TP d'Informatique

15 Février 2018

BT1

Durée : 2 heures

Nom :
Prénom:
Groupe:

Correcteur:

Kévin GAST, Alexandre MANUEL et Zinedine REBIAI

v. 1.0
12 pages
BT1_2022_TP_2018-02-15

Consignes pour ce TP

- Lire la totalité du sujet avant de commencer
- Ne pas savoir expliquer votre code si on vous le demande représente un cas de triche
- Posez vos questions aux assistants pendant le TP, et envoyez un mail à supbiotech-bioinfo-bt1@googlegroups.com en dehors
- Si vous avez un message à nous faire passer concernant votre TP, merci d'ajouter cela dans un fichier README.txt dans votre rendu. Les retours sur ce TP sont les bienvenus également.

Introduction

Le principe du TP

Le but de ce TP est de créer un solveur de sudoku en python grâce aux notions que l'on a abordé jusqu'à présent.

Le TP est divisé en **paliers** qui doivent être réalisés l'un après l'autre et **dans l'ordre.** En effet vous aurez besoin de réussir le premier palier avant de passer au second, et ainsi de suite.

(Lisez attentivement le sujet afin de bien comprendre ce que l'on attend de vous, normalement il suffit à lui seul pour réaliser le TP grâce aux indications que l'on vous fournit. Vous pouvez également poser des questions mais nous ne vous donnerons bien évidement pas la solution au problème).

Vous avez récemment appris à faire des fonctions récursives. Comme indiqué en TD, la récursivité est une façon de voir les choses. Le but est de choisir la façon de faire la plus commode et intuitive. Dans ce TP vous trouverez des fonctions qui sont plus faciles à réaliser en itératif, et d'autres plus faciles à réaliser en récursif. Comme dans la vie, vous êtes à tout moment libres de choisir la façon de coder que vous préférez, le plus facile étant de loin de ne pas se cantonner à une seule façon de faire pour ce TP.

Les règles du Sudoku

Voici un petit rappel des règles du sudoku pour ceux qui ne connaissent pas.

Le jeu est composé d'une grille de 9 par 9 dans laquelle se trouvent des chiffres ou non. Le but est d'arriver à compléter les cases vides avec des chiffres de 1 à 9 tout en faisant en sorte que la grille soit une grille de sudoku « valide ».

La grille est divisée en neuf carrés de 3 fois 3 cases. Pour qu'une grille soit valide, il faut qu'on ne trouve pas deux fois le même chiffre dans un même carré de 3x3, dans une même ligne ou dans une même colonne de la grille.

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

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

Une grille de sudoku vierge à gauche, et sa version résolue à droite.

Comment réaliser ce TP

Vous avez déjà vu comment créer et afficher une grille dans le TP morpion. Ici, nous allons donc sauter ces étapes. Il vous est fourni un fichier *application.py* qui contient la gestion graphique du TP sudoku. Vous devrez écrire le corps des fonctions dans les fichiers *palier_x.py*. Afin de tester votre TP, vous pouvez soit lancer *application.py*, soit lancer *check.py*.

Veillez à ne pas casser les signatures des fonctions dans les fichiers *palier_x.py*.

Rappel :

Si vous voulez ajouter un paramètre à une des fonctions sans casser sa signature, deux possibilités s'offrent à vous. Admettons que vous vouliez ajouter un paramètre *param* à la fonction *grid_is_complete(grid)*.

Vous pouvez soit ajouter le paramètre avec une valeur par défaut définie :

```
def is_finished(grid, param=42):
    # ...
```

Soit transformer is_finish en « fonction chapeau » :

```
def is_finished_sub(grid, param):
    # ...

def is_finished(grid):
    return is_finished_sub(grid, 42)
```

Avoir la signature suivant ne fonctionnera pas et occasionnera des points en moins sur votre note :

```
def is_finished(grid, param):
    # ...
```

La triche

La triche sera fortement sanctionnée. Tricher veut dire ne pas savoir expliquer votre code si on vous le demande. Vous avez en revanche le droit, et êtes même invités à collaborer avec vos camarades. Ne prenez pas le risque de rendre un code que vous ne comprenez pas. Nous ferons des interviews aléatoires pour vous demander d'expliquer votre TP.

