Sudoku

Marc Lorenzi - 10 mars 2018

In [1]:
```
import random

count = 0
```

1. Représentation des données

On représente un problème de Sudoku par une matrice 9x9. Les cases inconnues sont remplies par des zéros. Ci-dessous deux problèmes faciles.

In [2]:
```
pb01 = [
 [0,3,0,0,6,0,1,0,5],
 [0,2,0,4,0,0,3,9,0],
 [0,0,7,0,3,0,4,0,0],
 [0,6,1,0,5,0,9,0,0],
 [8,0,2,7,0,6,5,0,1],
 [0,0,5,0,4,0,7,6,0],
 [0,0,4,0,8,0,2,0,0],
 [0,7,8,0,0,1,0,4,0],
 [2,0,3,0,7,0,0,5,0]]
```

In [3]:
```
pb02 = [
[0,6,9,2,0,7,4,0,0],
[0,0,1,9,0,0,0,0,0],
[2,0,0,0,0,0,0,6,0],
[0,1,0,6,0,0,9,0,0],
[7,0,0,1,0,2,0,0,4],
[0,0,5,0,0,3,0,7,0],
[0,2,0,0,0,0,0,0,6],
[0,0,0,0,0,4,3,0,0],
[0,0,4,5,0,1,7,9,0]
]
```

Les problèmes `harder` et `ai_escargot` ci-dessous sont réputés très difficiles. Voir ce site (http://aisudoku.com/index_en.html). Nous allons voir que leur réputation est surfaite.

In [4]:
```
harder = [
[8,0,0,0,0,0,0,0,0],
[0,0,3,6,0,0,0,0,0],
[0,7,0,0,9,0,2,0,0],
[0,5,0,0,0,7,0,0,0],
[0,0,0,0,4,5,7,0,0],
[0,0,0,1,0,0,0,3,0],
[0,0,1,0,0,0,0,6,8],
[0,0,8,5,0,0,0,1,0],
[0,9,0,0,0,0,4,0,0]
]
```

In [5]:
```
ai_escargot = [
[1,0,0,0,0,7,0,9,0],
[0,3,0,0,2,0,0,0,8],
[0,0,9,6,0,0,5,0,0],
[0,0,5,3,0,0,9,0,0],
[0,1,0,0,8,0,0,0,2],
[6,0,0,0,0,4,0,0,0],
[3,0,0,0,0,0,0,1,0],
[0,4,0,0,0,0,0,0,7],
[0,0,7,0,0,0,3,0,0]
]
```

Rajoutons à cela le problème vide, qui n'est pas un vrai problème pouisqu'il ne possède pas vraiment une unique solution :-).

In [6]:
```
vide = [
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,0,0]
]
```

2. Conversions entre problèmes et grilles

Lors de la résolution du problème on a besoin, pour chaque case, de mémoriser les possibilités pour remplir cette case. Nous appellerons dans la suite "grille" une matrice 9x9 contenant à la ligne i, colonne j la liste des possibilités pour remplir la case (i, j).

Voici une fonction qui permet de transformer un problème en grille. Tout d'abord, on remplit les cases de la grille par la liste des entiers de 1 à 9. Puis, pour chaque case connue du problème, on impose sa valeur dans la grille.

La fonction `imposer`, qui reste à écrire, force la case (i, j) de la grille à une valeur c. Puis elle élimine de la ligne i, de la colonne j, et du carré 3x3 contenant la case (i, j), la possibilité c.

In [7]:
```
def probleme_vers_grille(probleme):
    g = 9 * [None]
    for k in range(9): g[k] = 9 * [None]
    for i in range(9):
        for j in range(9):
            g[i][j] = list(range(1, 10))
    for i in range(9):
        for j in range(9):
            c = probleme[i][j]
            if c != 0:
                imposer(g, i, j, c)
    return g
```

La fonction réciproque prend en paramètre une grille censée être résolue : toutes les cases de la grille contiennent des listes à 1 élément. Elle renvoie le problème correspondant. Elle lève une exception en cas de case de la grille qui n'est pas un singleton.

