alors par :

```
c = c + 1
si c > 9 alors  c = 1
                l = l + 1
tant que GS[l, c] ≠ 0 répéter
        c = c + 1
        si c > 9 alors c = 1
                       l = l + 1
```

Pour reculer il faudra procéder de manière analogue.

L'usage des listes chainées semble plus efficace, il faut bien sûr effectuer les calculs de chainages mais on ne le fait qu'une seule fois !

Nous laissons le lecteur juge de la pertinence de l'une ou l'autre structure.