Le rendu du TP

Votre rendu doit être sous la forme d'une archive (fichier compressé). Cette dernière devra se nommer de la manière suivante :

```
tp8_login.zip
```

Remplacez login par **votre** login qui doit être sur la feuille de mot de passe.

Cette archive **doit** contenir un unique dossier portant le même nom que l'archive (sans le .zip à la fin). Ce dossier devra contenir au moins les fichiers suivants :

```
palier_1.py
palier_2.py
palier_3.py
palier_4.py
```

Vous pouvez ajouter un fichier README.txt dans ce même dossier si vous avez un message à nous faire passer concernant votre rendu ou un retour à faire sur le sujet du TP.

Tester votre TP

Vous pouvez à tout moment tester votre TP pour avoir une idée de si celui-ci est correct ou non. Pour cela plusieurs possibilités s'offrent à vous :

L'interface graphique

Vous pouvez exécuter le fichier application.py, celui-ci ouvrira une interface de sudoku. A partir de là, vous pouvez essayer d'insérer des nombres dans des cases pour vérifier si les paliers 1 à 3 fonctionnent, ou cliquer sur le bouton Résoudre pour tester le palier 4.

La test suite

En exécutant le fichier *check.py*, vous aurez un retour sur vos paliers. Ce retour est à titre indicatif. Les tests que nous utiliserons pour vous noter seront similaires mais différents.

1. Palier 1 : déterminer s'il est possible d'insérer un chiffre dans la grille

Afin de déterminer si une grille est valide, il va falloir vérifier qu'aucun chiffre ne rend la grille invalide. Pour cela, nous allons avoir besoin d'une fonction qui détermine si un chiffre à une certaine position rend la grille invalide ou non.

Un chiffre invalide une grille si un chiffre identique se trouve dans la même ligne, dans la même colonne ou dans le même carré.

Il va donc falloir écrire le corps des trois fonctions suivantes :

- *can_insert_in_row(row, digit)* : qui vérifie s'il est possible d'insérer *digit* dans une ligne donnée. *row* est la liste des chiffres dans la ligne à analyser.
- *can_insert_in_column(column, digit)* : qui vérifie s'il est possible d'insérer *digit* dans une colonne donnée. *column* est la liste des chiffres dans la colonne à analyser.
- *can_insert_in_square(square, digit)* : qui vérifie s'il est possible d'insérer *digit* dans un carré donné. *square* contient une matrice de 3x3 chiffres représentant le carré à analyser.

Note : Un zéro représente une case vide. Le cas où l'on essaye d'insérer une case vide ne sera pas testé.

Exemples :

```
>>> can_insert_in_row([5, 3, 0, 0, 7, 0, 0, 0, 0], 1)
True
>>> can_insert_in_row([5, 3, 0, 0, 7, 0, 0, 0, 0], 3)
False
>>> can_insert_in_column([5, 6, 0, 8, 4, 7, 0, 0, 0], 2)
True
>>> can_insert_in_column([5, 6, 0, 8, 4, 7, 0, 0, 0], 6)
False
>>> can_insert_in_square([[5, 3, 0], [6, 0, 0], [0, 9, 8]], 1)
True
>>> can_insert_in_square([[5, 3, 0], [6, 0, 0], [0, 9, 8]], 9)
False
```

2. Palier 2 : extraire une ligne, une colonne et un carré de la grille

Il va maintenant falloir réussir à extraire une ligne, une colonne ou un carré de la grille afin de pouvoir utiliser les fonctions du palier 1.

Soit la grille suivante :

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

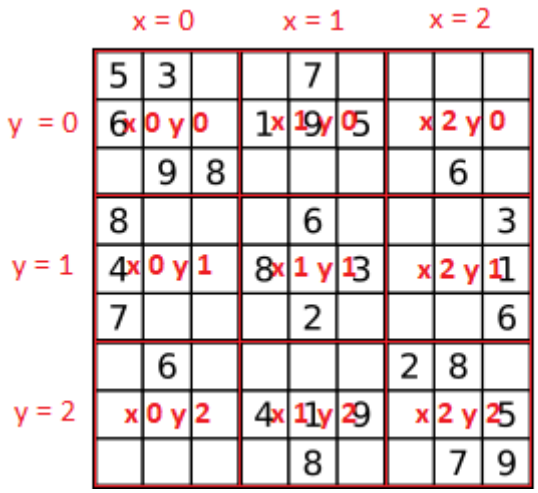
La matrice correspondante à cette grille est de cette forme :

