ISM MPSI/PCSI, Informatique
Année 2024/2025

TP Informatique 19

Copier le fichier `ClassePile.py` dans le répertoire de travail puis, pour chaque programme, commencer par réaliser l'importation `from ClassePile import *` pour manipuler la classe `ClassePile`. Celle-ci est munie des opérations primitives suivantes :

- `Pile()` qui crée une pile vide ;
- `P.vide()` qui renvoie `True` si la pile `P` est vide et `False` sinon ;
- `P.empiler(x)` qui empile l'élément `x` dans la pile `P` ;
- `P.depiler()` qui dépile le sommet de la pile non vide `P`.

On dispose également de l'opération `P.affiche()` pour afficher le contenu de la pile `P`.

Exercice 1

On modélise un labyrinthe par une liste de listes d'entiers. On fixe la convention suivante :

- les murs sont numérotés par des 2,
- les cases non visitées par des 0 ;
- et une case visitée par 1.

Une position est désignée par une liste `[i,j]` où i désigne le numéro de ligne et j le numéro de colonne. Dans l'exemple ci-contre saisi dans le fichier `laby.py`, le point en haut à gauche admet pour position `[0,0]`, en bas à droite `[6,6]` et la sortie `[3,6]`.

Les déplacements dans un labyrinthe sont vers le haut, vers le bas, vers la droite ou vers la gauche (pas de déplacement en diagonal).

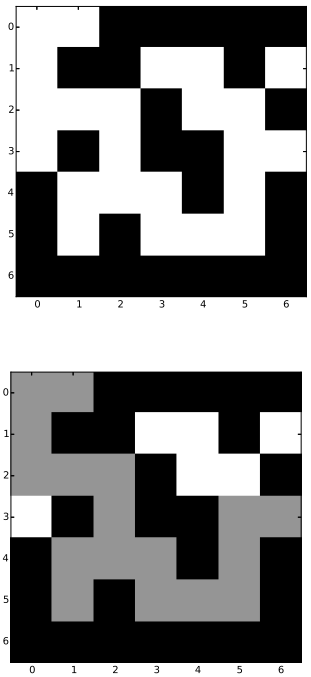

FIGURE 1 – Labyrinthe parcouru

1. Écrire des fonctions `h(pos)` pour haut, `b(pos)` pour bas, `g(pos)` pour gauche et `d(pos)` pour droite d'argument `pos` une position sous forme de liste `[i,j]` qui renvoient une nouvelle liste contenant la position après déplacement (on ne modifiera pas la liste fournie en argument). On supposera le déplacement possible (absence de mur dans la direction du déplacement).
2. Écrire une fonction `visite(laby,pos)` qui passe l'état de la case en position `pos` à 1, c'est-à-dire à l'état *visité*.
3. Écrire une fonction `etat(laby,pos)` qui renvoie l'état de la case du labyrinthe située à la position `pos`. La fonction devra renvoyer 2, c'est-à-dire un mur si `pos` repère une position hors du labyrinthe.

4. Écrire une fonction `voisinage(laby,pos)` qui renvoie `True` si la case en position `pos` admet une position voisine non visitée vers laquelle un déplacement est possible.
5. Écrire une fonction `voisin(laby,pos)` qui, pour une position admettant une position voisine libre non visitée, en renvoie une. Les priorités de déplacement sont laissées au choix du programmeur.
6. Écrire une fonction `dedale(laby,deb,fin)` d'arguments un labyrinthe `laby`, une position initiale `deb`, une position à atteindre `fin`. La fonction `dedale` modifie directement les cases de `laby` en les passant à l'état visité jusqu'à trouver un chemin menant à la sortie si celui-ci existe et sinon jusqu'à épuisement des cases accessibles. La fonction `dedale` renvoie un `True` et une pile contenant un chemin jusqu'à la sortie si celui-ci existe, `False` et une pile vide sinon.

L'algorithme est le suivant : la position courante est initialisée à la position de début et empilée dans un chemin ; puis, tant qu'on n'est pas sorti et que le chemin n'est pas vide, on marque la position courante comme *visitée* (il se peut qu'elle le soit déjà), on regarde s'il y a des cases voisines libres non visitées : si oui, on empile la case courante et on se déplace sur l'une de ces positions voisines, sinon, on dépile la dernière position empilée et on y retourne (on fait marche arrière sur une case déjà visitée donc). On sort de la boucle « tant que » soit parce qu'on est sorti du labyrinthe, soit parce que le chemin est vide ce qui signifie qu'on a visité toutes les cases accessibles sans atteindre la sortie.

Exercice 2

Dans le fichier `laby_rd.py`, la fonction `lab(N,M,p)` d'arguments `N` nombre de lignes, `M` nombre de colonnes et `p` la probabilité qu'une case soit un mur génère un labyrinthe de manière élémentaire. Mais ceci n'est pas pleinement satisfaisant car un tel labyrinthe peut être sans solution.

On se propose de générer aléatoirement un labyrinthe *parfait*, où chaque case libre est reliée à une autre par un unique chemin. Un tel labyrinthe est sans sans boucle, sans îlots, ... On part d'un environnement constitué uniquement de murs qu'on va creuser pour fabriquer le labyrinthe. Pour qu'il soit parfait, on ne creuse pas une case qui est voisine d'un case déjà creusée. On conserve les conventions fixées dans l'exercice précédent. On utilisera également les fonctions `etat` et `voisinage`.

1. Écrire une fonction `creuse(laby,pos)` qui passe l'état de la case en position `pos` à 0 et une fonction `mur(laby,pos)` qui passe son état à 2.
2. Écrire une fonction `avance(laby,pos)` qui renvoie les cases creusables depuis la position `pos`. Par convention, on ne creuse pas un mur qui débouche sur une case libre.
3. Écrire une fonction `gener(laby,deb)` d'arguments un labyrinthe `laby` constitué de murs et `deb` une position initiale et qui modifie directement les cases de `laby` pour générer un labyrinthe depuis la position `deb`.

L'algorithme est le suivant : on empile la position de départ puis, tant qu'on n'est pas revenu en position initiale, on détermine la liste des cases creusables depuis la position courante, on creuse la case où l'on se trouve, s'il n'y a pas de cases creusables, on dépile la dernière position ce qui permet de revenir sur ses pas et on mure la position de repli (pour ne pas fausser le caractère éventuellement creusable des positions voisines), sinon on en choisit une au hasard parmi celles creusables et on l'empile.

On utilisera la fonction `rd.randint(a,b)` qui permet de réaliser un tirage aléatoire d'un entier dans $[\![a\,;\, b-1]\!]$.