ITC – TD n°12-4

Parcours de labyrinthe

I. Généralités

On utilise les graphes pour représenter des labyrinthes sur des damiers : chaque sommet correspond à une case autorisée, les cases n'ayant pas de sommet associé étant des murs. Lorsque deux cases sont adjacentes, elles sont reliées par une arête.

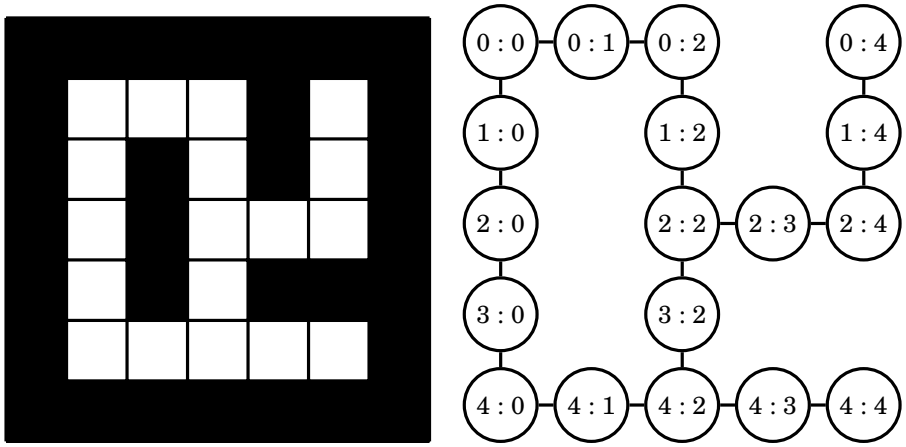

Figure 1 : Exemple de labyrinthe et de graphe associé

On pourra récupérer et afficher des graphes-labyrinthes déjà préparés dans le module habituel ; les sommets sont nommés selon les coordonnées sur le damier et la configuration d'affichage est adaptée :

```
import module_graphes as gr
lab1 = gr.recuperer_graphe('l1')
gr.afficher_graphe(lab1)
```

On importera également des modules pour des structures de pile et de file :

```
from queue import Queue, LifoQueue
```

Pour rappel :

- on peut créer une nouvelle pile (respectivement file) et lui attacher une référence à l'aide de `pile = LifoQueue()` (respectivement `file = Queue()`) ;
- on peut tester si une pile/file est vide avec `pile.empty()` ;
- on peut ajouter un élément `x` à une pile/file avec `pile.put(x)` ;
- on peut enlever le dernier (respectivement premier) élément d'une pile (respectivement file) avec `x = pile.get()` .

Le but de ce TD est, étant donné un sommet de départ et un sommet d'arrivée, de trouver une façon de parcourir le labyrinthe (le graphe, donc) qui permette d'aller de l'un à l'autre, si un tel chemin existe.

En plus du graphe lui-même, nous allons manipuler deux dictionnaires donnant des informations sur les sommets :

- `visité` est un dictionnaire `str, bool` tel que `visité[u]` soit `True` si le sommet u a déjà été visité au cours du parcours, et `False` sinon ;
- `précédent` est un dictionnaire `str, str` tel que `précédent[u]` soit la chaîne vide `''` si le sommet u n'a jamais été visité, et v si u a déjà été visité au cours du parcours depuis le sommet v, à travers l'arête (v, u).

1 – Étant donné le dictionnaire `précédent` , on peut remonter de proche en proche depuis un sommet d'arrivée jusqu'au point de départ. Écrire une fonction `extraire_chemin` qui prend en paramètres le graphe-labyrinthe, le dictionnaire `précédent` et le sommet de sortie, et renvoie un graphe contenant tous les sommets du labyrinthe mais uniquement les arêtes du chemin.

II. Backtracking : parcours en profondeur

Nous allons maintenant parcourir le graphe en visitant chaque sommet : cela correspond à l'algorithme suivant, dans lequel on possède les dictionnaires `précédent` et `visité` , ainsi qu'une pile contenant des sommets :

- prendre le sommet du haut de la pile, notons-le u ;
- parcourir tous ses sommets adjacents v : s'ils ne sont pas encore visités, on les place sur la pile et on note `précédent[v] = u` ;
- marquer u comme visité.

En visitant ainsi chaque sommet de la pile depuis le départ, on peut atteindre tout sommet de la composante connexe : on parle de **parcours en profondeur**.

2 – Éxécuter à la main cet algorithme pour le graphe de la figure 1, en partant du sommet $(0, 0)$ et pour arriver au sommet $(4, 0)$; on supposera que lorsqu'on visite un sommet, on récupère ses sommets adjacents dans l'ordre suivant : droite, bas, gauche, haut. Expliquer pourquoi on peut parler de backtracking.

3 – Écrire une fonction `visiter_sommet` qui prend en paramètre un graphe-labyrinthe, les dictionnaires `visité` et `précédent` , ainsi qu'une pile de sommets en cours de traitement, qui récupère le sommet u au-dessus de la pile et le visite. La fonction renvoie ensuite u.

4 – Écrire une fonction `trouver_chemin` qui prend en paramètre un graphe-labyrinthe et deux sommets `entrée` et `sortie` , et exécute le parcours en profondeur à partir du sommet `entrée` . Si on visite le sommet `sortie` , on renvoie un graphe du chemin avec la fonction `extraire_chemin` ; si non, on renvoie un graphe contenant tous les sommets du labyrinthe mais aucune arête.

5 – Tester le parcours avec les labyrinthes fournis :

```
lab1, lab2 = gr.recuperer_graphe('l1'), gr.recuperer_graphe('l2')
entrée, sortie = '0:0', '0:9'

g_ch1 = trouver_chemin(lab1, entrée, sortie)
gr.afficher_graphes_multiples([lab1, g_ch1])

g_ch2 = trouver_chemin(lab2, entrée, sortie)
gr.afficher_graphes_multiples([lab2, g_ch2])
```

6 – Optionnel : reprendre ce parcours avec une version récursive de la fonction `visiter_sommet` .

III. Plus court chemin : parcours en largeur

On se propose à présent d'appliquer le même algorithme en remplaçant la pile par une file.

7 – Éxécuter à la main cet algorithme pour le graphe de la figure 1, en partant du sommet $(0, 0)$ et pour arriver au sommet $(4, 0)$; on supposera que lorsqu'on visite un sommet, on récupère ses sommets adjacents dans l'ordre suivant : droite, bas, gauche, haut. Quel avantage a-t-on par rapport au parcours en profondeur ?

8 – Écrire une fonction `trouver_chemin_plus_court` qui prend en paramètre un graphe-labyrinthe et deux sommets `entrée` et `sortie` , et exécute le parcours en largeur à partir du sommet `entrée` . Si on visite le sommet `sortie` , on renvoie un graphe du chemin avec la fonction `extraire_chemin` ; si non, on renvoie un graphe contenant tous les sommets du labyrinthe mais aucune arête.

9 – Tester les parcours avec les labyrinthes fournis :

```
lab1, lab2 = gr.recuperer_graphe('l1'), gr.recuperer_graphe('l2')
entrée, sortie = '0:0', '0:9'

g_p1 = trouver_chemin(lab1, entrée, sortie)
g_l1 = trouver_chemin_plus_court(lab1, entrée, sortie)
gr.afficher_graphes_multiples([lab1, g_p1, g_l1])

g_p2 = trouver_chemin(lab2, entrée, sortie)
g_l2 = trouver_chemin_plus_court(lab2, entrée, sortie)
gr.afficher_graphes_multiples([lab2, g_p2, g_l2])
```