TP 9 INFO Graphes

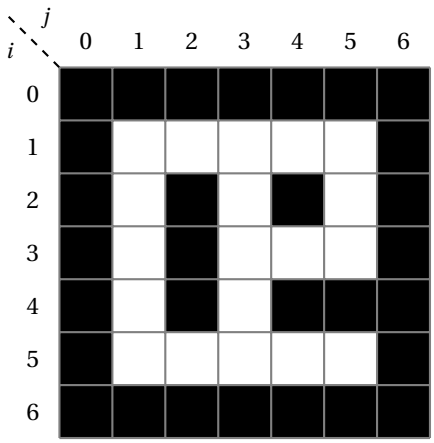
Dans ce TP, nous allons mettre en œuvre les algorithmes étudiés en cours sur les graphes.

Pour commencer, récupérez l'ensemble des fichiers liés à ce TP dans la rubrique informatique du site se trouvant à l'adresse `http://vonbuhren.free.fr`, puis placez ces derniers au même emplacement que votre fichier Python.

Partie I Généralités sur les labyrinthes

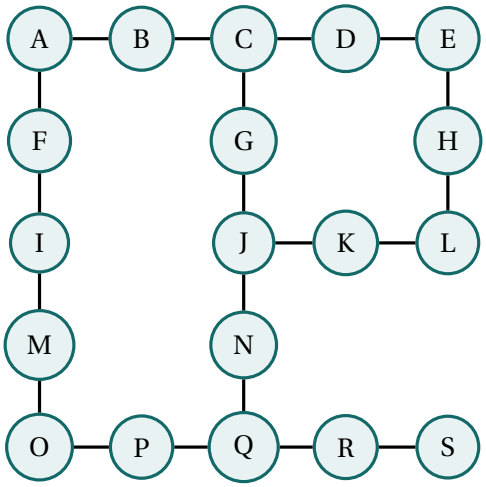
I.A - Implémentation en Python

Dans ce TP, on considère des labyrinthes rectangulaires composés de cases blanches et de cases noires comme celui ci-dessous.



Les cases noires représentent des murs, tandis que les cases blanches sont celles où un personnage peut se positionner. Dans ce TP, on suppose que le personnage ne peut que se déplacer horizontalement ou verticalement entre deux cases blanches adjacentes. Pour éviter des complications liées aux bords, on ne considérera que des labyrinthes entièrement entourés de murs comme dans l'exemple ci-dessus.

Ces labyrinthes peuvent se représenter comme un graphe non orienté dont les sommets correspondent aux cases blanches et dont les arêtes sont les déplacements possibles pour le personnage entre deux cases blanches voisines.



En Python, le labyrinthe sera sauvegardé par un tuple de chaînes de caractères. Chaque élément du tuple représente une ligne du labyrinthe où une case noire est symbolisée par le caractère `'#'` et une case blanche par le caractère `'.'` . Par exemple, le labyrinthe précédent est mémorisé avec le tuple `laby` suivant.

```
laby = (
'#######',
'#.....#',
'#.#.#.#',
'#.#...#',
'#.#.###',
'#.....#',
'#######')
```

En particulier, pour accéder au symbole indiquant la couleur de la case d'indices (i, j), il suffit d'utiliser l'expression `laby[i][j]`. De plus, le nombre de lignes et le nombre de colonnes du labyrinthe sont donnés par les deux expressions suivantes.

```
nb_lignes = len(laby)
nb_colonnes = len(laby[0])
```

Dans la suite, les sommets du graphe associé au labyrinthe `laby` seront identifiés avec leurs indices (i, j).

I.B - Chargement et affichage d'un labyrinthe

Nous allons utiliser un module écrit spécifiquement pour ce TP afin de pouvoir charger facilement différents labyrinthes et de pouvoir les afficher.

```
from module_laby import *
```

Les dix fichiers `laby0.txt`, ..., `laby9.txt` contiennent chacun un labyrinthe. Il suffira d'utiliser l'instruction suivante pour récupérer un labyrinthe directement depuis un des fichiers textes.

```
# Chargement du labyrinthe à partir du fichier texte : numéro à choisir entre 0 et 9
laby = charger_laby(0)
```

Vous pourrez afficher le labyrinthe chargé avec l'instruction suivante.

```
# Affichage simple du labyrinthe
afficher(laby)
```

Si vous disposez d'un dictionnaire `marquages` dont les clés sont des sommets du graphe associé au labyrinthe `laby`, vous pouvez afficher les valeurs du dictionnaire sur le labyrinthe avec l'option `infos` comme ci-dessous.

```
# Affichage du labyrinthe avec des informations supplémentaires
afficher(laby, infos = marquages)
```

Si vous souhaitez tracer un chemin sur le labyrinthe, vous pouvez utiliser l'option `chemin` comme ci-dessous.

```
# Affichage du labyrinthe avec un chemin tracé
afficher(laby, chemin = [(1, 1), (1, 2), (1, 3)])
```

Partie II Préliminaires

Question 1 : Écrire une fonction `nb_cases_blanches(laby) -> int` qui renvoie le nombre de cases blanches dans le labyrinthe `laby`.

