Séance 10: TROUVER SON CHEMIN DANS UN LABYRINTHE

Université Paris-Diderot

Objectifs:

— Travailler avec des listes de listes

— Utiliser les boucles `for` et `while`

Dans ce TP nous allons travailler avec des listes de listes d'entiers qui représentent un labyrinthe. À l'aide des boucle `while` et `for` nous allons implémenter un simple algorithme permettant de calculer le nombre minimal de cases à parcourir pour aller du coin haut gauche au coin bas droit dans le labyrinthe.

Rédiger les réponses aux questions et les tests pour chaque réponse dans le fichier *labyrinthe.py* qui vous est fourni.

Dans le fichier *labyrinthe1.csv* vous est donnée la description d'un labyrinthe. On vous donne également dans *labyrinthe.py* une fonction `chargeLabyrinthe` qui prend en argument un nom de fichier contenant la description d'un labyrinthe et revoie la liste de listes d'entiers correspondant au labyrinthe. Ainsi l'instruction `chargeLabyrinthe('labyrinthe1.csv')` renverra une liste de listes d'entiers.

Dans cette liste, chaque sous-liste représente une ligne du labyrinthe, la première liste représente la première ligne, la deuxième liste la deuxième ligne, etc. Pour le moment cette liste ne contient que des 0 et des 1, mais nous verrons que nous y stockerons d'autres valeurs entières positives. L'idée est qu'un 0 représente un mur dans le labyrinthe et une valeur strictement positive une case libre.

Le but est de savoir si il existe un chemin dans le labyrinthe depuis la position en haut à gauche jusqu'à la position en bas à droite. Pour se déplacer dans le labyrinthe, on peut seulement aller vers le haut, vers le bas, à gauche et à droite sans passer par des cases où se trouvent des murs.

Pour se faire l'algorithme commence par écrire 2 dans la case en haut à gauche, ensuite pour chaque case voisine écrite avec un 1, il met un 3, ensuite pour chaque case écrite contenant un 3, il met un 4 dans chaque voisine contenant un 1, etc L'algorithme s'arrête soit si il a changé l'entier dans la case en bas à droite soit si il n'a plus de cases à changer. Voilà un exemple des différentes étapes.

1	1	1	1
0	1	0	1
0	1	1	1

2	1	1	1
0	1	0	1
0	1	1	1

2	3	1	1
0	1	0	1
0	1	1	1

2	3	4	1
0	4	0	1
0	1	1	1

2	3	4	5
0	4	0	1
0	5	1	1

2	3	4	5
0	4	0	6
0	5	6	1

2	3	4	5
0	4	0	6
0	5	6	7

Dans cet exemple, à la fin de l'algorithme la case en bas à droite contient un 7 donc on peut en déduire qu'il faut traverser 6 cases pour traverser le labyrinthe depuis la case en haut à gauche.

Exercice 1 (À vous de jouer !, ⋆-⋆⋆)

Dans ce qui suit, le labyrinthe désignera une liste de listes d'entiers. **Testez vos fonctions régulièrement avec le labyrinthe qui vous est fourni.**

1. *Écrire une fonction* `afficheLab` *qui prend en argument un labyrinthe et qui l'affiche dans le terminal ligne par ligne en mettant un caractère* `'X'` *si il y a un mur et un caractère* `' '` *(espace) si il n'y a pas de mur.*
2. *Nous souhaitons travailler sur une copie du labyrinthe. Écrire une fonction* `copieLab` *qui prend en argument un labyrinthe et renvoie une copie de ce labyrinthe.*
3. *Écrire une fonction* `caseHaut` *qui prend arguments un labyrinthe et un numéro de ligne et un numéro de colonnes correspondant à une case (on suppose que ces numéros commençant à 0 sont dans les limites du labyrinthe) et qui renvoie l'entier contenu dans la case au-dessus si celle-ci existe et si elle n'existe pas (dans le cas par exemple où la case donnée en arguments se trouve sur la première ligne), la fonction renvoie -1.*
4. *De la même façon, écrire des fonctions* `caseBas`, `caseGauche` *et* `caseDroite` *qui prennent en arguments un labyrinthe et un numéro de ligne et un numéro de colonnes correspondant à une case et qui renvoie l'entier contenu dans la case au-dessous, respectivement à gauche respectivement à droite si celles-ci existent et si elle n'existent pas, ces fonctions renvoient -1.*
5. *En utilisant les fonctions des questions précédentes, écrire une fonction* `voisinsLibres` *qui prend en arguments un labyrinthe et un numéro de ligne et un numéro de colonnes correspondant à une case et qui renvoie une liste de listes de deux entiers où chaque sous-liste contient deux entiers (numéro de ligne et numéro de colonne) correspondant à une case voisine de la case donnée en arguments et dans laquelle se trouve un 1.*

