Cet article est rédigé par des élèves. Il peut comporter des oublis ou des imperfections, autant que possible signalés par nos relecteurs dans les notes d'édition.

Sudoku

2016-2017

Nom, prénom et niveaux des élèves : Gaspard Misery, Julien Biancucci, Paulin Roman, Claudia Ingrazia, élèves de première SSI/ES.

Établissement : Lycée du Mont-Blanc René-Dayve

Enseignant(s) : Hélène Langlais, Cyril Masson

Chercheur(s) : Martin Gander, Université de Genève; Pierre-Alain Cherix, Université de Genève

Table des matières

Résumé

Études autour du sudoku : résolution d'un sudoku par un algorithme et création de matrices de grilles de sudoku aléatoires (2 versions). L'objectif est de tester l'algorithme de résolution sur les grilles générées aléatoirement, et de compter le nombre de solutions obtenues pour chacune des grilles. On cherche à savoir s'il existe une valeur (pour le nombre de chiffres pré remplis) à partir de laquelle la grille n'a qu'une seule solution. Les algorithmes présentés ici ont été écrits par les élèves, avec l'aide des chercheurs et des professeurs, en Scilab.

1 Introduction

Voici donc la synthèse de nos études sur les sudokus. D'abord qu'est-ce qu'un *sudoku*? C'est une grille de 9 cases sur 9 que l'on doit remplir avec des chiffres de 1 à 9, et on ne doit pas mettre deux fois le même chiffre dans une même colonne, une même ligne ou dans un des 9 carrés de 9 cases (3×3) que comporte la grille. Il existe donc plusieurs stratégies de résolutions manuelles de sudoku, par exemple le fait de chercher les chiffres manquants dans une ligne et voir si il est possible de le placer dans une des cases en prenant le soin de vérifier que le même chiffre n'est pas présent dans le carré ou dans l'autre ligne correspondante.

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

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

Il faut donc procéder ainsi ou d'une autre façon en respectant les règles du Sudoku jusqu'à obtention d'une grille intégralement remplie. La difficulté varie selon la taille des sudokus, le nombre de chiffres qu'ils contiennent mais essentiellement selon la disposition des chiffres. Les niveaux de difficulté varient de facile, en passant par moyen, puis difficile et même expert ou démoniaque. Certaines personnes résolvent des sudoku en peu de temps mais un ordinateur sera toujours plus rapide : l'algorithme met environ 4 secondes. De plus le fait de rajouter des lettres dans un sudoku déstabilise même les plus doués en sudoku [1] .

Le mot *sudoku* vient du japonais "Suji wa dokushi ni kagiru" et "Su" veut dire chiffre et "doku" veut dire unique. Info : L'une des plus anciennes grilles de sudoku française connue à été publié dans le quotidien "La France" en 1895. Nous avons donc cherché à résoudre des grilles de Sudoku, à générer nos propres grilles, puis nous avons fait des statistiques sur les grilles pour savoir à partir de combien de cases pré-remplies une seule solution est trouvée [2] . Nos algorithmes ont tous été fait sur le logiciel Scilab disposant de son propre langage éponyme.

2 Résolution

Le but premier était de créer un algorithme capable de résoudre tous les Sudoku quelle que soit leur forme, et le nombre de chiffres qu'ils contiennent. Les chercheurs nous ont donné la piste d'un fonctionnement récursif [3] . Ainsi on rentre le Sudoku sous forme de matrice dans le logiciel en symbolisant les cases vides par des 0.

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

Pour résoudre un sudoku, nous avons écrit la fonction Sudoku(M) (Voir annexe A), M étant la matrice associée au sudoku. C'est une fonction récursive, que nous avons écrit avec l'aide des chercheurs.