Pour parcourir les déplacements possibles pour le personnage, vous pourrez utiliser la syntaxe ci-dessous.

```
for (di, dj) in [(1,0), (-1,0), (0,1), (0,-1)]:
        # Corps de la boucle
```

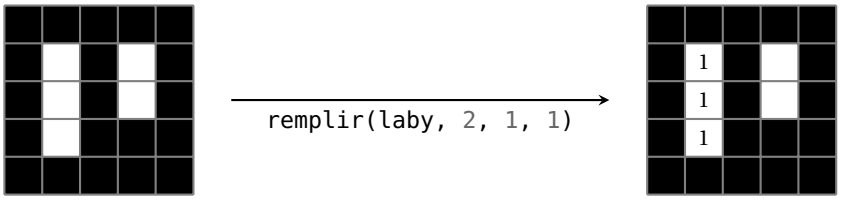
Question 2 : Écrire une fonction `generer_dico(laby) -> dict` qui renvoie le dictionnaire dont les clés sont les sommets du graphe associé à `laby` et les valeurs sont les listes d'adjacence de ce graphe.

Question 3 : En utilisant votre fonction précédente, écrire une fonction `nb_aretes(laby) -> int` qui renvoie le nombre d'arêtes dans le graphe non orienté associé au labyrinthe `laby`.

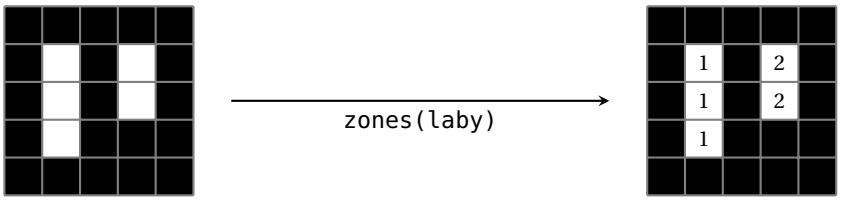
Partie III Parcours en profondeur

Dans cette partie, on pourra utiliser la structure de pile étudiée en cours.

Question 4 : Écrire une fonction `remplir(laby, i_d, j_d, x) -> dict` qui renvoie un dictionnaire dont les clés sont les sommets accessibles via un chemin depuis le sommet `(i_d, j_d)` et qui associe à chacun de ces sommets la valeur `x`. Voici ci-dessous un exemple d'exécution de la fonction `remplir`.



Question 5 : Écrire une fonction `zones(laby) -> dict` qui renvoie un dictionnaire dont les clés sont les sommets du graphe associé à `laby` et de sorte que les valeurs respectives associées à deux sommets sont identiques si et seulement si il existe un chemin entre ces deux sommets. Voici ci-dessous un exemple d'exécution de la fonction `zones`.



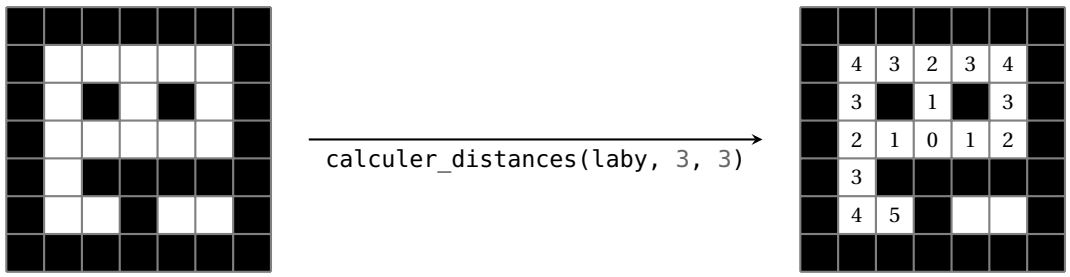
On dit qu'un labyrinthe est parfait si pour toute paire de cases blanches distinctes, il existe un unique chemin reliant ces deux cases.

Question 6 : Écrire une fonction `est_parfait(laby) -> bool` qui renvoie un booléen indiquant si le labyrinthe `laby` est parfait.

Partie IV Parcours en largeur

Dans cette partie, on étudie le problème du plus court chemin dans un labyrinthe. Comme la distance entre deux cases blanches adjacentes est constante dans le labyrinthe, il est inutile d'utiliser l'algorithme de Dijkstra, il suffit d'effectuer un parcours en largeur de ce dernier. On pourra utiliser la structure de file étudiée en cours.

Question 7 : Écrire une fonction `calculer_distances(laby, i_d, j_d) -> dict` qui renvoie un dictionnaire dont les clés sont les sommets du graphe associé à `laby` et dont la valeur associée à chaque sommet `(i, j)` est la distance de la case d'indices `(i_d, j_d)` à la case d'indices `(i, j)`. Voici ci-dessous un exemple d'exécution de la fonction `calculer_distances`.



Question 8 : Écrire une fonction `distance(laby, i_d, j_d, i_a, j_a) -> int` qui renvoie la distance de la case blanche d'indices `(i_d, j_d)` à la case blanche d'indices `(i_a, j_a)` du labyrinthe `laby`.

Question 9 : Écrire une fonction `chemin(laby, i_d, j_d, i_a, j_a) -> list` qui renvoie une liste contenant les sommets d'un plus court chemin allant de la case blanche d'indices `(i_d, j_d)` à celle d'indices `(i_a, j_a)` du labyrinthe `laby`. Voici un exemple d'exécution de la fonction avec le labyrinthe ci-dessus.

```
>>> lst = chemin(laby, 3, 3, 5, 2)
>>> lst
[(3, 3), (3, 2), (3, 1), (4, 1), (5, 1), (5, 2)]
```