In [8]:
```
def grille_vers_probleme(g):
    pb = [[0 for i in range(9)] for j in range(9)]
    for i in range(9):
        for j in range(9):
            if len(g[i][j]) != 1: raise Exception
            else: pb[i][j] = g[i][j][0]
    return pb
```

3. Affichage

La fonction `afficher_probleme` ... affiche joliment notre problème de Sudoku.

In [9]:
```
def afficher_probleme(pb):
    s = ''
    for i in range(9):
        for j in range(9): s += ('%2d' % pb[i][j])
        if i != 8: s += '\n'
    print(s)
```

In [10]:
```
afficher_probleme(ai_escargot)
```

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

4. Résolution

4.1 La fonction de résolution

Nous y voilà. La fonction `resoudre` prend une grille g en paramètre.

- On commence par faire un peu de ménage en regardant les cases de la grille qui sont des singletons (ménage sur ligne, colonne, carré).
- Puis on choisit une case de la grille qui contient le moins possible d'éléments (le cas des singletons étant retenu en dernière extrémité).
- Si la case retenue est un singleton c'est que le problème est résolu.
- Si la case retenue est vide c'est que le problème est impossible.
- Sinon, on fait une copie toute neuve, g_1 , de la grille g. On sélectionne une valeur c au hasard parmi toutes les possibilités pour $g[i][j]$ et on l'impose dans g_1 . On appelle ensuite récursivement `resoudre` sur g_1 . Si cela réussit, tant mieux, on a fini. Sinon, on retire la possibilité c pour la case (i, j) de g et on appelle résoudre sur le g ainsi légèrement modifié.

In [11]:
```
def resoudre(g):
    global count
    count = count + 1
    for (i,j) in random_walk():
        if len(g[i][j]) == 1:
            c = g[i][j][0]
            imposer(g, i, j, c)
    i, j = choisir_case(g)
    if (i, j) == (-1, -1): return (g, True) # Grille résolue
    elif len(g[i][j]) == 0: return (g, False) # Grille impossible
    else:
        g1 = copie(g)
        c = choisir_hasard(g1[i][j])
        imposer(g1, i, j, c)
        (g2, b) = resoudre(g1)
        if not b: # échec !
            g[i][j].remove(c) # car c ne convient pas
            return resoudre(g)
        else:
            return (g2, True)
```

4.2 Copie d'une grille

Sans commentaire, voici la fonction de copie.

In [12]:
```
def copie(g):
    g1 = [[None for i in range(9)] for j in range(9)]
    for i in range(9):
        for j in range(9):
            g1[i][j] = g[i][j][:]
    return g1
```

4.3 Choix d'une case de la grille

La fonction `choisir_case` choisit une case de la grille ayant un nombre minimal d'éléments, mais si possible pas un seul. Elle renvoie les coordonnées de la case, sauf ti toutes les cases ont un seul élément (problème résolu !). Dans ce cas, la fonction renvoie le couple $(-1, -1)$. La grille est parcourue au hasard pour donner un peu plus de chance à la chance.

In [13]:
```
def random_list(n):
    s = list(range(n))
    for i in range(n):
        j = random.randint(0, i)
        s[i], s[j] = s[j], s[i]
    return s
```

In [14]:
```
print(random_list(81))
```

```
[76, 62, 60, 45, 0, 31, 77, 6, 58, 74, 50, 38, 17, 5, 65, 66, 29, 9,
16, 4, 25, 73, 72, 71, 41, 52, 47, 42, 20, 15, 39, 28, 10, 40, 18, 8
, 64, 21, 19, 1, 63, 70, 56, 33, 59, 51, 80, 54, 43, 27, 44, 14, 67,
12, 22, 7, 79, 32, 46, 3, 53, 68, 24, 23, 75, 37, 69, 11, 49, 78, 30
, 36, 2, 48, 35, 26, 13, 57, 55, 34, 61]
```

Voici une fonction qui renvoie un parcours au hasard des cases d'un tableau 9x9.

