Labyrinthes et structures de données

INFORMATIQUE COMMUNE - TP n° 3.2 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ utiliser les listes imbriquées,
- ☞ utiliser des dictionnaires,
- ☞ utiliser un tableau bidimensionnel sous la forme de liste imbriquées,
- ☞ représenter graphiquement un labyrinthe,
- ☞ générer un labyrinthe de deux manières différentes.
- ☞ explorer un labyrinthe.

Ce TP est construit dans l'esprit d'une épreuve de concours et de manière progressive. Il permet de réviser de nombreux éléments du programme de première année : listes, tableaux, importation de fonctions de bibliothèques, programmation défensive (assertion et tests), passage de paramètres, muabilité, complexité, récursivité, tracer de graphiques, parcours d'une structure de données. Il permet également de découvrir les piles et les dictionnaires.

A Contexte

Le contexte de ce TP est l'exploration de labyrinthes. On considère des labyrinthes parfaits, c'est à dire dont deux cases quelconques peuvent toujours être reliées par un unique chemin. Par convention, ces labyrinthes sont des pièges et on ne peut pas en sortir : leur périphérie est constituée de murs (cf. figure 1). On choisit arbitrairement que la case de départ est en bas à gauche et l'arrivée en haut à droite (cf. figure 2).

On cherche à :

1. modéliser un tel labyrinthe,
2. afficher à l'écran un labyrinthe,
3. générer des labyrinthes,
4. explorer un labyrinthe, c'est à dire à en sortir **sans savoir où se trouve la sortie**.

B Modéliser et implémenter un labyrinthe

On choisit de modéliser un labyrinthe sous la forme d'un tableau bidimensionnel de taille fixe. La constante `N_COLS` désigne le nombre de colonnes et `N_ROWS` le nombre de lignes de ce tableau. Ce tableau bidimensionnel est **implémenté** par une liste imbriquée en Python, c'est à dire une liste de listes. Par exemple, `maze[0][2]` renvoie la cellule de la première ligne et troisième colonne du labyrinthe.

