INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN NORMANDIE

Projets GM3
2022/2023

Résolution de sudoku

S1V3 - MIPP

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

Étudiants : Constance BAU et Paul MEHAUD
Encadrant : Cendrella CHAHINE

Table des matières

1. Présentation du sujet

Le sudoku est un jeu de réflexion inventé en 1979 par l'Américain Howard GARNS. Ce jeu est notamment inspiré du carré latin.

Le but du jeu est de remplir une grille 9×9 par des chiffres qui ne se trouvent jamais plus d'une fois dans la même ligne, la même colonne ou le même carré (bloc de 3×3 cases). Un exemple de grille non résolue est donnée en page de couverture de ce rapport.

La difficulté du jeu est inversement proportionnelle au nombre de chiffres présents sur la grille de départ. Moins il y en a, moins il y a de solutions de placement évidentes, ce qui oblige à « faire des paris » : on place un chiffre sur une case en espérant que ce soit la bonne solution mais sans en être sûr.

Pour résoudre ce sudoku de manière automatique, nous avons implémenté un algorithme qui remplit les cases ne présentant qu'une seule possibilité de chiffre. Elle répète cette action jusqu'à ce qu'aucune case ne puisse être remplie de cette manière. Ensuite pour les cases vides restantes nous avons implémenté une solution de backtracking, un algorithme qui fait des hypothèses sur les chiffres à placer. C'est un parcours récursif en profondeur du sudoku qui, s'il mène à un échec, reviens à l'étape précédente. C'est donc ce qu'on appelle un algorithme glout

2. Modélisation

Dans ce chapitre nous allons expliquer le fonctionnement de notre programme. Pour résoudre une grille de sudoku le programme va tout d'abord remplir les cases dans lesquelles un seul chiffre est possible (comme le ferait un humain). La procédure qui va le faire s'appelle *resoudreUnePossibilite*. Puis une fois qu'aucun chiffre ne peut plus être trouvé de cette manière, s'il reste des cases vides, nous allons utiliser la « force brute » aussi appelé algorithme de backtracking pour remplir les cases. C'est à dire que l'algorithme va mettre dans la première case vide le premier chiffre possible puis continuer ainsi jusqu'à ce qu'il n'y ait plus aucune possibilité pour une case, il reviendra alors à la case précédemment modifiée pour changer le chiffre mis en place. Il fera cela de manière récursive jusqu'au remplissage complet de la grille. Cette technique est aussi utilisée par les humains pour résoudre les sudokus mais à moindre échelle, c'est à dire seulement quand il y a deux possibilités pour plusieurs cases et qu'on ne peut pas déterminer laquelle est la bonne. On appelle cette méthode « faire des hypothèses ». La fonction *resoudreSudoku* va utiliser la même méthode mais à grande échelle.

I. Pseudo-code des algorithmes récursifs

Voici le pseudo-code des deux algorithmes récursifs du programme :

```
Procédure resoudreUnePossibilite(E/S grille : tableau[1..9][1..9])
Var
  i, j, chiffreTest, tmp, compteur1, compteur2 : entier
Début
  compteur2 <- 0
  Pour i <- 1 à 9 inc +1 Faire
    Pour j <- 1 à 9 inc +1 Faire
      Si grille[i][j] = 0
        Alors compteur1 <- 1
          Pour chiffreTest <- 1 à 9 inc +1 Faire
            Si (nonDansCarre(i,j,chiffreTest) et nonDansLigne(i,chiffreTest) et
nonDansColonne(j,chiffreTest))
              Alors tmp <- chiffreTest
                compteur1 <- compteur1 + 1
            FinSi
          FinPour
          Si compteur1 = 1 {on regarde s'il y a un seul chiffre possible pour la case et
si oui on place le chiffre}
            Alors grille[i][j] <- tmp
              compteur2 = 1
          FinSi
      FinSi
    FinPour
```

```
  FinPour
  Si compteur2 = 1 {si un chiffre a été placé on recommence la recherche de possibilité
unique dans la grille}
    Alors resoudreUnePossibilite()
  FinSi
Fin


Fonction resoudreSudoku(E i, j : entier) : booléen
Var
  chiffre, i_apres, j_apres : entier
Début
  chiffre <- 1
  i_apres <- 0
  j_apres <- 0
  Si grille[i][j] ≠ 0
    Alors Si (i = 9 et j = 9)
        Alors retourner(VRAI)
      FinSi
      Si i < 9
        Alors i <- i + 1
        Sinon Si j < 9
            Alors i <- 0
              j <- j + 1
          FinSi
      FinSi
      Si resoudreSudoku(i,j)
        Alors retourner(VRAI)
        Sinon retourner(FAUX)
      FinSi
  FinSi
  Si grille[i][j] = 0
    TantQue chiffre <= 9 Faire
      Si (nonDansCarre(i,j,chiffre) et nonDansLigne(i,chiffre) et
nonDansColonne(j,chiffre) {on vérifie si le chiffre est valide}
        Alors grille[i][j] <- chiffre
          Si (i=9 et j=9)
            Alors retourner(VRAI)
          FinSi
          Si i < 9
            Alors i_apres <- i + 1
            Sinon i_apres <- 0
              j_apres <- j + 1
          FinSi
          Si resoudreSudoku(i_apres,j_apres)
            Alors retourner(VRAI)
          FinSi
```

```
      FinSi
      chiffre <- chiffre + 1
    FinTantQue
    grille[i][j] <- 0
    retourner(FAUX)
  FinSi
Fin
```