In [15]:
```
def random_walk():
    s = random_list(81)
    return [(x // 9, x % 9) for x in s]
```

In [16]:
```
print(random_walk())
```

```
[(4, 0), (2, 8), (6, 3), (4, 8), (7, 8), (6, 0), (7, 7), (5, 8), (0,
0), (5, 2), (6, 8), (0, 5), (1, 0), (0, 2), (6, 7), (2, 7), (6, 1),
(8, 7), (8, 2), (4, 1), (4, 3), (2, 5), (1, 8), (1, 3), (4, 6), (0,
7), (7, 3), (5, 7), (5, 4), (3, 7), (3, 3), (5, 6), (2, 6), (8, 6),
(5, 5), (7, 1), (5, 1), (3, 8), (4, 5), (7, 4), (6, 2), (4, 7), (7,
0), (8, 5), (1, 1), (0, 6), (7, 5), (1, 6), (6, 4), (3, 1), (3, 6),
(2, 2), (5, 0), (0, 8), (7, 2), (2, 1), (7, 6), (6, 5), (2, 0), (5,
3), (8, 0), (3, 2), (3, 0), (8, 8), (0, 4), (6, 6), (8, 1), (1, 2),
(2, 4), (8, 4), (0, 3), (3, 4), (0, 1), (2, 3), (8, 3), (1, 4), (1,
7), (1, 5), (4, 2), (3, 5), (4, 4)]
```

In [17]:
```
def note(g, i, j):
    l = len(g[i][j])
    if l == 1: return 1000000
    else : return l
```

In [18]:
```
def choisir_case(g):
    u, v = 0, 0
    for (i, j) in random_walk():
        nij = note(g, i, j)
        nuv = note(g, u, v)
        if nij <= nuv: u, v = i, j
    if note(g, u, v) == 1000000: u, v = -1, -1
    return (u, v)
```

Fonction ci-dessous : RAS :-)

In [19]:
```
def choisir_hasard(s):
    k = random.randint(0, len(s) - 1)
    return s[k]
```

In [20]:
```
choisir_hasard([1,7,2,8,5,3])
```

Out[20]: 3

4.4 Imposer une valeur à une case

Dernière ligne droite : imposer une valeur à une case. Facile ...

In [21]:
```
def imposer(g, i, j, c):
    g[i][j] = [c]
    ajuster_ligne(g, i, j, c)
    ajuster_colonne(g, i, j, c)
    ajuster_carre(g, i, j, c)
```

... euh oui, facile lorsqu'on aura écrit les fonctions d'ajustement !

In [22]:
```
def ajuster_ligne(g, i, j, c):
    for k in range(9):
        if k != j and c in g[i][k]:
            g[i][k].remove(c)
```

In [23]:
```
def ajuster_colonne(g, i, j, c):
    for k in range(9):
        if k != i and c in g[k][j]:
            g[k][j].remove(c)
```

In [24]:
```
def ajuster_carre(g, i, j, c):
    for (k, l) in carre(i, j):
        if (k != i or l != j) and c in g[k][l]:
            g[k][l].remove(c)
```

In [25]:
```
def carre(i, j):
    coin_x = 3 * (i // 3)
    coin_y = 3 * (j // 3)
    return [(coin_x + k, coin_y + l) for k in range(3) for l in range(
```

5. Tests

Nous sommes maintenant en mesure de tester notre fonction de résolution. Tant qu'à faire, on encapsule toutes les étapes dans une unique fonction que nous appellerons `solution` .

In [26]:
```
def solution(probleme):
    g = probleme_vers_grille(probleme)
    count = 0
    g1, b = resoudre(g)
    if not b:
        print("Pas de solution")
    else:
        afficher_probleme(grille_vers_probleme(g1))
```

On tente un problème facile ?

In [27]:
```
solution(pb01)
```

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

Réponse donnée en 0 seconde. On essaie plus difficile ?

In [28]:
```
solution(pb02)
```

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

Le problème ci-dessous est réputé difficile.

In [29]:
```
solution(harder)
```

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

La solution est donnée en moins de 10 secondes, au pire.

Selon ce site (http://aisudoku.com/index_en.html), voici un problème vraiment très très difficile. Bof.

In [30]:
```
solution(ai_escargot)
```

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

Difficile pour un humain ... mais pas pour un python :-).

6. Derniers doutes levés

Après quelques grilles vérifiées à l'oeil, je me décide. La fonction `verifier` prend un problème en paramètre et sa solution (?) et teste si le problème est vraiment résolu.