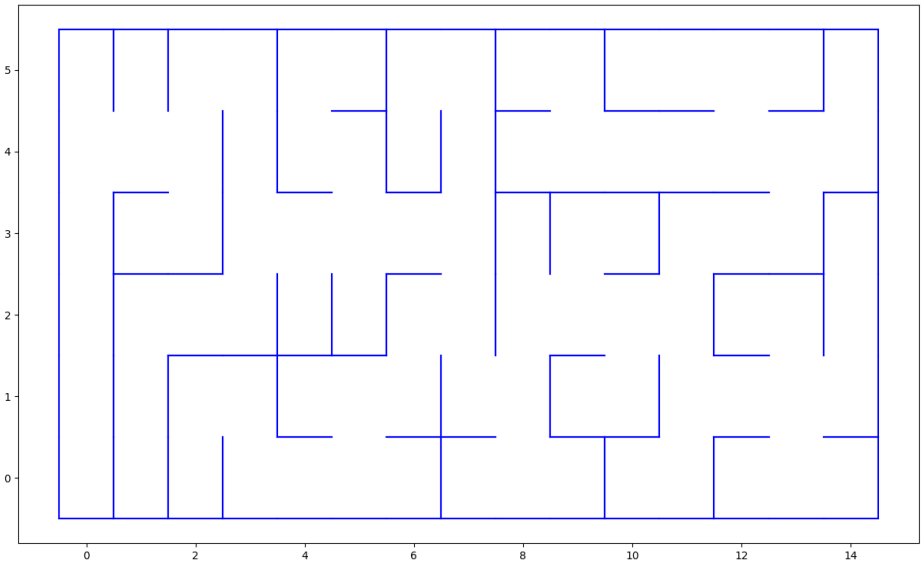

FIGURE 1 – Labyrinthe parfait rectangulaire

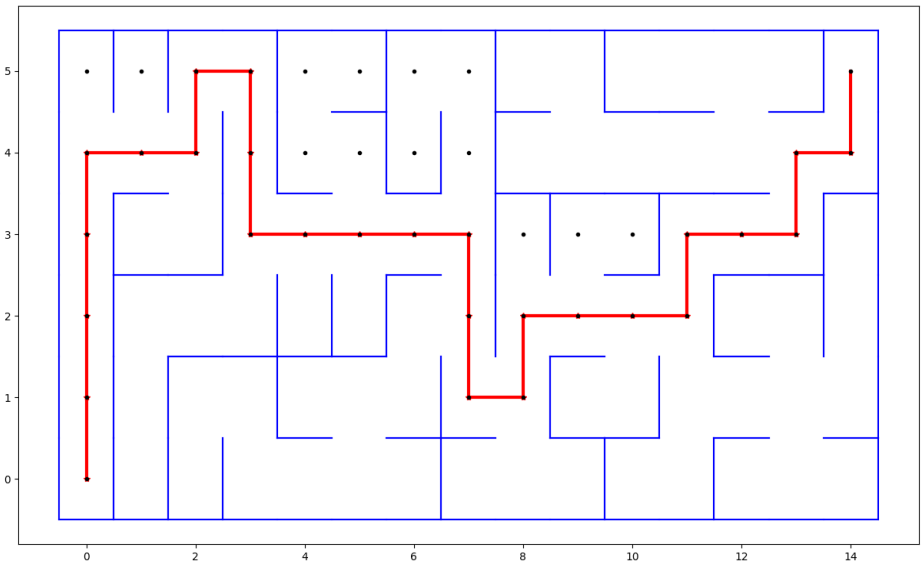

FIGURE 2 – Exploration d'un labyrinthe parfait rectangulaire. Le départ est effectué en bas à gauche et l'arrivée est en haut à droite.

Chaque case du tableau modélise une cellule du labyrinthe. Une cellule possède quatre points cardinaux North, East, South, West que l'on notera `"N"`, `"E"`, `"S"`, `"W"` en Python. Selon ces directions, on peut construire ou abattre un mur autour de la cellule. On choisit d'**implémenter** une cellule par un dictionnaire dont les clefs sont les quatre points cardinaux et les valeurs un booléen. **La convention adoptée est qu'un mur existe si ce booléen vaut True.**

B1. En considérant le labyrinthe de la figure 1, créer un dictionnaire Python qui modélise les cellules d'indice (ligne, colonne) :
   (a) (0,0),
   (b) (1,4),
   (c) (4,3),
   (d) (14,3)

B2. Écrire une fonction de prototype `create_closed_cell()` qui renvoie une cellule complètement murée, c'est à dire qui possède des murs dans toutes les directions.

B3. Écrire une fonction de prototype `is_closed(cell)` où le paramètre `cell` est un dictionnaire représentant une cellule du labyrinthe. Cette fonction renvoie un booléen qui atteste que la cellule possède un mur dans chaque direction, c'est à dire qu'elle est complètement close.

B4. Écrire une fonction de prototype `create_all_closed_maze(n_rows, n_cols)` qui renvoie un labyrinthe de taille `n_rows`×`n_cols` et dont les cellules sont complètement murées. Le tester en utilisant une assertion et la fonction précédente, c'est à dire vérifier que toutes les cellules sont closes.

B5. Écrire une fonction qui permette d'abattre un mur d'une cellule selon une certaine direction. Son prototype est `tear_down(maze,i,j,direction)`, où `maze` est le labyrinthe, `i` et `j` les indices de ligne de colonne de la cellule considérée et `direction` la direction `"N"`, `"E"`, `"S"` ou `"W"` selon laquelle on souhaite abattre le mur [1]. Cette fonction ne renvoie rien et il faut être capable d'expliquer pourquoi ;-) Tester cette fonction avec des assertions et la fonction `is_closed`. Cette fonction sera utile à la section E.

B6. Créer le tableau bidimensionnel de booléens nommé `visited` de la même dimension que le labyrinthe sous la forme d'une liste imbriquée. Lorqu'on explore le labyrinthe, ce tableau représente le fait d'être passé sur une cellule : si le booléen vaut True, c'est qu'on est déjà passé sur la cellule. On initialisera à `False` toutes les cases du tableau. Pour savoir si on est passé sur la cellule (3,4) , on testera la valeur de `visited[3][4]`.