II. Fonctions complémentaires

Afin de faire fonctionner ces deux algorithmes, nous aurons besoins de fonctions et procédures annexes, dont voici les présentations

void remplissageGrille() **:** Cette procédure remplit la grille qui est une variable globale du programme (sous forme d'un tableau statique de taille 9×9) grâce à un fichier de données choisi par l'utilisateur.

int dansLaLigne(int ligne, int chiffre) **:** Cette fonction permet de savoir si le chiffre mis en entrée est dans la ligne mise en entrée, s'il l'est elle renvoie 1, 0 sinon.

int dansLaColonne(int colonne, int chiffre) **:** Cette fonction permet de savoir si le chiffre mis en entrée est dans la colonne mise en entrée, s'il l'est elle renvoie 1, 0 sinon.

int quelleBorne(int indice) **:** Cette fonction est utile pour trouver les borne du carré dans lequel la fonction *dansLeCarre* recherche un chiffre.

int dansLeCarre(int i, int j, int chiffre) **:** Cette fonction permet de savoir si le chiffre mis en entrée est dans le carré comportant la case de coordonnées [i][j] , i et j étant les coordonnées mises en entrée.

void afficheSudoku() **:** Cette procédure affiche la grille mise en variable globale du programme.

void resoudreUnePossibilite() **:** Cette procédure remplit les cases vides de la grille .

int resoudreSudoku(int i, int j) **:** Cette fonction remplit les cases vides de la grille avec la force brute, elle fait des suppositions, des hypothèses sur les chiffres à mettre dans les cases et changent ces hypothèses dès qu'une case vide ne possède plus aucun chiffre possible. Elle fait cela jusqu'à pouvoir remplir entièrement la grille.

3. Choix d'implémentations et explications

Pour l'implémentation nous avons choisi des tableaux statiques car la taille d'une grille de sudoku est toujours la même. On a choisi que la grille de sudoku soit une variable globale du programme donc une variable de profondeur 0 qui peut être modifié dans chaque fonction du programme.

Pour représenter les lignes et les colonnes, nous avons choisi de les nommer i et j, comme pour les matrices en mathématiques. Cette notation nous a permit un facilité lors de l'écriture du programme.

Pour coller à l'implémentation des tableaux en C, les colonnes, lignes et carrés sont numérotés dans le programme de 0 à 8.

La variable *chiffre* représente un chiffre que l'on peut mettre dans une case de la grille, il est donc numéroté de 1 à 9.

4. Résultats des jeux d'essais

Pour tester notre algorithme nous avons choisi différentes grilles d'essais, dont certaines sont fausses. Vous pouvez trouver avec ce rapport tous les fichiers de données et de résultats.

Afin mesurer le temps d'exécution et vérifier si la difficulté d'une grille (nombre de chiffres déjà présents dans une grille) influait sur le temps d'exécution, nous avons implémenté un timer grâce au site suivant.

I. Grilles de difficulté faible

Pour commencer, nous avons testé la résolution de deux grilles de difficulté faible :

```
+-------+-------+-------+
| 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 |
+-------+-------+-------+
```

FIGURE 4.1. Grille 1

```
+-------+-------+-------+
|   4   | 3     |   7   |
|       | 2 6   | 1 3   |
|     3 | 4   1 | 5   2 |
+-------+-------+-------+
|     1 | 6 7 4 |     5 |
|   6   | 1 5 9 |   4   |
| 9   4 |       |     1 |
+-------+-------+-------+
|     5 | 7 4 6 | 9     |
| 4 9   |       | 3 1   |
| 8   6 | 9   3 |       |
+-------+-------+-------+
```

