Les piles

L'essentiel du cours

1 Présentation

Une pile est une structure de données qui prend la forme d'une liste en PYTHON. Il faut voir une pile comme une pile d'assiettes. Chaque assiette est un élément de la pile.
On peut poser ou enlever des assiettes de la pile, mais c'est toujours la dernière assiette posée qui est susceptible d'être enlevée, contrairement aux files (analogie aux file d'attente).
Une pile n'est donc modifiable que de deux façons différentes :

- soit on ajoute un élément en haut de la pile : c'est l'opération d'empilement
- soit on supprime un élément toujours du haut de la pile : c'est l'opération de désempilement.

En PYTHON, il y a trois fonctions (appelées primitives) qui sont systématiquement fondamentales dans tout problème gérant les piles :

- l'empilement d'un objet x dans la pile P : `P.append(x)`
- le désempilement de la pile P : `P.pop()` ; cette syntaxe fait deux choses : premièrement elle modifie la pile P et deuxièmement, elle met en mémoire éventuellement l'élément désempilé. La syntaxe `element=P.pop()` fait ces deux choses et l'élément désempilé de la pile `P` est tout de même mis en mémoire dans `element`
- le test de la pile vide `P==[]` ; cette condition permet de savoir si le problème posé a un non une solution.

2 Liens avec la recherche en profondeur dans un graphe

Un graphe est la donnée $(\mathscr{S}, \mathscr{A})$ d'un ensemble $\mathscr{S}$ de sommets et d'un ensemble $\mathscr{A}$ d'arêtes. Ces arêtes peuvent être orientées auquel cas les éléments de $\mathscr{A}$ seront des couples (a, b) pour modéliser l'arête partant de a pour aller à b ou bien ces arêtes peuvent ne pas être orientées auquel cas les éléments de $\mathscr{A}$ seront des paires $\{a, b\}$ pour modéliser l'arête d'extrémités a et b.
Le principe d'une recherche en profondeur dans un graphe orienté $(\mathscr{S}, \mathscr{A})$ par exemple est le suivant :
étant donnés un sommet de départ D et un sommet d'arrivée A, est-il possible de cheminer de D vers A le long d'arêtes du graphe ? Plus formellement, existe-t-il $r \in \mathbb{N}$ et $(S_0, \cdots, S_r)$ tel que :

$$\forall k \in [\![0, r-1]\!],\ (S_k, S_{k+1}) \in \mathscr{A} \text{ et } \begin{cases} S_0 = D \\ S_r = A \end{cases}.$$

Il existe plusieurs algorithmes pour répondre à ce problème, mais le plus intuitif est celui de la recherche en profondeur.

- on part du sommet de départ D : la pile P est initialisée à $[D]$
- à chaque étape, si l'on est à un sommet S, on note $\mathscr{V}$ l'ensemble des sommets directement accessibles à partir de S – autrement dit $\mathscr{V} = \left\{R \in \mathscr{S} \mid (S,R) \in \mathscr{A}\right\}$ mais on ne regarde que l'ensemble $\mathscr{W}$ des éléments de $\mathscr{V}$ qui n'ont pas encore été visités.
- si $\mathscr{W}$ est non vide, on pioche aléatoirement par exemple un sommet S' dans $\mathscr{W}$ et on procède à l'empilement du sommet S' dans la pile P : on vient d'avancer d'un pas
- si $\mathscr{W}$ est vide, il n'y a aucun nouveau sommet accessible : on est dans une impasse. On procède au désempilement de la pile P : on recule d'un pas. Le sommet désempilé ne fait plus partie des sommets accessibles.
- Dans le processus, soit on trouve un sommet qui vaut A et on vient de résoudre le problème ; le chemin trouvé n'est pas forcément le chemin optimal en terme de longueur de trajet, soit la pile devient vide à un moment donné et dans ce cas le problème n'a pas de solution.

Exercice 1

On souhaite programmer de plusieurs manières différentes une fonction qui prend en argument une liste et restitue la liste où les éléments ont été inversés. Par exemple, la liste $[3,2,3,4]$ renvoie à la liste $[4,3,2,3]$.