B7. Écrire une fonction de prototype `dir_unvisited_neighbours(visited, i, j)` qui renvoie les directions des cellules voisines de la case (i,j) qui n'ont pas été visitées sous la forme d'une liste, par exemple `["N", "E"]`. Pour bien tester cette fonction, certaines cases de `visited` doivent être à `True` . Cette fonction sera utile à la section E.

C Afficher un labyrinthe

On suppose pour l'instant qu'on a réussi à générer un labyrinthe, à l'explorer et à récupérer toutes ces informations. Pour répondre aux questions qui suivent, il est nécessaire de télécharger le fichier Pyhon maze_example.py. Il contient le labyrinthe de l'exemple dessiné sur la figure 1, le chemin d'exploration trouvé et les cellules visitées. L'objectif est de programmer les fonctions nécessaires pour afficher le labyrinthe et les résultats comme sur la figure 2. Les constantes du labyrinthe sont pour cet exemple :

1. Il faut remarquer qu'un mur est constitué de deux cloisons…

```
N_ROWS = 6  # Maze's width
N_COLS = 15  # Maze's height
```

Les fonctions à programmer dans cette section ne renvoient rien, elles tracent seulement. Elles s'appuient sur la bibliothèque **matplotlib** et la **convention suivante** :

- les coordonnées (i,j) de la cellule dans le tableau correspondent au centre de la case telle qu'elle est dessinée sur la figure,
- les points d'exploration sont matérialisés au centre de la case,
- les chemins passent par le centre de cette case,
- les murs se situent à égale distance du centre de la case et ne débordent pas sur l'autre case,
- les cloisons d'un mur entre deux cellules sont superposées.

C1. À l'aide d'une directive d'importation de bibliothèque, importer tous les éléments du fichier téléchargé.

C2. Vérifier que la valeur des constantes `N_ROWS` et `N_COLS` définies au début de votre programme sont cohérentes avec les dimensions du labyrinthe `maze_ex` donné en exemple.

C3. À l'aide d'une directive d'importation de bibliothèque, importer le module `pyplot` de la bibliothèque `matplotlib` sous le nom `plt`.

C4. Écrire une fonction de prototype `draw_visited(maze, visited)` qui marque les cellules visitées lors de l'exploration **d'un point noir au centre de la cellule**. Dans le programme principal, effectuer un `plt.show()` pour visualiser le résultat.

C5. Écrire une fonction de prototype `draw_path(maze, path)` qui trace le chemin `path` passé en paramètre sur le labyrinthe `maze`. Ce chemin a l'apparence d'**une ligne rouge**. Les puristes ajouteront une étoile rouge au centre de la case traversée.

C6. Écrire une fonction de prototype `draw_maze(maze)` qui trace le labyrinthe passé en paramètre. Les murs sont bleus.

D Explorer et sortir d'un labyrinthe

On souhaite maintenant pouvoir trouver la sortie du labyrinthe. A priori, on doit pouvoir définir n'importe quelle cellule comme cellule de sortie. Au niveau de l'implémentation, pour distinguer cette cellule des autres, on la matérialise **en ajoutant à son dictionnaire une clef** `"EXIT"` dont la valeur est `True`.

On dispose de l'algorithme 1 qui permet de sortir d'un labyrinthe en l'explorant récursivement[2].

D1. À quoi sert la ligne 4 de l'algorithme 1 ? Comment pouvez-vous la traduire en Python ?

D2. Au cours de l'algorithme 1, passe-t-on deux fois sur la même case ?

D3. L'algorithme 1 parcourt-il toutes les cases ?

D4. Écrire une fonction Python qui implémente l'algorithme 1. Vérifier que le chemin obtenu correspond avec celui donné à la section précédente.

D5. Pourrait-on définir plusieurs sorties dans le labyrinthe ?