FIGURE 4.2. Grille 2

Ce qui donne alors, après exécution du programme :

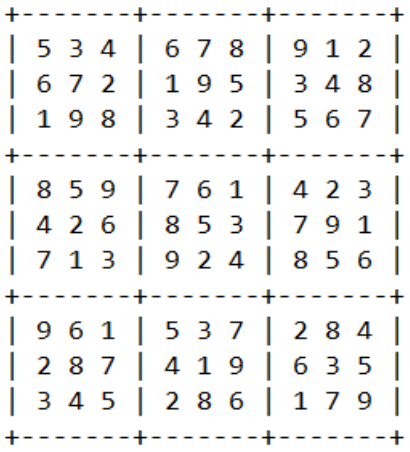

FIGURE 4.3. Résultat Grille 1

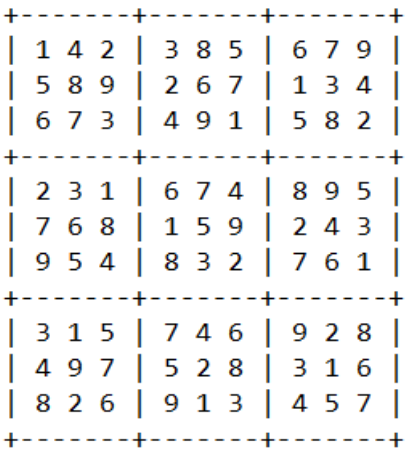

FIGURE 4.4. Résultat Grille 2

Pour la grille 1, le temps d'exécution est de 0,000 25 secondes et, pour la grille 2, il est de 0,000 31 secondes.

II. Grille de difficulté élevée

Pour poursuivre les tests, nous avons créé spécialement une grille très difficile :

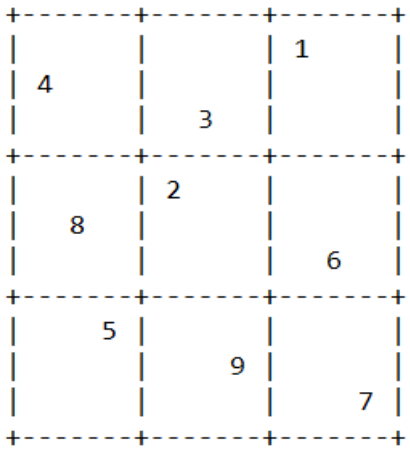

FIGURE 4.5. Grille 3

La difficulté de cette grille réside dans le fait qu'il n'y a qu'une seule occurrence de chaque chiffres. Ces chiffres sont répartis aléatoirement dans la grille de telle manière qu'il n'y en ai qu'un par ligne, colonne et carré.

Voici ce qu'on obtient comme résultat :

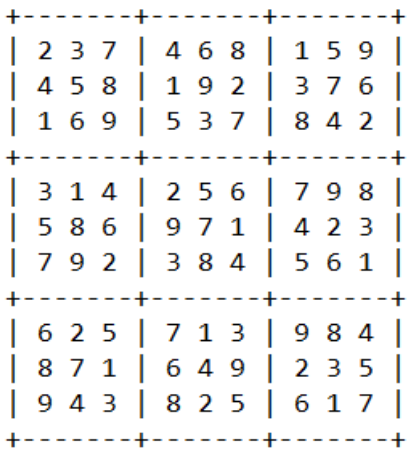

FIGURE 4.6. Résultat Grille 3

En terme de temps d'exécution, on obtient : 0,000 41 secondes.

III. Grille vide

Afin de tester notre algorithme sur des cas limites, nous avons commencé par tester s'il pouvait résoudre une grille vide. Voici le résultat obtenu :

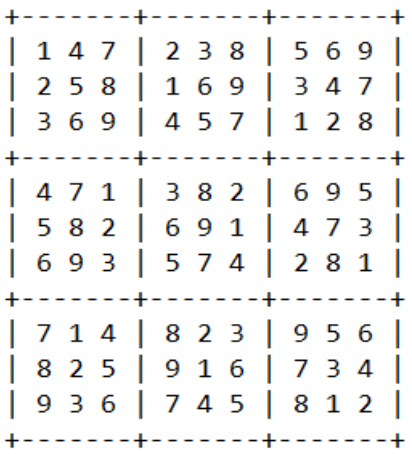

FIGURE 4.7. Résultat Grille vide

Ce résultat est obtenu au bout de 0,000 43 secondes, à peine plus que la grille précédente. Ici, on voit bien comment l'algorithme fonctionne : il remplit la grille colonne par colonne, en partant de la gauche.