1. Rédiger une telle fonction récursivement.
2. Rédiger une telle fonction en utilisant une pile.

Exercice 2

Un labyrinthe est dit parfait s'il existe un seul chemin (sans boucle) reliant deux points du labyrinthe.
On se donne deux entiers strictement positifs p et q et on considère des labyrinthe de format $p \times q$ cases.
Informatiquement, un labyrinthe est un tableau M de type array où chaque valeur du tableau vaut 0 ou 1, avec la convention suivante :

$$M[i,j] = 0 \iff \text{ la case } [i,j] \text{ est un obstacle.}$$

Tous les labyrinthes seront bordés de murs à leur périphérie.
On va utiliser un tableau de 0/1 est_visite tel qu'à chaque instant, on sait si la case $[i,j]$ a déjà été visitée une ou plusieurs fois :

$$\text{est\_visite}[i,j] = 0 \iff \text{ la case } [i,j] \text{ n'a jamais été visitée.}$$

Dès qu'une case sera empruntée dans le labyrinthe, la valeur de est_visite[i,j] sera mise égale à 1 et dès que l'on empruntera une case, toute case $[k,\ell]$ qui lui est voisine verra la valeur de est_visite$[k,\ell]$ augmenter de $\frac{1}{2}$.
Une case [i,j] où est_visite[i,j]$\leqslant \frac{1}{2}$ reste donc encore libre d'accès. Une case [i,j] où est_visite[i,j]$> \frac{1}{2}$ constitue un obstacle autrement dit un mur dans le labyrinthe.

1. Écrire une fonction voisins(case,est_visite)) qui renvoie la liste des cases voisines et qui ne sont pas des obstacles.
2. Écrire une fonction choisir(L) qui prend en entrée une liste L et renvoie un élément aléatoire de cette liste L, lorsque celle-ci est non vide.

On souhaite maintenant écrire une fonction labyrinthe(p,q) renvoyant un labyrinthe parfait de format $p \times q$. Le principe est le suivant :

- on initialise le tableau est_visite à un tableau rempli de 0, sauf sur les bords où la valeur vaut 1.
- on crée une pile pile contenant les cases au fur et à mesure de la progression dans le labyrinthe. Au départ, elle contiendra par exemple la première case $[1, 1]$
- on crée une liste chemin qui comptabilise toutes les cases qui ont transité à un moment donné dans la pile pile
- puis, jusqu'à épuisement de la pile, en notant case l'élément au sommet de la pile, s'il n'y a plus de voisins disponibles, on dépile le sommet appelé case. Sinon, on choisit aléatoirement l'un des voisins disponibles appelé suivant, on met à jour le tableau est_visite pour les cases qui lui sont adjacentes (uniquement lorsque case provient d'un empilement et non si case est reconsidérée suite à un désempilement de la pile pile, on empile suivant dans la pile pile et on met à jour la liste chemin.
- on renvoie la liste chemin des cases finalement valides pour notre labyrinthe.

3. Implémenter cet algorithme.
4. Une fois un labyrinthe parfait construit, implémenter le calcul du chemin permettant d'accéder d'une case de départ D à une case d'arrivée A.
5. Prendre $p = 30$ et $q = 50$ par exemple. En utilisant le module pylab et une fonction d'affichage de type imshow, afficher le labyrinthe parfait construit. Prendre dans les cases libres aléatoirement la case D de départ dans le quadran $\left[\!\left[\frac{2p}{3}, p\right]\!\right] \times \left[\!\left[0, \frac{q}{3}\right]\!\right]$, la case d'arrivée une case libre aléatoire dans le quadran $\left[\!\left[0, \frac{p}{3}\right]\!\right] \times \left[\!\left[\frac{2q}{3}, q-1\right]\!\right]$.

   Faire une figure où apparaissent les éléments suivants :
   - tous les obstacles du labyrinthe sont de couleur grise $\left(\frac{2}{3}, \frac{2}{3}, \frac{2}{3}\right)$
   - toutes les cases libres du labyrinthe sont de couleur claire $(0.99, 0.95, 0.9)$
   - la case de départ D est de couleur magenta $(1, 0, 1)$
   - la case d'arrivée A est de couleur rouge $(1, 0, 0)$
   - toutes les cases du seul chemin reliant D à A (sauf ces deux extrémités) sont de couleur jaune $(1, 1, 0)$.
   -

   ainsi que le chemin reliant les points D et A. On adoptera les codes couleurs suivant : toute case libre est blanche. Tout obstacle est gris. Le chemin est affiché en cyan. La case de départ est verte et la case d'arrivée est rouge.

Voici des exemples d'images possibles :