2. Cette technique est un algorithme classique d'exploration nommé retour sur trace ou backtracking.

Algorithme 1 Trouver la sortie d'un labyrinthe en explorant récursivement

```
1 : Fonction EXPLORER_SORTIR(maze, i, j, visited, path)      ▷ au premier appel, path est une liste vide.
2 :     visited[i][j] ← Vrai                                 ▷ au premier appel, visited est totalement à Faux.
3 :     Ajouter (i,j) à path
4 :     si maze[i][j] est la sortie alors                                                          ▷
5 :         renvoyer Vrai
6 :     sinon
7 :         pour chaque direction "N", "S","E", "W" répéter      ▷ On ne traduira pas par une boucle.
8 :             si un case voisine non visitée existe dans cette direction alors
9 :                 p,q ← les coordonnées de ce voisin non visité dans cette direction
10 :                si EXPLORER_SORTIR(maze, p, q, visited, path) alors
11 :                    renvoyer Vrai                                          ▷ On a trouvé la sortie!
12 :                sinon
13 :                    Retirer (p,q) du path          ▷ C'est la dernière case visitée et ajoutée qu'on retire.
14 :    renvoyer Faux                                                          ▷ Pas de sortie par ici...
```

E Générer un labyrinthe par exploration exhaustive

Les labyrinthes parfaits, c'est à dire qui relient deux cellules quelconques par un unique chemin, peuvent être générés par exploration exhaustive de toutes les cellules. La situation de départ est un labyrinthe dont toutes les cellules sont closes. Puis, une cellule est aléatoirement choisie comme point de départ et on la marque comme étant *visitée*. Parmi les cellules voisines, on en choisie une non visitée et on abat la cloison entre les deux cellules. On réitère la procédure avec cette cellule voisine comme nouvelle cellule. Le labyrinthe est fini de construire lorsque l'exploration est finie, c'est à dire lorsqu'il n'y a plus de cellules non visitées.

L'algorithme 2 décrit cette procédure de manière itérative à l'aide d'une pile pour garder la trace des cellules dont il faut chercher les voisins : tant qu'on n'a pas visité tous les voisins d'une cellule, celle-ci doit être présente dans la pile.

Pour l'implémentation, on utilisera une `list` Python pour implémenter une pile, en utilisant les méthodes `append((i,j))` pour empiler et `pop()` pour dépiler. Il faudra également se servir des fonctions programmées à la section B, `create_all_closed_maze`, `tear_down` et `dir_unvisited_neighbours`.

E1. Pour choisir une cellule au hasard, on choisit d'utiliser les fonctions `randrange` et `choice` de la bibliothèque `random`. En utilisant une directive d'importation, importer ces fonctions de cette bibliothèque en une seule ligne de code.

E2. Écrire la fonction `explore_generate(n_rows, n_cols)` qui implémente l'algorithme 2.

E3. Tester l'algorithme sur des dimensions quelconques et visualiser les résultats. Que peut-on dire de la complexité temporelle de cet algorithme ?

E4. Tracer l'évolution du temps d'exécution de la fonction `explore_generate` en fonction des dimensions du labyrinthe. On pourra supposer que celui-ci est carré.

E5. Peut-on trouver la sortie de n'importe quel labyrinthe ainsi généré ? Y-t-a-il une limite de dimension du labyrinthe ? Pourquoi ? Comment pourrait-on y remédier ?

Algorithme 2 Générer un labyrinthe en explorant

```
1 : Fonction EXPLORER_GÉNÉRER(n_lignes, n_cols)
2 :     maze ← un labyrinthe dont les cellules sont fermées de dimension (n_lignes, n_cols)
3 :     visited ← un tableau de booléens initialisés à faux de même dimension que maze
4 :     (i,j) ← une cellule de départ choisie au hasard
5 :     pile ← une pile vide                          ▷ On l'implémentera par une list Python
6 :     Ajouter (i,j) à pile
7 :     visited[i][j] ← Vrai
8 :     tant que la pile n'est pas vide répéter       ▷ Il y a encore des cellules voisines à explorer.
9 :         (i,j) ← dépiler(pile)                     ▷ On prend la dernière qu'on a visité.
10 :        n_dir ← les directions des voisins non visités
11 :        si n_dir possède au moins deux éléments alors   ▷ Il y a plusieurs voisins à considérer.
12 :            empiler(pile, (i,j))                ▷ Il faudra donc revenir sur cette cellule.
13 :        si il y a des voisins non visités alors
14 :            dir ← une direction choisie aléatoirement dans n_dir   ▷ un voisin à la fois.
15 :            selon cette direction dir, on a la cellule (p,q)
16 :            abattre le mur reliant (p,q) à (i,j)
17 :            empiler(pile, (p,q))                ▷ C'est notre nouvelle cellule!
18 :            visited[p][q] ← Vrai
19 :    renvoyer maze
```