- On demande de trouver les zéros dans la matrice. Si on trouve des zéros, on rentre alors dans la boucle :
  - on travaille sur la première case vide (ligne i, colonne j), et on teste les chiffres de 1 à 9 (variable k) un par un, successivement.
  - on vérifie grâce à la fonction (nommée MoveIsPossible, voir annexe A) qu'on peut placer k dans la case (i, j). La fonction MoveIsPossible renvoie 1 lorsqu'on peut placer k en (i, j), c'est à dire lorsque les règles du sudoku sont respectées, et 0 sinon.
  - si on peut placer k en (i, j), on remplit la case (i, j) de la matrice M avec cette valeur k et on relance la fonction Sudoku avec la nouvelle matrice M. [4]
  - si on ne peut pas, on teste le chiffre suivant [5] .
- et enfin, quand il ne reste plus de case vide, l'algorithme affiche la matrice trouvée.

Une fois qu'une solution est trouvée, l'algorithme ne s'arrête pas : Comme on teste toutes les valeurs de 1 à 9 sur toutes les cases vides, l'algorithme trouve toutes les solutions. Le fonctionnement récursif de la fonction Sudoku(M) permet de trouver toutes les solutions avec la seule variable M [6] .

3 Génération aléatoire d'une grille de Sudoku

Notre second objectif était de créer aléatoirement une grille de Sudoku pour ensuite pouvoir la résoudre. Il y a deux possibilités pour générer une grille de Sudoku :

- les grilles créées à partir de rien en ajoutant des chiffres (complètement aléatoire).
- les grilles faites en retirant des chiffres sur une grille qui existe déjà (résultat assuré).

Nous avons d'abord testé la première possibilité ; la fonction MR(n), pour Matrice Random, crée des des grilles en ajoutant n chiffres à partir d'une grille vide (voir annexe B). Mais chaque chiffre placé doit l'être en respectant les règles du Sudoku. C'est à nouveau la fonction MoveIsPossible qui assure cet objectif. Cependant cette création aléatoire ne permet pas toujours d'obtenir une grille remplie avec n chiffres, notamment lorsque n est grand. Parfois, aucun chiffre ne peut être placé dans la case (l,c), car quel que soit le chiffre placé, les règles du sudoku ne sont plus respectées. On teste donc jusqu'à 100 valeurs (en supposant qu'avec 100 essais, on aura testé au moins 1 fois chacun des chiffres de 1 à 9), puis on abandonne en indiquant "no matrix found"

Exemple de grille avec la case (1,7) qu'on ne peut pas remplir ;

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

La grille respecte jusqu'à présent les règles du sudoku, mais aucun chiffre ne peut être placé à la place du point d'interrogation.

Lorsque n est grand (à partir de 40), il est difficile de générer une grille de cette façon. Car l'algorithme bloque sur une case qu'il ne peut pas remplir. Pour surmonter cette difficulté, nous avons cherché un autre moyen pour générer des grilles. Nous avons utilisé une grille complètement remplie, et nous enlevons aléatoirement $81-n$ chiffres pour obtenir une grille à n chiffres pré-remplis. L'intérêt de cette méthode est que l'on est sûr d'obtenir une grille, et on est également sûr qu'elle aura au moins une solution. La fonction StatSudoku(n,k) permet ainsi de générer n grilles pré remplies avec k chiffres (voir annexe B) [7] . Cette fonction enregistre également le nombre de solutions trouvées par l'algorithme de résolution, et ce pour chacune des matrices générées.

Nos algorithmes de génération ne nous donnaient pas une grille avec une solution unique. Les grilles avaient la plus part du temps plusieurs solutions possibles, nous avons donc créé

un autre algorithme compteur (SudokuSol) pour déterminer le nombre de solutions (celui-ci sera expliqué au paragraphe suivant).

4 Statistiques

Une fois en possession des algorithmes permettant la création et la résolution de Sudoku, nous nous sommes penchés sur la question "À partir de combien de chiffres pré-remplis un Sudoku ne possède plus qu'une seule solution ?" [8]

Nous avons donc créé des nouveaux programmes pour déterminer le nombre moyen de solutions en fonction du nombre de chiffres pré remplis, et cela pour chacun des deux modes de génération de matrices.