   Exemple : *Sur le premier labyrinthe dessiné ci-dessus, pour la case (0,0), la fonction* `voisinsLibres` *renverra la liste* `[[0,1]]` *et pour la case (1,1), la fonction* `voisinsLibres` *renverra la liste* `[[0,1], [2,1]]`.
6. *Écrire maintenant une fonction* `changeVoisins` *qui prend en arguments un labyrinthe, deux entiers correspondant à une case (numéro de ligne et numéro de colonne) et un entier* `i` *et qui change le contenu de toutes les cases voisines de la case donnée et qui contiennent un 1 en y mettant à la place la valeur* `i+1`*. Cette fonction ne renvoie rien mais elle change le labyrinthe.*
7. *Écrire maintenant une fonction* `etapeParcours` *qui prend en arguments un labyrinthe un entier* `i`*, qui parcourt toutes les cases du labyrinthe et pour chaque case qui contient l'entier* `i` *elle change ses voisins dans lesquels figurent un 1 en le remplaçant par* `i+1`*. Cette fonction ne renvoie rien mais elle change le labyrinthe.*
8. *Écrire une fonction* `finParcours` *qui servira à tester si le parcours de notre labyrinthe peut s'arrêter. Cette fonction prend en arguments un labyrinthe et renvoie* `True` *si dans la case en bas à droite il y a une valeur différente de 1 ou si toutes les cases contenant une valeur différente de 1 n'ont pas de voisins avec une valeur différente de 1. Dans les autres cas, c'est fonction renverra* `False`*.*
9. *Nous avons maintenant tous les ingrédients pour parcourir notre labyrinthe selon l'algorithme décrit au début de ce TP.*

   Écrire une fonction `parcours` *qui prend en arguments un labyrinthe. Cette fonction commence par écrire un 2 dans la case en haut à gauche du labyrinthe. Ensuite tant que le labyrinthe ne vérifie pas les conditions de fin de parcours données dans la question précédente, elle appelle la fonction* `etapeParcours` *en lui donnant en arguments le labyrinthe et les entiers 2,3,4, etc.*

   Quand cette fonction s'arrête, si le chiffre dans la case en bas à droite est strictement plus grand que 1 alors il correspond au nombre de cases minimal moins un à visiter pour traverser le labyrinthe.

   Pensez à tester votre fonction. Combien de pas faut-il pour traverser le labyrinthe contenu dans labyrinthe1.csv *?*
10. **Bonus :** *On peut aussi se servir de la fonction* `parcours` *pour générer des labyrinthes "corrects" de façon aléatoire. Pour cela, vous générez une liste de listes d'entiers remplies de 0 et vous choisissez un entier* `a`*. Cette liste de listes représentera notre labyrinthe. Vous mettez un 1 dans la case en haut à gauche et ensuite vous tirez* `a` *cases au hasard (rappelons qu'une case est donnée par un numéro de lignes et un numéro de colonnes) et si dans ces cases, il y a un 0 vous le remplacez par un 1. Vous testez ensuite si votre labyrinthe peut être traverser. Si c'est le cas, vous vous arrêtez sinon vous*

recommencer à tirer `a` *cases au hasard et à modifier leur contenu et ainsi de suite jusqu'à avoir un labyrinthe traversable.*

Écrire une fonction `genereLab` *qui prend en arguments trois entiers* `n`, `m` *et* `a` *et qui génère un labyrinthe à* `n` *lignes et* `m` *colonnes en tirant à chaque tour avant de tester si le labyrinthe est traversable* `a` *cases. Cette fonction renvoie un labyrinthe.*

Vous pouvez tester votre fonction avec la fonction `afficheLab`*.*

□