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

Vous devez écrire le corps des fonctions suivantes :

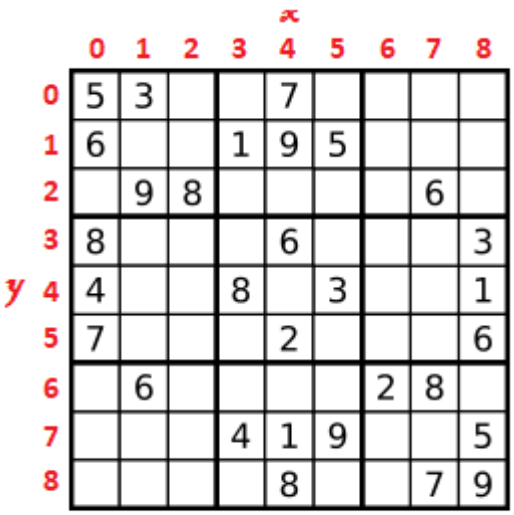
- *get_row(grid, row_number)* : qui retourne une liste contenant la row_numer ième ligne de la grille, avec les lignes numérotées de 0 à 8 (inclus) en partant du haut.
- *get_column(grid, column_number)* : qui retourne une liste contenant la column_number ième colonne de la grille, avec les colonnes numérotées de 0 à 8 (inclus) en partant de la gauche.
- *get_square(grid, square_pos_x, square_pos_y)* : qui retourne le carré à la position x et y dans la grille. x et y varient entre 0 et 2 (inclus), positionnés dans le sens de lecture.

Pour plus de clarté, les positions pour la fonction *get_square* fonctionnent comme ceci :



Les carrés de la grille identifiés par leur position x et y.

Les lignes (y) et colonnes (x) sont quant à elles numérotées comme ceci :



Exemples :

```
>>> get_row(grid, 1)
[6, 0, 0, 1, 9, 5, 0, 0, 0]
>>> get_column(grid, 8)
[0, 0, 0, 3, 1, 6, 0, 5, 9]
>>> get_square(grid, 0, 2)
[[0, 6, 0], [0, 0, 0], [0, 0, 0]]
```

3. Palier 3 : vérifier les caractéristiques de la grille à l'aide de nos fonctions

Nous allons maintenant pouvoir, encore à l'aide des fonctions déjà réalisées, faire des vérifications au niveau de la grille et non plus seulement au niveau des lignes, colonnes et carrés de façon séparée.

Note : Vous pouvez faire une division euclidienne à l'aide des opérateurs // et %. a // b donne le quotient de a divisé par b, et a % b donne le reste de la division de a par b. Par exemple, 42 // 10 est égal à 4, et 42 % 10 est égal à 2. Cela vous sera utile pour localiser le carré contenant une certaine case.

Vous devez écrire le corps des fonctions suivantes :

- *can_insert(grid, row, column, digit)* : qui vérifie s'il est possible d'insérer un chiffre à une case donnée dans la grille tout en gardant la dite grille valide. Pour cela, le chiffre doit donc pouvoir être inséré dans sa ligne, sa colonne, et son carré. La case visée ne doit pas nécessairement être vide. Le cas où l'on essaye d'insérer un chiffre identique à celui déjà présent dans la case ne sera pas testé.
- *is_valid(grid)* : qui vérifie si une grille est valide.
- *is_filled(grid)* : qui vérifie si toutes les cases d'une grille sont remplies.
- *is_finished(grid)* : qui vérifie si la grille est remplie **et** valide.

Exemples :

```
>>> can_insert(grid, 2, 0, 2)
True
>>> can_insert(grid, 2, 0, 5)
False
>>> is_valid(grid)
True
>>> x = 2
>>> y = 0
>>> grid[y][x] = 5
>>> is_valid(grid)
False
>>> is_filled(grid)
False
>>> is_finished(grid)
False
```

