

Saé S1.02 : comparaison d'approches algorithmiques

Quelques techniques de résolution de Sudoku

Lors de la SAE S1.01, vous avez écrit un programme où c'est l'utilisateur qui "remplissait" la grille de Sudoku qui lui était proposée. Lors de cette SAE S1.02, vous allez programmer des algorithmes qui permettent de résoudre une grille de Sudoku automatiquement.

Cette première séance détaille quelques techniques que vous serez amenés à programmer. Ces techniques sont basées sur la notion de "valeur candidate".

1. Notion de valeur candidate (ou candidat)

Tous les adeptes de Sudoku le savent, il est plus facile de résoudre un Sudoku si on établit d'abord, pour chaque case, la liste des valeurs qui peuvent être inscrites dans cette case, c'est-à-dire la liste des valeurs qui, pour cette case, respectent les règles du Sudoku (pas deux fois la même valeur sur la ligne, sur la colonne et dans le bloc). Nous appellerons ces valeurs les **candidats** de la case.

Par exemple sur cette grille, la troisième case a pour candidats les valeurs 1, 2 et 4, et la neuvième a 2, 4 et 8 comme candidats.

candidats : 1, 2 et 4
(les autres valeurs sont
présentes soit dans la ligne,
soit dans la colonne,
soit dans le bloc)

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

candidats : 2, 4 et 8

Le premier travail consiste donc à noter (enregistrer) les candidats de chaque case libre de la grille de départ. Sur la grille de l'exemple précédent, cela donne¹ :

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

Exercice 1

Dans la grille fournie sur Moodle (fichier Grille.ods), ajoutez les candidats dans chacune des cases libres.

¹ Les valeurs de la grille de départ sont en gras et, dans les cases vides, les candidats sont disposés en mosaïque .


2. Quelques techniques de résolutions

Toutes les techniques qui suivent sont basées sur le même principe, celui qui consiste à éliminer des candidats d'une case jusqu'à ce qu'il n'en reste plus qu'un seul : ce sera alors la valeur à affecter dans la case.

2.1. Technique n° 1 : singleton nu (ou évident)

Règle de base, cette technique consiste à détecter les cases à candidat unique. C'est par exemple le cas dans le cinquième bloc (bloc central) de la grille précédente :

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



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

Ici, la case centrale n'a qu'un seul candidat, on peut donc :


- l'affecter à cette case,
- et éliminer cette valeur de la liste des candidats des cases qui se trouvent dans le **même bloc**, sur la **même ligne** et sur la **même colonne**.

À noter que faisant cela, la septième case se retrouve avec un seul candidat ; on peut donc à nouveau appliquer cette technique.

2.2. Technique 2 : singleton caché

Il s'agit ici de détecter, dans un bloc (ou une ligne ou une colonne), si une valeur est candidate dans une seule case du bloc (ou de la ligne ou de la colonne).

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



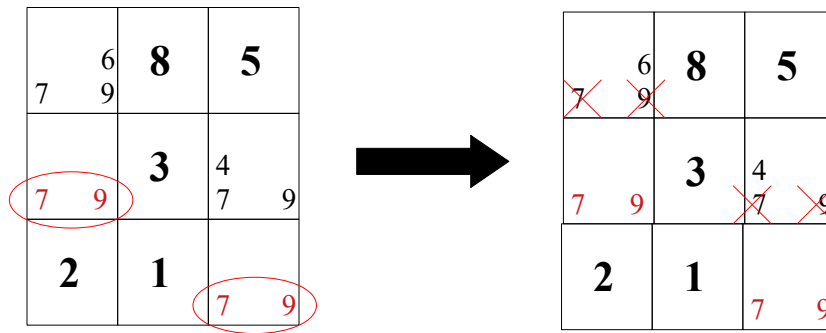
		1
7 8 9	7 8 9	
4 7 8	2	6
4 3 9	4 3 9	5

Dans l'exemple ci-dessus, la valeur **1** n'est candidate que dans une seule case du bloc, on peut donc :

- l'affecter à cette case,
- et éliminer cette valeur de la liste des candidats des cases qui se trouvent sur la **même ligne** et sur la **même colonne**.

