T.P. d'informatique n° 5

Récursivité : backtracking

Solution

CORRECTION

Le **backtracking** (que l'on peut traduire par « marche arrière » mais nommé en informatique « **retour sur trace** ») est une méthode couramment employée pour résoudre certains problèmes de programmation. Wikipédia en donne la définition suivante :

*Le **retour sur trace** (appelé aussi **backtracking** en anglais) est un algorithme qui consiste à revenir légèrement en arrière sur des décisions prises afin de sortir d'un blocage. La méthode des essais et erreurs constitue un exemple simple de backtracking.*

Le terme est surtout utilisé en programmation, où il désigne une stratégie pour trouver des solutions à des problèmes de satisfaction de contraintes.

Nous allons utiliser le backtracking dans la résolution d'une grille de sudoku par ordinateur et nous verrons l'intérêt de la récursivité.

Nous pourrions tenter de faire un programme "intelligent", qui remplisse la grille de façon logique. Seulement si cela est assez simple pour les sudokus faciles, il en va autrement avec les grilles plus complexes ... En effet, les méthodes de résolution dans ce cas seront plutôt complexes à écrire, ce qui nous amène à nous demander s'il ne serait pas plus judicieux de passer directement par une technique dite **de force brute** : il s'agit de tester, une à une, toutes les combinaisons possibles.

C'est là qu'un problème de complexité se pose : une force brute est-elle envisageable en pratique ? Pour N cases vides dans la grille, nous avons 9^N combinaisons possibles. Dans la grille ci-dessous, $N = 57$ la technique de force brute naïve (avec ses $2,5 \cdot 10^{54}$ essais) semble donc difficile à imaginer...

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

C'est là qu'intervient le principe du retour sur trace. Si nous remplissons la grille au fur et à mesure, en vérifiant constamment qu'elle reste toujours potentiellement valide, on arrive assez vite à des situations de blocage où il ne sert à rien de continuer. Dans un tel cas, on revient en arrière en évitant de continuer une exploration inutile. Avec la méthode du backtracking, le nombre de combinaisons à explorer est considérablement réduit, et la force brute devient envisageable dans ces conditions pour résoudre notre grille de sudoku.

1^{ère} étape : structurer les données. On propose d'utiliser un tableau bidimensionnel (i.e. une liste de listes) pour conserver la grille ; cette grille contient des entiers de 0 à 9, le 0 signifiant que la case est vide. Ainsi, la grille vue précédemment sera définie par :

```
s=[[8,0,0,9,0,0,0,0,1],
   [0,0,1,0,8,0,2,0,9],
   [5,0,0,0,4,0,0,0,0],
   [2,0,0,3,0,0,0,0,6],
   [0,0,0,2,0,0,8,0,0],
   [0,0,9,8,0,4,1,7,0],
   [0,0,0,0,5,0,0,0,0],
   [0,0,0,7,0,0,0,0,0],
   [0,0,0,0,2,6,0,0,8]]
```

Vous trouverez sur le site du lycée un fichier nommé **TP5grilles.py** contenant deux exemples de grilles incomplètes de sudoku dont l'exemple précédent.

On rappelle que le principe du sudoku consiste à remplir la grille avec les chiffres de 1 à 9, chaque chiffre devant apparaître une et seule fois dans chaque ligne, chaque colonne et chacun des 9 blocs 3×3 .

Quest. 1. On rappelle que si s est une grille de sudoku, pour $i \in \llbracket 0, 8 \rrbracket$, la ligne d'indice i est donnée par $s[i]$. Écrire une fonction python **colonne(s,j)** recevant une grille s et un indice $j \in \llbracket 0, 8 \rrbracket$ qui retourne la liste des éléments de la colonne j de s . Écrire une fonction python **bloc(s,k)** recevant une grille s et un indice $k \in \llbracket 0, 8 \rrbracket$ qui retourne la liste des éléments du bloc numéro k de s , les trois premiers blocs du haut étant numérotés de gauche à droite de 0 à 2, puis ceux du milieu de 3 à 5 et ceux du bas de 6 à 8.

Quest. 2. Écrire une fonction python **numBloc(i,j)** recevant deux indices i et j de $\llbracket 0, 8 \rrbracket$ et qui retourne le numéro de bloc dans lequel se trouve la case i, j .

On pourra utiliser $i//3$ et $j//3$ et respecter la numérotation vue à la question précédente.

2^{ème} étape : fonctions utiles.

Quest. 3. Écrire une fonction python **correcteLigne(l)** qui reçoit une liste l et teste la liste en vérifiant si elle ne contient pas deux fois l'un des entiers de 1 à 9 ; elle retourne **False** si l'un des entiers de 1 à 9 apparaît plusieurs fois et retourne **True** sinon. Les 0 ne sont eux pas testés.

Quest. 4. Écrire une fonction python **correct(s)** qui reçoit une grille de sudoku s non nécessairement complète et teste si la grille est valide, les 0 n'étant donc pas testés. Il suffit de faire un test sur chaque ligne, chaque colonne et chaque bloc 3×3 .

Quest. 5. Écrire une fonction python **absentsLigne(l)** qui reçoit une liste l et retourne la liste des entiers de 1 à 9 qui sont absents de la liste l .

Quest. 6. Écrire une fonction python **valPossibles(s,i,j)** recevant une grille de sudoku s incomplète et deux indices i et j de $\llbracket 0, 8 \rrbracket$ et déterminant en regardant la ligne, la colonne et le bloc contenant la case i, j la liste des valeurs qui peuvent être attribuées à la case i, j . On pourra écrire une fonction **intersect(u,v)** donnant l'intersection des deux listes u et v passées en argument.