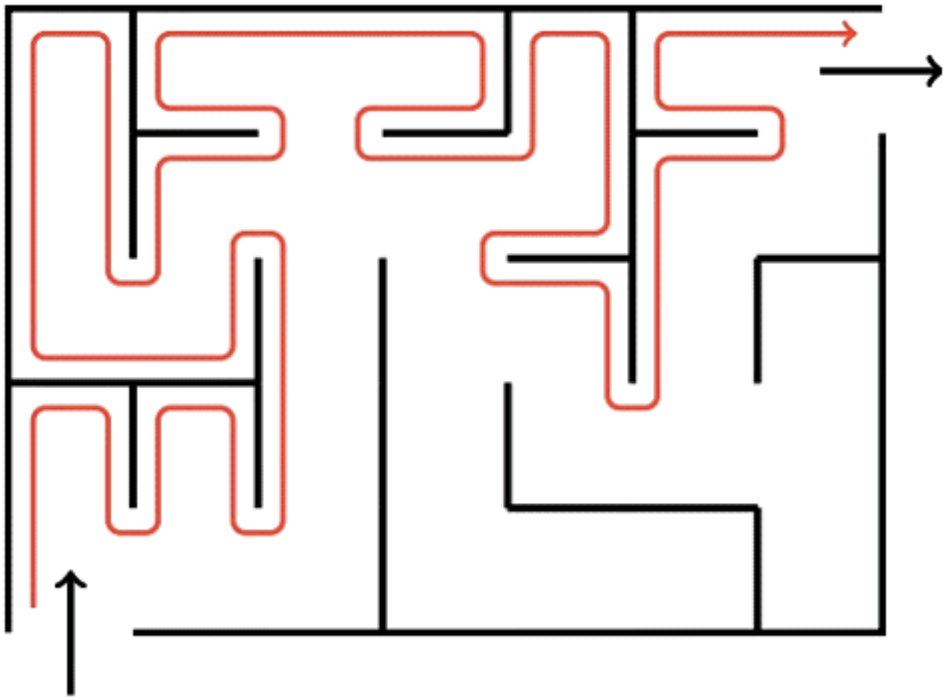
4. Palier 4 : Résoudre le sudoku automatiquement grâce au retour sur trace et à la récursivité

Tout d'abord, qu'est-ce que le retour sur trace ? Citons Wikipedia :

> *« Le retour sur trace (appelé aussi backtracking en anglais) est une famille d'algorithmes qui consistent à revenir 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. » - Page du retour sur trace sur Wikipédia*

Plus simplement, il s'agit, afin de trouver une solution à un problème, d'essayer une solution, et de revenir un peu en arrière chaque fois que l'on s'est planté.

Imaginez-vous dans un labyrinthe. Une bonne façon pour trouver la sortie sans se perdre est de parcourir le labyrinthe en laissant toujours votre main gauche collée au mur du labyrinthe. C'est ce qui s'appelle en informatique un parcours main gauche.

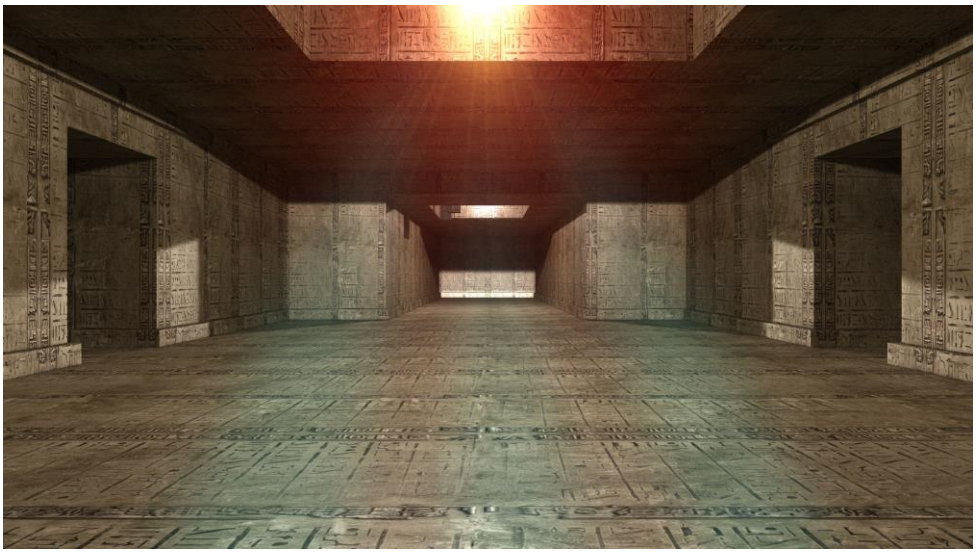


Un parcours main gauche dans un labyrinthe.

