TP Informatique : Manipulation de tableaux

1 Carré magiques

Un carré magique est un tableau à deux dimensions de taille $n \times n$ contenant des entiers tous distincts, où les sommes des entiers composant chaque ligne, chaque colonne et chacune des deux diagonales donnent toutes le même résultat (la somme de tous les éléments du tableau divisé par n).

Fréquemment, on demande également que, pour un carré magique de taille n, les entiers contenus dans le tableau soient les entiers de 1 à n^2, et c'est une condition que l'on exigera de nos carrés magiques. Dans cette situation, la somme sera nécessairement $n \times (n^2+1)/2$.

Ainsi, le tableau 3×3 suivant :

2	7	6
9	5	1
4	3	8

est un carré magique, car

$$2+7+6 = 9+5+1 = 4+3+8 = 2+9+4 = 7+5+3 = 6+1+8 = 2+5+8 = 4+5+6 = 15$$

1.a Écrire une fonction `SommeLigne(tab, i)` qui prend en argument un `numpy.array` `tab` et un entier `i` et retourne la somme des termes de la ligne d'index `i` de `tab`. On supposera que `tab` est un tableau à deux dimensions, et que i est positif strictement inférieur au nombre de lignes de `tab`.

1.b Écrire de même une fonction `SommeColonne(tab, i)`.

2. Écrire une fonction `ElementsCorrects(tab)` qui vérifie que les éléments du `numpy.array` passé en argument sont bien les éléments de 1 à n^2 (où n est le nombre de lignes du tableau). On peut vérifier la présence d'un élément `x` dans un `numpy.array` désigné par `tab` en écrivant simplement `x in tab`, comme on le ferait pour une liste.

3.a En utilisant les fonctions précédentes, écrire une fonction `EstMagique(tab)` qui renvoie `True` si le `numpy.array` contient un carré magique, et `False` dans le cas contraire. On n'oubliera pas de vérifier les diagonales !

3.b Vérifier que le tableau 3×3 fourni au début est magique.

4.a Écrire une fonction `Aleatoire3x3()` qui génère aléatoirement un `numpy.array` de taille 3×3 contenant les entiers de 1 à 9.

La fonction `random.sample(range(1, 10), 9)` qui retourne une liste contenant les entiers de 1 à 9 (inclus) dans un ordre aléatoire pourra sans doute ici vous être utile.

4.b Compter combien de tableaux, parmi 50000 générés aléatoirement, sont magiques.

4.c Combien pensez-vous qu'il existe de carrés magiques de taille 3×3 ?

5.a Écrire une fonction `Gen4()` qui retourne le `numpy.array` de taille 4×4 dans lequel $A_{i,j} = 1 + 4 \times i + j$ si $i \neq j$ et $i + j \neq 3$ (en commençant la numérotation à 0 pour i et j) et $A_{i,j} = 16 - 4 \times i - j$ sinon.

5.b Vérifier que l'on obtient ainsi un carré magique.

6.a Écrire une fonction `Gen(n, k)` qui retourne un `numpy.array` de taille $n \times n$ où

$$A_{i,j} = 1 + \big((i + k \times j) \bmod n\big) + \big((j + k \times i) \bmod n\big) \times n$$

6.b Vérifier qu'il existe des valeurs de n et k donnant des carrés magiques.

2 Automate de Conway (jeu de la vie)

Le jeu de la vie est un automate cellulaire évolutif, imaginé par John Horton Conway en 1970, basé sur des règles très simples, et qui pourtant fait émerger des comportements étonamment complexes.

On considère un tableau à deux dimensions de taille $n \times n$, ne contenant que des 0 et des 1. On peut considérer que les « 1 » dans le tableau représentent des cellules vivantes.

À partir de ce tableau, on calcule un *nouveau* tableau donc les cases contiennent des 1 si et seulement si :

- la case correspondante du tableau précédent contenait un 1 et deux ou trois de ses huit voisines contenaient également un 1 (mort des cellules isolées et des cellules « asphyxiées » par un trop grand nombre de voisines) ;
- la case correspondante du tableau précédent contenait un 0 et exactement trois de ses huit voisines contenaient également un 1 (« reproduction » des cellules).

1. Écrire une fonction `CompteVoisines(tab, i, j)` qui compte, parmi les huit cases voisines de la case (i, j) dans le tableau `tab`, le nombre de cases contenant un 1. Pour éviter les problèmes aux bords, on supposera le tableau « torique », c'est-à-dire que la voisine de gauche d'une case de la colonne de gauche est la case, sur la même ligne, dans la colonne de droite et inversement (et de même pour la ligne du haut et celle du bas).

On pourra donc utiliser `(colonne-1 % nbcol)` et `(colonne+1 % nbcol)` pour désigner les index de colonnes des voisines latérales (et même chose pour les colonnes).

2. Écrire une fonction `Next(tab)` qui prend en argument un `numpy.array` rempli de 0 et de 1, et retourne un *nouveau* `numpy.array` de même taille contenant des 0 et des 1 selon les règles d'évolution énoncées ci-dessus. Pourquoi est-il indispensable de construire un nouveau `numpy.array` plutôt que de modifier le précédent ?