3^{ème} étape : backtracking.

Quest. 7. Écrire une fonction récursive python **resoudre(s)** recevant une grille de sudoku s incomplète, choisissant un zéro et lui donnant toutes les valeurs possibles pour s'appeler récursivement ; elle doit rendre **True** dès qu'une valeur donne une grille valide et **False** sinon.

On pourra écrire et utiliser une fonction **trouveZero(s)** qui donne le meilleur 0 (à savoir celui pour lequel le nombre de valeurs possibles est minimal... ceci permettant d'optimiser la fonction).

Pour que les tests sur la grille puissent fonctionner, il faut veiller à actualiser la grille au cours de la descente récursive. Mais attention, il ne suffit pas d'enregistrer le choix effectué dans la grille avant de passer à l'appel suivant, mais il faut aussi réinitialiser la case à zéro en cas d'échec. Sans quoi, la grille risque de ressortir invalide...

Solution

```

def affiche(s):
    for i in range(len(s)):
        ligne = ""
        if i%3 == 0:
            print('-'*31)
        for j in range(len(s[i])):
            if j%3 == 0:
                ligne += '|'
            ligne += ' '+str(s[i][j])+' '
            print(ligne)
        print('-'*31)

def colonne(s, j):
    L = []
    for i in range(9):
        L.append(s[i][j])
    return(L)

def bloc(s, k):
    L = []
    for i in range(3*(k//3), 3*(k//3)+3):
        for j in range(3*(k%3), 3*(k%3)+3):
            L.append(s[i][j])
    return(L)

def numBloc(i, j):
    return(3*(i//3)+(j//3))

def correcteLigne(L):
    #teste si la liste L passée en argument
    #contient au plus une fois : 1, 2, ..., 9
    cn=0
    for j in range(9):
        if L[j] == n:
            cn+=1
            if cn>1:
                return(False)
    return(True)

def correct(s):
    for i in range(9):
        if not(correcteLigne(s[i])):
            return(False)
        if not(correcteLigne(colonne(s, i))):
            return(False)
        if not(correcteLigne(bloc(s, i))):
            return(False)
    return(True)

def absentsLigne(L):
    absents=[]
    for n in range(1,10):
        if not n in L:
            absents.append(n)
    return(absents)

def intersect(u, v):
    L = []
    for x in u:
        if x in v:
            L.append(x)
    return(L)

def valPossibles(s, i, j):
    B = bloc(s, numBloc(i, j))

```

```

L1=intersect(absentsLigne(s[i]), absentsLigne(colonne(s, j)))
return(intersect(absentsLigne(B), L1))

def complete(s):
    nb0 = 0
    for i in range(9):
        nb0 += 1
        for j in range(9):
            if s[i][j] == 0:
                vp = valPossibles(s, i, j)
                if len(vp) == 1:
                    s[i][j] = vp[0]
                    nb0 += 1
    print(nb0, "case a trouver")

def trouveZero(s):
    res = -1
    nb = 10
    for i in range(9):
        for j in range(9):
            if s[i][j] == 0 and len(valPossibles(s, i, j)) < nb:
                nb = len(valPossibles(s, i, j))
                res = [i, j]
    return(res)

compteur = 0
def resoudre(s):
    global compteur
    if correct(s):
        tz = trouveZero(s)
        if tz == -1:
            return(True)
        else:
            compteur += 1
            i, j = tz[0], tz[1]
            for n in range(1, 10):
                s[i][j] = n
                if resoudre(s):
                    return(True)
            s[i][j] = 0
            return(False)

def affiche(tab):
    for L in tab:
        char = ' '
        for x in L:
            char = char + ' ' + str(x) + ' '
        print(char)

def somme(L):
    som = 0
    for k in L:
        som += k
    return(som)

def resoudre2(s):
    global compteur
    if correct(s):
        tz = trouveZero(s)
        if tz == -1:
            return(True)
        else:
            compteur += 1
            print(compteur, somme(s[0]),
                  print(compteur, somme(s[1]), somme(s[2]), somme(s[3]),
                  print(compteur, somme(s[4]),
                  print(compteur, somme(s[5]), somme(s[6]), somme(s[7]), somme(s[8]),
                  print(compteur,
                  print(compteur, somme(s[1]), somme(s[2]), somme(s[3]),
                  print(compteur, somme(s[4]), somme(s[5]), somme(s[6]),
                  print(compteur, somme(s[7]), somme(s[8]),
                  print(compteur,

```

```
    i,j = tz[0],tz[1]
    for n in range(1,10):
        s[i][j] = n
        if resoudre2(s):
            return(True)
            s[i][j] = 0
    return(False)
```

```
# Exemples de grilles
```

```
s=[
[8,0,0,9,0,0,0,0,1],
[0,0,1,0,8,0,2,0,9],
[5,0,0,0,4,0,0,0,0],
[2,0,0,3,0,0,0,0,6],
[0,0,0,2,0,0,8,0,0],
[0,0,9,8,0,4,1,7,0],
[0,0,0,0,5,0,0,0,0],
[0,0,0,7,0,0,0,0,0],
[0,0,0,0,2,6,0,0,8]]
```

```
s2=[
[0,0,0,0,4,0,6,0,0],
[0,5,0,0,3,0,0,0,1],
[0,0,9,0,0,2,0,4,0],
[0,4,7,0,6,0,0,8,0],
[6,0,0,0,0,0,0,0,7],
[0,3,0,0,8,0,2,5,0],
[0,7,0,4,0,0,1,0,0],
[8,0,0,0,7,0,0,9,0],
[0,0,4,0,2,0,0,0,0]]
```