Une autre façon de décrire cette même technique est de dire que l'on prend d'abord toujours à gauche, et arrivé à un cul de sac, on recule jusqu'à la dernière intersection. Là on décide de prendre plutôt à droite (pour changer), puis on se remet à prendre toujours à gauche, jusqu'au prochain cul de sac. Il peut arriver qu'en reculant on revienne à une intersection où l'on a visité chaque

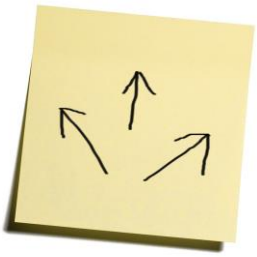
embranchement. Dans ce cas là, on continue simplement de reculer. C'est précisément la même chose qu'un parcours main gauche, sauf que la contrainte de ne jamais vous retourner implique de vous souvenir quels chemins vous avez essayé.

Admettons maintenant que vous avez sur vous des post-its et un stylo.



Vous arrivez là
Droit d'auteur: mik38 / 123RF Banque d'images

Vous commencez par noter sur un post-it qu'il y a trois chemins possibles.



Votre post-it

Conformément à votre technique, vous prenez à gauche. Vous vous apercevez qu'il s'agit d'un cul de sac, vous revenez à l'intersection numéro 1. Vous barrez le chemin de gauche, il ne mène nulle part.

Votre post-it mis à jour

Vous décidez de vous avancer vers le chemin viable le plus à gauche, c'est-à-dire ici le chemin vers l'avant. Vous avancez, et arrivez sur un croisement similaire. Vous prenez un autre post-it, écrivez trois flèches, et **empilez** ce post-it sur le premier.

Votre deuxième post-it, posé sur le premier

Vous essayez chaque chemin. Chaque fois qu'un chemin débouche sur un cul de sac, vous le barrez et en essayez un autre. Ils ne mènent tous à rien, vous avez barré les trois chemins. Il n'y a plus rien à voir ici. Vous décidez de **dépiler** ce post-it et rebrousser chemin. Vous barrez le chemin du centre du premier post-it.

Vous êtes revenus au premier post-it

Vous prenez le chemin de droite, arrivez à une nouvelle intersection de trois chemins. Vous reprenez donc un post-it, dessines trois flèches, vous **l'empilez**. Vous prenez le chemin de gauche en premier, victoire, c'était la sortie. Vous avez trouvé la sortie du labyrinthe par retour sur trace.

Il est fort commode d'effectuer un retour sur trace de façon récursive. Chaque chemin visité / post-it empilé représente un appel de fonction « chercher la sortie ce chemin là » et chaque post-it dépilé représente la fin d'un appel de fonction

Pour visiter un chemin, vous devez visiter tous les sous-chemins auquel il mène. On voit que la définition de « visiter un chemin » est récursive.

Revenons-en au solveur de sudoku. La technique de résolution qui va être présentée ici est également basée sur le retour sur trace. Vous allez, case par case, essayer de remplir la grille en faisant des hypothèses. Si dans la première case vide vous pouvez mettre un 1, un 2 ou un 7, ces trois chiffres sont l'équivalent de nos trois chemins de labyrinthe. Il va falloir essayer de résoudre le sudoku d'abord en essayant de le résoudre avec un 1, et si on y arrive pas, en essayant de résoudre le sudoku avec un 2 dans cette case, etc.

Vous allez devoir, pour ce palier, écrire la fonction *solve(grid)*. Cette fonction renvoie True si la grille est effectivement résolue, et False s'il n'y avait pas de solution possible.

Exemple :

```
>>> is_finished(grid)
False
>>> solve(grid)
True
>>> is_finished(grid)
True
```

Félicitations ! Vous êtes venus à bout de ce TP. N'oubliez pas de le rendre. Si vous avez eu des difficultés, pensez à les indiquer dans un fichier README.txt que nous lirons attentivement. Un code qui ne marche pas mais qui montre vos essais vaut plus que pas de code du tout.

N'hésitez pas à appeler un assistant lors du TP, ou à envoyer un mail à `supbiotech-bioinfo-bt1@googlegroups.com` si celui-ci est terminé.