3. Écrire une fonction `Init(nblignes, nbcolonnes, p)` qui retourne un `numpy.array` de taille `nblignes` × `nbcolonnes` contenant aléatoirement des 0 et des 1, la probabilité de

trouver un 1 étant égale à p (un flottant entre 0.0 et 1.0).

On pourra ensuite profiter d'une animation basée sur cet automate grâce au programme suivant :

```
import matplotlib.pyplot as plt
import matplotlib.animation as ani

T = Init(40, 70, 0.15)

img = plt.matshow(T, vmin=0, vmax=1, cmap="gnuplot")

def Update(i) :
    T[:] = Next(T)
    img.set_array(T)
    return [ img ]

anim = ani.FuncAnimation(plt.gcf(), Update, interval=20, blit=True)

plt.show()
```

Vous pouvez, si vous le souhaitez, changer les règles d'évolution. Celles présentées ici sont celles auxquelles est parvenu John H. Conway après de nombreux essais, et c'est un des rares jeux de règles qui n'aboutissent pas rapidement soit à une extinction, soit à une explosion démographique.

En fait, cet automate est tellement riche qu'il permet de créer des portes logiques ET, OU et NON, ce qui le rend Turing-complet : n'importe quel problème algorithmique peut être écrit grâce à cet automate ! Une autre conséquence est que déterminer l'avenir de la population (extinction, stabilité, etc.) à partir de l'état initial est un problème indécidable.

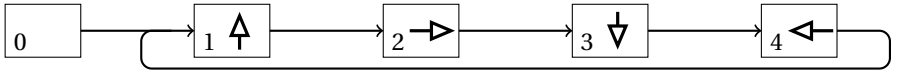
3 Rotorouteurs (ou Rotor-routers)

Les rotorouteurs sont un autre type d'automate, beaucoup plus récent (proposé par Jim Propp vers 2000). Ils sont liés aux problèmes, importants dans beaucoup de domaines des sciences, de « marche au hasard », mais sont parfaitement déterministes.

On considère une grille, initialement remplie de zéros, de taille $n \times n$. On peut poser sur une case quelconque de la grille un « panneau » avec une flèche indiquant une direction (vers le haut, vers la droite, vers le bas ou vers la gauche). Une case contenant un panneau contiendra les valeurs 1, 2, 3 ou 4 pour chacune des directions précitées.

Un automate va se « promener » sur la grille. Sa position initiale est `(n//2, n//2)`. À chaque itération, il obéit aux règles suivantes :

- si la case dans laquelle il se trouve est vide (c'est-à-dire contient 0), il place un panneau désignant le haut dans la case (soit 1) et retourne à sa position de départ.
- si la case n'est pas vide, il tourne le panneau d'un quart de tour dans le sens horaire, puis se déplace dans la direction indiquée par la flèche.



1. Construire un `numpy.array` désigné par `T`, de taille 61×61, contenant intégralement des zéros.

2. Érire une fonction `Step(T, x, y)` qui prend en argument un `numpy.array` `T` et des coordonnées `x` et `y` pour un mobile, modifie le tableau `T` et retourne la nouvelle position du mobile, selon les règles précédentes.

L'instruction `x, y = Step(T, x, y)` permet donc effectuer une étape de l'automate.

3. En partant du centre du tableau, c'est-à-dire de la position `(30, 30)`, effectuer en boucle l'instruction précédente jusqu'à ce que le mobile atteindre une ligne ou une colonne du bord du tableau (on peut montrer que cela arrive forcément), puis afficher le résultat avec

```
from matplotlib.pyplot import imshow, show
from matplotlib import colors

imshow(T, interpolation='nearest', vmin=-1, vmax=4,
       cmap=colors.ListedColormap(
           ['black', 'white', 'red', 'yellow', 'green', 'blue'] ))
show()
```

Les cases blanches sont celles qui n'ont pas été visitées. Les panneaux indiquant le haut sont rouges, ceux indiquant la droite sont jaunes, les panneaux indiquant le bas sont vert et ceux indiquant la gauche sont bleus.

Que pensez-vous du résultat obtenu ?

Les rotorouteurs ont la propriété intéressante de visiter tout l'espace qui leur est mis à disposition. Ils peuvent donc naturellement explorer intégralement un graphe, ou bien... trouver la sortie d'un labyrinthe.

Pour explorer un labyrinthe (le fichier `Labyrinthe.py` contient un `numpy.array` permettant de faire un essai), on effectue les quelques modifications suivantes aux règles énoncées ci-dessus :

- le tableau contient initialement des cases libres (0) mais aussi des « murs » (-1) ;
- on ne retourne pas au départ lorsque l'on pose un nouveau panneau, on reste sur place ;
- si la case vers laquelle on tente de se déplacer est un mur, on reste sur place.

On peut vérifier que l'automate parvient toujours à sortir !