Trouver son chemin dans un labyrinthe

Un exercice sur le parcours de graphe

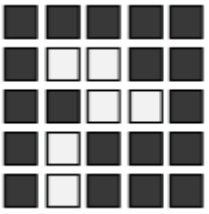
On considère un labyrinthe sous forme d'une grille carrée. Chacune des cases peut être libre ou bien bloquée.

Plus précisément, la grille est modélisée par une liste de listes.

Chacune des sous-listes correspond à une ligne de la grille.

Chacun des éléments d'une ligne sera l'entier 0 pour une case libre ou bien l'entier 1 pour une case occupée.

Par exemple le labyrinthe



est représenté par la grille :

```
grille_exemple = [
[1,1,1,1,1],
[1,0,0,1,1],
[1,1,0,0,1],
[1,0,1,1,1],
[1,0,1,1,1]
]
```

La première case tout en haut à gauche est la case $(0,0)$

Sur l'exemple, la case libre tout à droite est la case $(2,3)$

1. Écrire une fonction `cases_adjacentes(grille: list, case: tuple)-> list` qui retourne la liste des cases adjacentes **libres** à `case`.
   Sur l'exemple `cases_adjacentes(grille, (2,2))` retournerait `[(1,2), (2,3)]`
   (la case $(3,1)$ n'est pas adjacente par un côté)
   On rappelle l'instruction `(a,b) = case` pour récupérer les coordonnées d'un tuple.
2. Une case (i,j) est *atteignable* à partir de la case (a,b) s'il existe une suite de cases adjacentes allant de (a,b) à (i,j)
   Écrire une fonction `cases_atteignables(grille: list, case_depart: tuple)-> list` qui retourne la liste des cases atteignables à partir de la case `case_depart`.
   ↪ *On remarquera que cela revient à faire un parcours du graphe des cases libres, la fonction précédente donnant les "voisins/cases adjacentes" de chaque "sommet/case".*
   Pour marquer les cases "cochées" une liste n'est pas adaptée ici : on pourra utiliser un dictionnaire de clés les tuples des cases affectées de la valeur `True`.
   Par exemple `{(1,1):True , (1,3):True}` *signifie que les cases* $(1,1)$ *et* $(1,3)$ *ont déjà été cochées.*
3. UN PEU PLUS COMPLIQUÉ.
   On souhaiterait aussi obtenir le chemin à suivre pour atteindre effectivement une case donnée
   - Améliorez votre fonction pour qu'elle enregistre (dans un dictionnaire par exemple), pour chaque case, la case mère d'où elle a été "vue" dans le parcours.
   - En déduire une fonction `chemin(grille, case_depart, case_arrivee)` qui retourne la liste des cases à emprunter pour aller de `case_depart` à `case_arrivee`.
   - On pourra discuter de l'influence d'un parcours en largeur ou en profondeur ici.
4. POUR ENCORE PLUS DE CHALLENGE
   Question ouverte, vous serez amené à réécrire plusieurs fonctions
   Utilisez le principe de l'algorithme $A^\star$ pour que la fonction `chemin` explore en premier les cases en "direction" de la case d'arrivée.

   Ce genre d'algorithme constitue les prémices des algorithmes de *Pathfinding* utilisé dans les jeux vidéos.
   A l'exception qu'il faudrait aussi actualiser le parcours en temps réel en fonction des modifications de la case d'arrivée (qui suit les déplacements du joueur) et tout optimiser pour accélérer le calcul.