Dans la suite,

- k est le nombre de chiffres pré-remplis que l'on veut dans nos grilles ;
- n est le nombre de grilles ou d'essais que l'on veut faire.

La fonction stat(n, k) permet ainsi de créer n matrices aléatoires (avec la fonction MR(k)), puis, pour chaque matrice, on appelle la fonction sudokuSol qui compte le nombre de solutions obtenues (Voir annexe C) [9]

La fonction statSudoku(n, k) fait exactement la même chose, mais lorsque la matrice est générée à partir d'une grille pleine dans laquelle on enlève $81 - k$ valeurs.

Nous avons ensuite tracé les graphiques indiquant le nombre moyen de solution en fonction du nombre de chiffre pré remplis, pour chacun des deux modes de génération. Pour cela, nous avons lancé les deux fonctions stat(n, k) et statSudoku(n, k) avec $n = 1000$, et k allant de 27 (ou 31) à 40 (50). Et pour chacun de ces essais, nous avons calculé le nombre moyen de solutions obtenues. Les graphiques sont en annexe D.

On peut donc dire que les deux méthodes de génération de matrices donnent un résultat équivalent.

5 Conclusion

Pour conclure, nous avons réalisé un algorithme pour résoudre les grilles de Sudoku, il est capable de nous afficher la ou les solution(s) de la grille. En deuxième partie, nous avons réalisé deux autres algorithmes pour générer aléatoirement une grille de Sudoku, grâce à ces algorithmes, nous avons effectué un grand nombre de simulations afin d'obtenir des moyennes pour savoir à partir de combien de chiffres pré-remplis une grille de Sudoku n'a qu'une seule solution. D'après les graphiques que nous obtenons, c'est aux environs de 37 chiffres pré remplis que le nombre de solutions est proche de 1. Tout cela nous a mené à des résultats intéressants car nous pensions qu'il en faudrait beaucoup moins. Peut-être existe-t-il une propriété mathématique sur les matrices capable de prédire ce nombre, sinon, une nouvelle piste de recherche vient peut-être de s'ouvrir...

6 Annexes

A Résolution

```
function Sudoku(M)
    [i,j]=find(M'==0) // On cherche les 0 de la matrice M, on enregistre
leurs positions dans les matrices lignes i et j
    if length(i)>0 then // s'il y a encore des 0 dans la matrice
        i=i(1); j=j(1); //Num de ligne et colonne de la 1ère case vide
        for k=1:9
            if MoveIsPossible(M,i,j,k)==1 then // si on peut placer k en
(i,j)
                M(i,j)=k; Sudoku(M); // on remplit la case (i,j) avec k,
on relance l'algo avec la nelle matrice
            end
        end
    else
        disp("solution found:",M) // affichage de la solution trouvée
    end
endfunction

function trueorfalse=MoveIsPossible(M,i,j,k) // Peut-on placer k dans la
case (i,j) de M: si c'est possible, trueorfalse=1, et 0 sinon
    trueorfalse=1 // variable initialisée à 1: on peut placer k en (i,j)
    for l=1:9
        if M(l,j)==k then // si on trouve k sur la colonne j
            trueorfalse=0 //  la variable trueorfalse prend la valeur 0
        end
        if M(i,l)==k then // si on trouve k sur la ligne i
            trueorfalse=0  //  la variable trueorfalse prend la valeur 0
        end
    end
    A=DefinitionA(i,j,M) //A est la matrice constituée du petit carré
contenant la case (i,j)
```

[10]

```
    [l,c]=find (A==k) // recherche de k dans ce petit carré
    if length(l)>0 then // Si on a trouvé k dans ce petit carré
        trueorfalse=0 // la variable trueorfalse prend la valeur 0
    end
endfunction
```