IV. Grilles fausses

Enfin, pour continuer à tester notre algorithme, nous l'avons testé sur des grilles volontairement fausses. La première est une grille est une grille où les chiffres du premier carré ne respectent pas la règle de l'unicité de présence au sein des colonnes, lignes et carrés. Nous avons aussi rajouté 3 chiffres aléatoires.

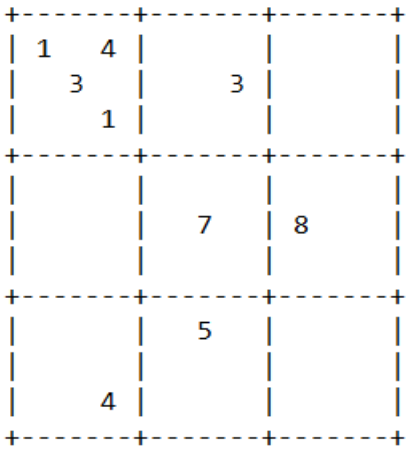

FIGURE 4.8. Grille fausse 1

Ensuite, nous avons testé une autre grille fausse, qui est la réplique de la grille 1, à l'exception près que le premier chiffre en haut à gauche a été remplacé par 10, un chiffre impossible au sudoku.

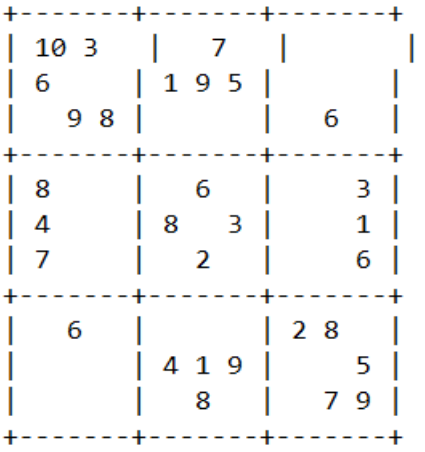

FIGURE 4.9. Grille fausse 2

A l'exécution, pour la première grille le programme ne rend rien mais arrête son exécution et nous donne un temps d'exécution de 2,495 90 secondes.

Pour la deuxième grille voici le résultat, obtenu en 0,000 25 secondes :

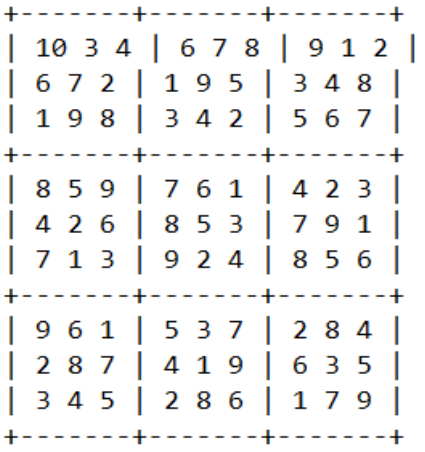

FIGURE 4.10. Résultat Grille fausse 2

Ce résultat est strictement identique à celui de la grille 1 (sauf le 10 au début).

V. Comparaison de résultats avec un algorithme sans *resoudreUnePossibilite()*

Dans une première version du programme (*sudoku2.c*) nous n'avions pas implémenté la procédure *resoudreUnePossibilite()*. Nous l'avons donc testée sur les trois premières grilles (cf. fichiers de résultats avec « *bis* » dans le nom). Voici les temps d'exécution que nous avons obtenu :

Pour la grille 1 : 0,000 93 secondes. Pour la grille 2 : 0,000 26 secondes. Pour la grille 3 : 0,000 51 secondes.

On remarque alors que pour la grille 1, le temps d'exécution est quasiment divisé par 4. Alors que pour les grilles 2 et 3, le temps d'exécution est augmenté de 0,000 10 secondes.

A ce jour, nous ne savons toujours pas expliquer pourquoi cette incohérence existe.

Malgré tout, les temps d'exécution restent très faibles.

5. Synthèse critique

Pour conclure, on a pu, grâce à ce projet, approcher une forme primitive d'intelligence artificielle : la force brute. Ces algorithmes sont peu optimisés mais très adaptés à un problème comme le sudoku. Cette idée est confirmée par les temps d'exécution qui sont très faibles. Cependant, on remarque que le temps d'exécution est corrélé avec la difficulté de la grille. Plus elle est difficile, plus il faut de temps pour la résoudre.

Malgré le fait que notre algorithme donne des résultats justes, nous ne pouvons toujours pas expliquer les différences de temps d'exécution soulevés dans la partie précédente.