F Générer un labyrinthe par fusion de chemins

Une seconde approche pour générer un labyrinthe est de créer des chemins et de les fusionner comme indiqué sur l'algorithme 3. La situation de départ est un labyrinthe dont toutes les cellules sont closes. Les cellules sont donc toutes isolées les unes des autres : le labyrinthe est constitué de `N_ROWS ×N_COLS` parties non connexes, des débuts de chemins.

Tant qu'il existe au moins deux composantes non connexes distinctes dans le labyrinthe, c'est à dire tant qu'il existe au moins deux chemins qui ne se rejoignent pas, on choisit au hasard un mur entre deux cellules n'appartenant pas au même chemin et on abat ce mur. On a ainsi relié deux chemins qui formaient deux composantes non connexes du labyrinthe.

L'algorithme s'achève lorsqu'il n'existe plus qu'un seul chemin. Comme à chaque tour de boucle on diminue d'un élément le nombre de composantes non connexes du labyrinthe, il est nécessaire de répéter ceci `N_ROWS×N_COLS` fois.

On choisit de représenter une composante connexe (un chemin) d'après la cellule qui l'a démarré. À chaque fois qu'on ajoute une cellule au chemin, celle-ci a pour parent la cellule qui l'a intégré dans le chemin. **En remontant les parents de chaque cellule de proche en proche, on finit par trouver la cellule qui a initié le chemin : celle-ci est son propre parent**.

Dans le code, cela se traduit par l'utilisation d'un dictionnaire nommé `parents` dont les clefs sont des cellule (i,j) et les valeurs d'autres cellules (p,q). Par exemple, si (p,q) a été intégrée au chemin par (i,j), alors `parents[(p,q)]` vaut (i,j). **Au démarrage, comme toutes les cellules sont closes, elles sont toutes leur propre parent.**

F1. Écrire les lignes de code pour initialiser le dictionnaire `parents` correctement au début du programme.

F2. On a extrait un chemin d'un dictionnaire `parents` :

Algorithme 3 Générer un labyrinthe par fusion de chemin

```
1 : Fonction FUSIONNER_GÉNÉRER(n_lignes, n_cols)
2 :     maze ← un labyrinthe dont les cellules sont fermées de dimension (n_lignes, n_cols)
3 :     parents ← un dictionnaire qui recense le parent de chaque cellule.
4 :     tant que il y a des chemins non connexes répéter
5 :         Choisir au hasard une cellule (i,j) puis une seconde (p,q) voisine de celle-ci et non connexe
6 :         La cellule à l'origine du chemin de (i,j) devient la cellule à l'origine du chemin de (p,q).
7 :         Abattre le mur reliant les cellules (i,j) et (p,q).
8 :     renvoyer maze
```

```
1 path = {(0, 0): (0, 0), (0, 1): (0, 0), (0, 2): (0, 1), (0, 3): (0, 2)}
```

(a) Quel chemin `path` représente-t-il? Quelle est la cellule qui a démarré le chemin?

(b) Écrire une fonction de prototype `find_origin(parents, c)` où `parents` est un dictionnaire comme on l'a décrit plus haut, `c` une cellule de type (i,j) et qui renvoie la cellule à l'origine du chemin auquel appartient `c`.

F3. Écrire une fonction de prototype `fusion_generate(n_rows, n_cols)` qui implémente l'algorithme 3.

G Comparer les méthodes de génération

Les deux méthodes de génération de labyrinthe (par exploration et par fusion de chemins) engendre des résultats très différents.

G1. Comparer la longueur des chemins nécessaires à la découverte de la sortie d'un labyrinthe selon que le labyrinthe a été engendré par la première ou la seconde méthode. On choisira des labyrinthes de taille 30×50 et on répètera l'expérience une centaine de fois.

G2. Interpréter les résultats.