B Génération aléatoire d'une grille de Sudoku

Première méthode pour générer une matrice aléatoire

```
function M=MR(n)  // Génération aléatoire d'une grille à n chiffres pré
remplis
    count=0 // initialisation d'un compteur de tentatives à 0
    M=zeros(9,9) // initialisation de la matrice avec des 0
    for i=1:n // n boucles, qui placent chacune un chiffre
        y=int(rand()*9)+1 // tirage aléatoire du chiffre à placer
        l=int(rand()*9)+1 // tirage aléatoire de la ligne
        c=int(rand()*9)+1 // tirage aléatoire de la colonne
        while M(l,c)<>0 then //si la case (l,c) est non vide
            l=int(rand()*9)+1 //tirage d'un autre numéro de ligne
            c=int(rand()*9)+1 //tirage d'un autre numéro de colonne
        end
        while MoveIsPossible(M,l,c,y)==0 & count<100 then //Tant que y ne
 peut pas être placé en (l,c)
            y=int(rand()*9)+1  //tirage d'un autre chiffre à placer
            count=count+1 //compteur de tentatives
        end
        if count==100 then disp("NoMatrixFound") // si 100 tentatives
infructueuses
            else M(l,c)=y ; count=0 //on place y dans la case (l,c)
        end
    end
    // disp(M)
endfunction
```

Deuxième méthode pour générer une matrice aléatoire

```
function N=statSudoku(n,k)
    // n nb de répétition
    // k nb de chiffres dans la matrice
    global NbSol
    N=[] // Liste pour enregistrer le nombre de solutions obtenues
    for i=1:n
M=[1.,2.,3.,7.,5.,6.,4.,8.,9.;7.,5.,6.,4.,8.,9.,1.,2.,3.;9.,8.,4.,1.,2.,3
.,7.,6.,5.;4.,3.,5.,8.,6.,7.,2.,9.,1.;2.,6.,7.,5.,9.,1.,8.,3.,4.;8.,9.,1
.,2.,3.,4.,6.,5.,7.;3.,4.,9.,6.,7.,8.,5.,1.,2.;5.,7.,8.,9.,1.,2.,3.,4.,6
.;6.,1.,2.,3.,4.,5.,9.,7.,8.] // Initialisation de la matrice initiale
        for j=1:81-k //Il faut supprimer 81-k chiffres
            [l,c]=find(M<>0) // recherche des cases non nulles
            r=int(rand()*length(l))+1 // Tirage aléatoire d'une de ces
cases
            M(l(r),c(r))=0 // Mise à 0 de la case tirée au sort
        end
        sudokuSol(M) // Résolution de la matrice avec comptage du nombre
de solution
        N=[N,NbSol] // Ajout du nombre de solutions trouvées à liste N
    end
    disp (N)
endfunction
```

C Statistiques

Nombre de solutions avec génération via la fonction MR(k)

```
function N=stat (n,k)
    // n nb de répétition
    // k nb de chiffres dans la matrice
    global NbSol // compteur du nombre de solution, variable globale
utilisée dans plusieurs fonctions
    N=[] // Liste des nombres de solutions pour les n matrices testées

    for i=1:n
        M=MR(k) // Génération d'une matrice aléatoire avec k chiffres
        [i,j]=find(M'==0) //Identification des 0 de la matrice
        if length (i)==81-k then // si la matrice a bien k chiffres pré
remplis
```

```
            sudokuSol (M) // Recherche des solutions avec mémorisation du
 nombre de solutions
            N=[N,NbSol] //ajout du nombre de solutions trouvées dans N
        end
    end
    disp (N) // Affichage de N
endfunction
```

Compteur de solutions

```
function sudokuSol(M)
    global NbSol  //NbSol variable globale de comptage
    NbSol=0 // Initialisation à 0
    Sudokuv2(M) //Fonction identique à sudoku, mais avec comptage du
nombre de solution
    disp(NbSol)
endfunction
```