2.3. Technique 3 : paires nues (ou évidentes)

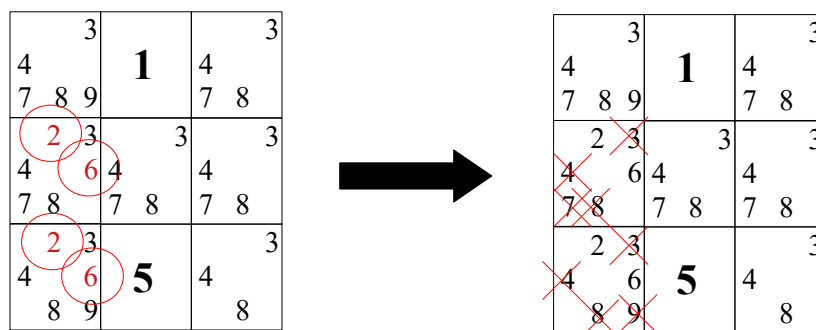
Cette technique consiste à détecter deux cases d'un bloc possédant la même paire de valeurs candidates (et uniquement cette paire).



Dans ce bloc, la quatrième et la neuvième case possède exactement la même paire de candidats. On peut donc éliminer ces candidats de toutes les autres cases **du bloc**.

2.4. Technique 4 : paires cachées

Cette technique consiste à détecter une paire de candidats dans un bloc (ou une ligne ou une colonne), et que cette paire n'est présente dans aucune autre case du bloc (ou de la ligne ou de la colonne). On peut alors éliminer les autres candidats de ces deux cases.



Dans ce bloc, la paire de candidats {2, 6} est présente dans la quatrième et dans la septième case, et dans aucune autre case du bloc. On peut donc éliminer les autres candidats dans **ces deux cases** du bloc.

Il existe d'autres techniques pour tenter de résoudre un Sudoku (triplets nus, triplets cachés, paires associées, X-Wing, Y-Wing, etc.). Regardez par exemple le site sudoku.com.

3. Implémentation

3.1. Structures de données

Les techniques présentées précédemment nécessitent de pouvoir stocker la liste des candidats de chaque case libre d'une grille. La structure de données d'une grille doit en tenir compte. Le type `tGrille` définit une matrice $n^2 \times n^2$ de cases.

```
#define N 3
#define TAILLE (N*N)
typedef tCase tGrille[TAILLE][TAILLE];
```

Pour le type `tCase`, deux propositions sont détaillées ci-dessous.

3.1.1 Un tableau de candidats à longueur variable

Dans cette première proposition, chaque case de la grille est caractérisée par :

- sa valeur (ou 0 si c'est une case libre),
- un tableau constitué d'au plus n^2 candidats ; dans ce tableau les candidats sont ajoutés séquentiellement,
- le nombre de candidats.

Par exemple la case

	3
4	
7	8

 sera représentée par la structure :

valeur :	0										
candidats :	<table border="1" style="display: inline-table;"><tr><td>3</td><td>4</td><td>7</td><td>8</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>	3	4	7	8						
3	4	7	8								
nbCandidats :	4										

Le type `tCase` devient :

```
typedef struct {
    int valeur;
    int candidats[TAILLE];
    int nbCandidats;
} tCase1;
```

3.1.2 Un tableau de booléens

Dans cette autre proposition, chaque case de la grille est caractérisée par :

- sa valeur (ou 0 si c'est une case libre),
- un tableau de booléens tels que $t[i]=\text{vrai}$ si la valeur i est candidate pour cette case (et $t[i]=\text{faux}$ sinon),
- le nombre de candidats.