In [31]:
```
def verifier(pb, sol):
    b = True
    comparer(pb, sol)
    for i in range(9): b = b and verifier_ligne(sol, i)
    for j in range(9): b = b and verifier_colonne(sol, j)
    for x in range(3):
        for y in range(3):
            b = b and verifier_carre(sol, x, y)
    return b
```

In [32]:
```
def comparer(pb, sol):
    b = True
    for i in range(9):
        for j in range(9):
            if pb[i][j] != 0:
                b = b and pb[i][j] == sol[i][j]
    return b
```

In [33]:
```
def verifier_ligne(pb, i):
    s = 0
    for j in range(9):
        s = s + pb[i][j]
    return s == 45
```

In [34]:
```
def verifier_colonne(pb, j):
    s = 0
    for i in range(9):
        s = s + pb[i][j]
    return s == 45
```

In [35]:
```
def verifier_carre(pb, x, y):
    s = 0
    for i in range(3):
        for j in range(3):
            s = s + pb[3 * x + i][3 * y + j]
    return s == 45
```

In [36]:
```
def solution_verifiee(probleme):
    g = probleme_vers_grille(probleme)
    g1, b = resoudre(g)
    if not b:
        print("Pas de solution")
    else:
        sol = grille_vers_probleme(g1)
        afficher_probleme(sol)
        print('')
        if verifier(probleme, sol): print('Grille résolue !!!')
        else: print('Aie aie aie, l''impossible s''est produit.')
```

In [37]:
```
%%time
solution_verifiee(ai_escargot)
```

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

Grille résolue !!!
CPU times: user 535 ms, sys: 2.43 ms, total: 537 ms
Wall time: 537 ms
```

In [38]:
```
%%time
count = 0
solution_verifiee(harder)
print("Nombre d'appels à resoudre : ", count)
```

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

Grille résolue !!!
Nombre d'appels à resoudre :  1115
CPU times: user 1.17 s, sys: 3.34 ms, total: 1.17 s
Wall time: 1.17 s
```

7. Des milliers de grilles ...

Le fichier joint `sudoku17.txt` contient 49151 problèmes de Sudoku avec 17 cases connues au départ. 17 est à ce jour le nombre minimum connu de cases nécessaires pour qu'un problème de Sudoku ait une solution unique. Il se peut qu'on trouve mieux un jour ...

Certains de ces problèmes donnent du fil à retordre à l'algorithme, le numéro 39949 par exemple (plus d'une minute de recherche pour obtenir la solution).

La fonction `get_sudoku` prend un entier k en paramètre et renvoie le $k$ième problème du fichier.

In [39]:
```
def get_sudoku(k):
    lines = open('sudoku17.txt', 'r').readlines()
    s = lines[k]
    pb = [[0 for i in range(9)] for j in range(9)]
    for i in range(9):
        for j in range(9):
            pb[i][j] = int(s[9*i+j])
    return pb
```

In [40]:
```
%%time
pb = get_sudoku(2018)
afficher_probleme(pb)
print()
solution_verifiee(pb)
```

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

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

Grille résolue !!!
CPU times: user 4.69 s, sys: 16.9 ms, total: 4.71 s
Wall time: 4.74 s
```

La fonction `random_sudoku` renvoie un problème pris au hasard dans le fichier, ainsi que le numéro de ce problème.

In [41]:
```
def random_sudoku():
    k = random.randint(0, 49150)
    return (get_sudoku(k), k)
```

In [43]:
```
%%time
pb, k = random_sudoku()
print('Problème numéro ', k)
afficher_probleme(pb)
print()
solution_verifiee(pb)
```

```
Problème numéro  8145
 0 0 0 3 0 0 0 1 0
 0 7 0 0 9 0 0 0 0
 2 0 0 0 0 0 0 0 0
 0 4 0 0 0 0 6 0 7
 0 0 5 1 0 0 0 0 0
 0 0 0 8 0 0 0 0 0
 1 0 3 0 0 0 0 0 0
 0 0 0 0 7 0 5 0 0
 8 6 0 0 0 0 0 0 0

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

Grille résolue !!!
CPU times: user 611 ms, sys: 9.15 ms, total: 620 ms
Wall time: 618 ms
```