Résolution avec compteur de solutions

```
function Sudokuv2(M)
    global NbSol
    [i,j]=find(M'==0)
    if length(i)>0 then
        i=i(1); j=j(1);
        for k=1:9
            if MoveIsPossible(M,i,j,k)==1 then
                M(i,j)=k; Sudokuv2(M);
            end
        end
    else
        NbSol=NbSol+1
        disp("solution␣found:",M,NbSol)
    end
endfunction
```

D Graphiques

Méthode 1 : génération aléatoire

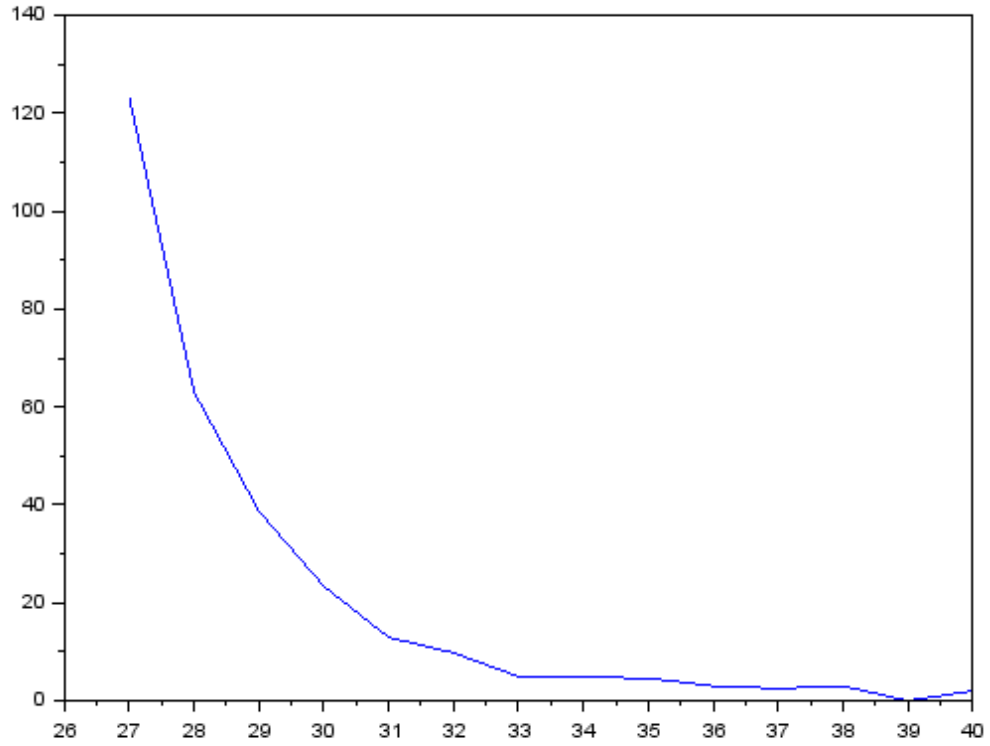

FIGURE 1 – Nb moyen de solutions en fonction du nb de chiffres pré-remplis méthode 1