La case

	3
4	
7	8

 sera alors représentée par la structure :

valeur :	0																					
candidats :	<table border="1" style="width: 100%; text-align: center;"><tr><td style="width: 5%;"></td><td style="width: 15%;">false</td><td style="width: 15%;">false</td><td style="width: 15%;">true</td><td style="width: 15%;">true</td><td style="width: 15%;">false</td><td style="width: 15%;">false</td><td style="width: 15%;">true</td><td style="width: 15%;">true</td><td style="width: 15%;">false</td></tr><tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td></tr></table>		false	false	true	true	false	false	true	true	false		0	1	2	3	4	5	6	7	8	9
	false	false	true	true	false	false	true	true	false													
	0	1	2	3	4	5	6	7	8	9												
nbCandidats :	4																					

Notez que la case d'indice 0 est inutilisée.

Le type `tCase` devient :

```
typedef struct {
    int valeur;
    bool candidats[TAILLE+1]; // la case d'indice 0
                               // est neutralisée
    int nbCandidats;
} tCase2;
```

3.1.3 Des fonctions utilitaires

Pour gérer plus facilement les candidats d'une case, on se dote de quatre procédures/fonctions :

```
procédure ajouterCandidat( entF/sortF laCase : tCase1,  
                           entF val:entier)
```

qui ajoute la valeur `val` à la liste des candidats de la case passée en paramètre d'entrée/sortie,

```
procédure retirerCandidat( entF/sortF laCase : tCase1,  
                           entF val : entier)
```

qui supprime la valeur `val` de la liste des candidats de la case passée en paramètre d'entrée/sortie,

```
fonction estCandidat( entF laCase : tCase1,  
                     entF val : entier) délivre booléen
```

qui retourne vrai si `val` est l'un des candidats de la case passée en paramètre d'entrée et faux sinon,

```
fonction nbCandidats(entF laCase : tCase1) délivre entier
```

qui retourne le nombre de candidats de la case passée en paramètre d'entrée.

Exercice 2

Écrivez en pseudo code ces quatre procédures/fonctions pour chaque structure de données proposée (`tCase1` puis `tCase2`).

3.2. Algorithme

Voici une proposition d'algorithme pour la résolution d'une grille. Le programme s'arrête si la grille est pleine ou bien si au cours d'une itération, aucune des techniques n'a permis de faire "progresser" la solution², c'est-à-dire si aucune technique n'a permis de remplir une case vide ou d'éliminer le moindre candidat.

```
programme deduction
  g : tGrille;
  progression : booléen;
  nbCasesVides : entier;

  nbCasesVides = chargerGrille(g);
  initialiserCandidats(g);
  progression = true;

  tant que (nbCaseVides <> 0 ET progression) faire
    progression = false;

    // technique du singleton nu
    pour chaque case libre de la grille faire
      si la case n'a qu'un seul candidat alors
        • affecter ce candidat à la case
        • nbCasesVides = nbCasesVides - 1;
        • retirer ce candidat de toutes les cases de la
          même ligne, de la même colonne et du même bloc
        • progression = true;
      finsi
    finfaire

    // technique du singleton caché
    ...

    // autres techniques...
    ...

  finfaire
fin
```

Vous pouvez remarquer que seule la première technique est mise en œuvre dans cet algorithme. Vous pourrez bien sûr, au cours de votre SAE, ajouter l'implémentation d'autres techniques pour améliorer la résolution.

2 En effet, l'algorithme ne garantit pas une résolution complète de la grille.

3.3. Critères de performance

Comme on l'a vu précédemment, l'algorithme présenté ne garantit pas la résolution complète d'une grille de Sudoku (ça dépendra du niveau de difficulté de la grille). Pour juger de la performance du programme, on se propose de mettre en place deux critères :

- le **nombre de cases remplies** par le programme (et donc le **taux de remplissage**) : plus le programme aura réussi à inscrire des valeurs dans les cases, meilleure sera la résolution ;
- le **nombre total de candidats éliminés** par le programme (et donc le **pourcentage d'élimination**) : plus le programme aura éliminé de candidats, meilleure sera la résolution.

Exercice 3

Écrivez en pseudo code une procédure :

```
void afficherStats (...)
```

qui sera appelée en fin de programme et qui affichera :

- le nombre de cases remplies par le programme,
- le taux de remplissage (déduit de la valeur précédente),
- le nombre total de candidats éliminés,
- le pourcentage d'élimination.