M1102: Introduction à l'algorithmique et à la programmation

Feuille de TD n°11

Le labyrinthe

INSTRUCTIONS: à la fin du TD, vous devez déposer sous Celene une archive zip contenant les fichiers Python correspondant à ce TD (fournis avec la feuille). Vous pourrez déposer une version améliorée de votre travail jusqu'à la **VEILLE** du TD suivant.

L'évaluation de la période 2 sera basée sur ces rendus.

Objectifs

Cette feuille a pour objectif de renforcer la notion d'API de programmation et aussi de travailler sur des aspects algorithmiques sur les matrice.

Exercice 1 *Nouvelles matrices*

Constant Denlechangeman, votre chef de projet, a encore décidé de changer la représentation des matrices. Il veut, cette fois, utiliser un dictionnaire contenant les informations suivantes : le nombre de lignes, le nombre de colonnes et un champ contenant les valeurs (stockées comme vous le souhaitez).
Écrire cette nouvelle implémentation dans le fichier `matriceAPI3.py` et vérifier que vos fonctions des feuilles précédentes sur les matrices marchent toujours.
Constant Denlechangeman voudrait se lancer dans l'implémentation de jeu de plateau. pour cela il a besoin de représenter des *labyrinthe* sous la forme de matrices. Dans la suite une grille 2D va représenter un labyrinthe dans lequel on devra faire déplacer des personnages. Un labyrinthe sera donc une grille 2D dans laquelle les cases qui contiennent la valeur 1 sont considérées comme des murs, les cases contenant des 0 comme des couloirs et les cases contenant une valeur différente de 0 ou 1 sont des couloirs contenant soit un personnage.
La figure 1 montre une matrice et sa représentation sous forme de labyrinthe. Remarquez que les éléments $a_{0,0}$ et $a_{8,8}$ contiennent des personnages.

$$\begin{pmatrix} 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 3 \end{pmatrix}$$

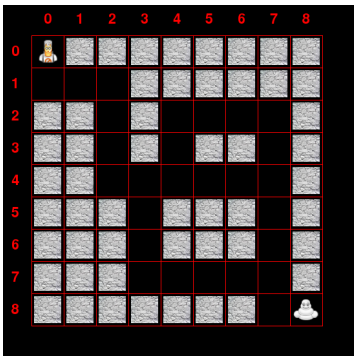

Figure 1 – Une matrice et le labyinthe correspondant

Dans le fichier `labyrinthe.py` vous trouverez l'implémentation des deux fonctions d'entrée-sortie sur les matrice dont `sauveMatrice(mat,nomFic)` et `chargeMatrice(nomFic)` qui permettent respectivement de sauvegarder une matrice dans un fichier texte et de recharger une

matrice à partir d'un fichier texte. Les fonctions à écrire sont à compléter à la fin de ce fichier.
Dans la suite une position est un couple d'entiers.
Les fonctions des deux exercices suivants peuvent être testée grâce au script `test_labyrinthe.py`.

Exercice 2 *Accessibilité dans un labyrinthe*

L'objectif de cet exercice est d'écrire une fonction `estAccessible(mat,pos1,pos2)` qui permet de savoir si on peut atteindre `pos2` à partir de la case `pos1` c'est-à-dire que l'on peut aller de l'une à l'autre en ne passant que par des couloirs. Évidemment si l'une des deux cases est un mur la fonction devra rendre `False`.
L'algorithme que l'on va utiliser s'appelle l'*inondation*. Il consiste à initialiser une matrice (que l'on va appeler calque) de la taille du labyrinthe à 0 sauf la case de départ qui sera mise à 1. Ensuite on parcourt toutes les cases de cette matrice et si une case encore à 0 est voisine d'une case marquée d'un 1, et que cette case est un couloir du labyrinthe on va marquer cette case d'un 1. On recommence le processus jusqu'à soit ne plus arriver à marquer de nouvelles cases soit avoir marqué la position d'arrivée.
Pour arriver à ce résultat on vous demande d'implémenter les fonctions suivantes :

1. `marquageDirect(calque,labyrinthe,val,marque)` : cette fonction représente une itération de marquage. Son principe est de marquer avec la valeur `marque` chaque case non marquée du calque
   — qui sont voisines d'une case du calque dont la valeur est `val` et
   — qui sont un couloir de la matrice `labyrinthe` (c'est-à-dire que la case correspondante dans `labyrinthe` ne vaut pas 1).

   Remarque : pour l'accessibilité `val` et `marque` valent toutes les deux 1 mais on verra dans la suite l'intérêt d'avoir deux paramètres.
2. `estAccessible(mat,pos1,pos2)` qui retourne vrai si `pos2` est accessible à partir de `pos1`.
3. En utilisant la fonction `estAccessible`, écrire une fonction `labyrintheValide(mat)` qui permet de savoir si un labyrinthe est valide (c'est-à-dire que l'on peut aller de la case 0,0 à la case la plus au sud-est)

Exercice 3 *Plus court chemin dans un labyrinthe*

Il s'agit maintenant de retrouver le plus court chemin entre deux points. Si il n'y a pas de chemin la fonction devra retourner la liste vide. L'algorithme va se dérouler en deux phases.
— La première consiste à utiliser une version modifiée de l'algorithme d'inondation. Dans cette nouvelle version on va marquer avec la valeur `i+1` les cases voisines d'une case marquée par la valeur `i`. De cette manière, la marque indique la plus petite distance entre la position de départ et la case marquée.
— La deuxième phase va consister à reconstituer un des plus courts chemins possibles. Pour cela il suffit de partir de la position d'arrivée est de remonter les cases de valeurs décroissantes jusqu'au point de départ.

Pour cet exercice vous devez écrire les fonctions

1. `estAccessible2(mat,pos1,pos2))` qui calcule le calque de la première phase. Cette fonction retourne le calque si un chemin existe entre les deux positions et `None` sinon.
2. `cheminDecroissant(calque,pos1,pos2)` qui effectue le calcul de la seconde phase de l'algorithme en supposant que le calque contient les information nécessaire à cette phase. La fonction retourne le chemin sous la forme d'une liste de postions.

3. `plusCourtChemin(mat,pos1,pos2)` qui retourne sous la forme d'une liste de couples, un des plus courts chemins entre `pos1` et `pos2` (si les deux cases ne sont pas connectées la fonction retourne la liste vide)