Méthode 2 : génération par suppression de chiffres dans une matrice complète

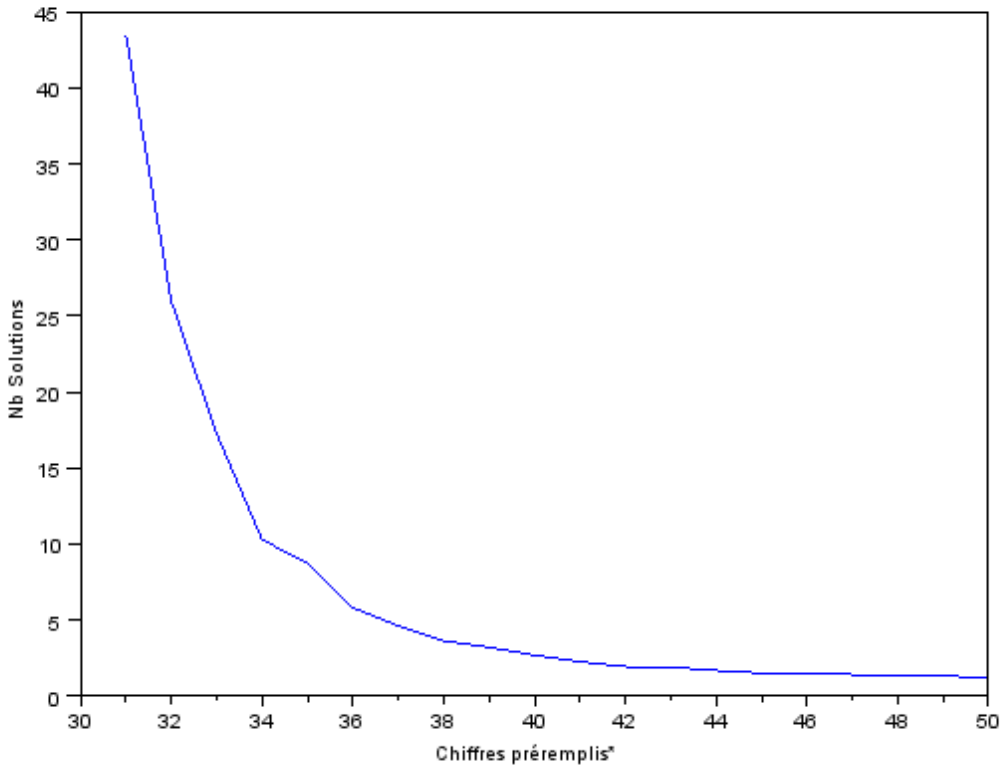

FIGURE 2 – Nb moyen de solutions en fonction du nb de chiffres pré-remplis méthode 2

Méthode 1 et méthode 2

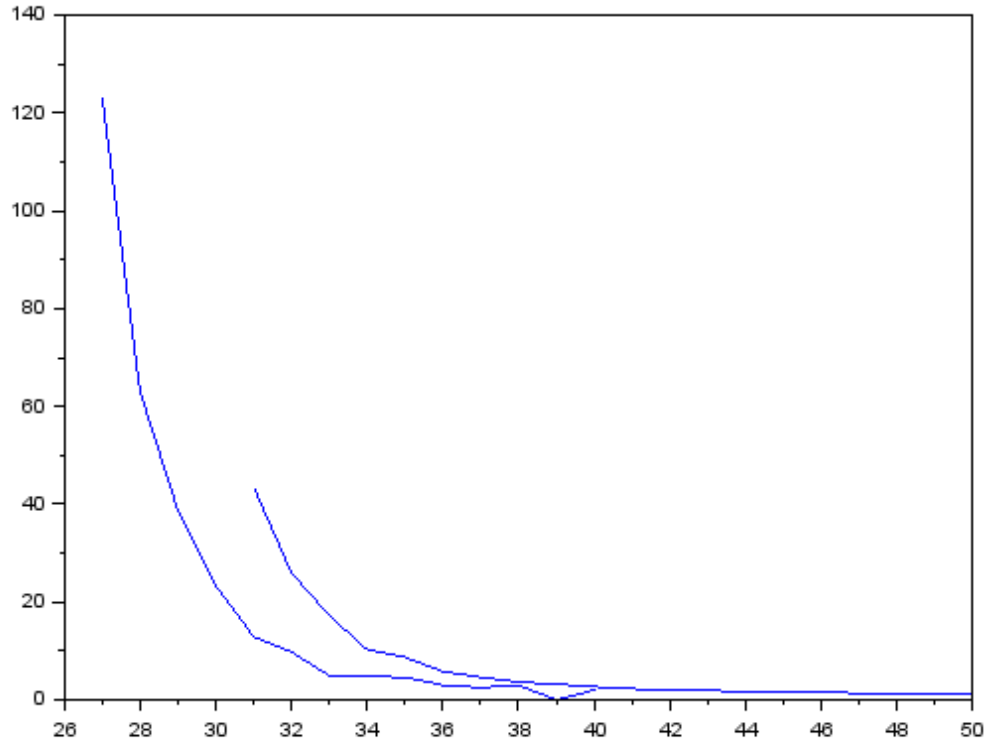

FIGURE 3 – Nb moyen de solutions en fonction du nb de chiffres pré-remplis méthode 1 et 2

Notes d'édition

[1] Les relecteurs n'ont pas compris ce que signifie“rajouter des lettres dans un sudoku".
[2] Cette formulation laisse penser qu'il y a toujours au moins une solution et que seul le nombre de cases pré-remplies est important pour qu'il n'y en ait qu'une, ce qui n'est pas le cas ; en fait c'est un nombre moyen qui sera évalué.
[3] Un algorithme récursif est un algorithme qui fait appel à lui-même, comme dans l'exemple type du calcul de la factorielle : *si* $n = 0$ *factorielle*$(n) = 1$, *sinon* *factorielle*$(n) = n \times$ *factorielle*$(n-1)$. À l'exécution, on a une suite d'appels emboîtés à la fonction elle-même qui s'arrêtent ici lorsque $n = 0$; alors chacune des fonctions en action retourne une valeur à celle qui l'a appelée, permettant le calcul de proche en proche.
[4] La matrice passée en paramètre dans l'appel récursif a une case vide de moins. On ne pourra donc pas avoir plus d'appels récursifs emboîtés qu'il n'y a de cases vides au départ. Si on arrive à remplir toutes les cases vides en respectant les règles, on obtient une solution qui sera affichée au dernier appel emboîté. Il peut aussi y avoir moins d'appels emboîtés si à une certaine étape on ne peut plus placer aucun chiffre dans la case vide sélectionnée : ceci se produira pour les grilles sans solutions, et pour les fausses pistes où les chiffres testés ne donnent pas de solutions.
[5] En fait, le programme cherche toutes les solutions de la grille et teste le chiffre suivant dans tous les cas, sauf si on est arrivé à $k = 9$: alors la boucle est terminée et le programme retourne à la fonction qui a appelé celle qui est en action, ou s'arrête si on est au premier niveau.
Le fait que la matrice ait été modifiée n'a pas d'incidence, ni si on passe au chiffre suivant qui remplacera k dans la case (i, j), ni lorsque on revient à la fonction appelante où la matrice se retrouve telle qu'elle était.
[6] Le fait qu'il y ait une seule variable M n'est pas vraiment lié à la récursion et est un peu artificiel. D'une part parce que les solutions ne sont pas stockées mais affichées au fur et à mesure qu'elles sont trouvées ; de l'autre parce qu'en fait la matrice est dupliquée à chaque appel récursif. On aurait pu travailler avec une seule variable globale M (sans dupliquer les données), à condition de reconstituer la matrice à chaque étape en ajoutant l'instruction $M(i, j) = 0$ après l'appel récursif.
[7] D'après le programme donné annexe B, la matrice initiale complètement remplie est la même pour toutes ces grilles. On peut se demander si les résultats obtenus au paragraphe suivant ne dépendent justement pas de cette matrice initiale.
[8] Voir note 2
[9] La fonction stat(n, k) écarte les matrices pour lesquelles moins de k chiffres ont été remplis, c'est-à-dire celles pour lesquelles MR(k) a abouti à une impossibilité de continuer, et qui correspondent donc à des grilles sans solution. Mais il n'est pas dit si cela est pris en compte dans le calcul du nombre moyen de solutions.
[10] Il manque le programme pour cette fonction DefinitionA(i, j, M) qui donne la matrice 3×3 extraite de M correspondant au carré 3×3 de la grille contenant (i, j).