Travaux pratiques

Parcours dans un labyrinthe

À partir d'une grille rectangulaire $p \times q$ on obtient un labyrinthe en ajoutant un certain nombre de murs pour séparer deux cases voisines. On convient en outre que ce labyrinthe est entouré de murs sur toute sa périphérie, de sorte qu'il n'est pas possible de sortir de la grille. La case située en haut à gauche, de coordonnées $(0,0)$, est la case de départ, celle située en bas à droite, de coordonnées $(p-1, q-1)$, celle d'arrivée. Nous supposerons que tous ces labyrinthes ont une solution, à savoir un chemin reliant le départ à l'arrivée.

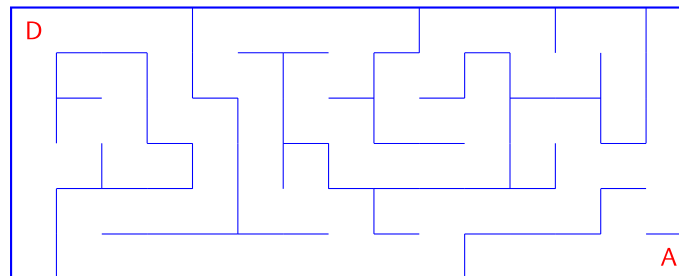

Figure 1 – Un exemple de labyrinthe tracé sur une grille 6×15.

Dans un premier temps, nous allons nous intéresser à la façon de trouver une solution ; dans un second temps nous verrons comment générer un certain type de labyrinthe : les labyrinthes *parfaits*, dans lesquels deux cases quelconques peuvent toujours être reliées par un unique chemin.

1. Parcours de labyrinthe

Récupérer à l'adresse `http://info-llg.fr/commun-mp/?a=td` le fichier `laby.py` et sauvegardez-le dans votre répertoire de travail. Votre document principal, *qui devra être enregistré dans le même répertoire que le fichier* `laby.py`, devra débuter par la commande :

```
from laby import *
```

Vous pouvez en consulter le contenu mais ce n'est pas nécessaire. Ce fichier contient la définition d'une classe `Labyrinthe`. On obtient un labyrinthe en créant une instance de cette classe ; deux arguments doivent être donnés : le nombre de lignes et le nombre de colonnes.

```
lab = Labyrinthe(15, 25)
```

Une fois créé, un labyrinthe peut être visualisé grâce à la méthode `show()` :

```
lab.show()
```

Comme vous pouvez le constater, à sa création un labyrinthe est constitué de $p \times q$ cases toutes séparées par des murs. Plus précisément, à sa création un labyrinthe possède trois attributs :

- `self.p` est le nombre de lignes ;
- `self.q` est le nombre de colonnes ;
- `self.tab` est un tableau de p lignes et q colonnes contenant des *cases*. Ces dernières possèdent quatre attributs `N`, `E`, `S`, `W` initialement égaux à `False`, qui indiquent la présence ou non d'un passage dans la direction correspondante.

Par exemple, pour le labyrinthe représenté figure 1 la case de départ est `self.tab[0][0]` et ses attributs valent :

```
self.tab[0][0].N = False        self.tab[0][0].E = True
self.tab[0][0].S = True         self.tab[0][0].W = False
```

La méthode `create` permet de percer un certain nombre de murs pour créer un labyrinthe fonctionnel :

```
lab.create()
lab.show()
```

Recherche d'une solution
Maintenant que le labyrinthe est créé, nous allons nous intéresser à la manière d'en sortir. La démarche que nous allons suivre pour sortir du labyrinthe s'apparente à celle de Thésée dans l'antre du Minotaure : utiliser un fil d'Ariane pour garder trace du chemin, et un morceau de craie pour marquer les cases déjà explorées. Le principe de l'algorithme est le suivant : lorsqu'on arrive sur une case, celle-ci est marquée puis on se rend sur l'une de ses voisines non encore marquée. Lorsqu'il n'en existe pas, on revient sur nos pas en rembobinant le fil d'Ariane jusqu'à la dernière intersection possédant une branche non encore explorée. Cette démarche porte le nom de *parcours en profondeur* car chaque route est explorée dans son entier avant d'en explorer une nouvelle.

Question 1. Pour marquer les cases explorées, on utilise un tableau `dejavu` de taille $p \times q$ contenant initialement la valeur `False` dans chacune de ses cases. Marquer une case revient à faire passer cette valeur à `True`.
Le fil d'Ariane sera représenté par une pile : avancer dans le labyrinthe revient à empiler les coordonnées (i, j) des cases parcourues ; revenir sur ses pas revient à supprimer le sommet de la pile.
À l'aide de ces deux indications, rédiger une fonction `explorer(lab)` qui prend en argument un labyrinthe et qui renvoie une pile contenant les cases à parcourir pour relier le départ à l'arrivée. On utilisera la classe `Pile` fournie dans le fichier `laby.py`, qui possède les méthodes `empty`, `push` et `pop`.

Remarque. Pour pouvoir visualiser la solution trouvée, la méthode `show` de la classe `Labyrinthe` possède un argument optionnel qui, lorsqu'il est présent et égal à une pile de cases, affiche le chemin que cette pile décrit. Ainsi, si `p` est la pile que vous a renvoyée votre fonction `explorer`, vous pourrez vérifier que votre solution est correcte en écrivant :

```
lab.show(p)
```

2. Génération de labyrinthes

Nous allons maintenant nous intéresser à la génération « aléatoire » de labyrinthes. Il n'existe pas de solution canonique à ce problème, aussi allons nous nous restreindre à la création de labyrinthes dits *parfaits*, c'est à dire lorsque deux cases quelconques peuvent toujours être reliées par un *unique* chemin.

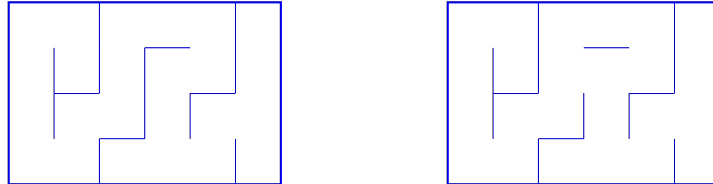

Figure 2 – À gauche, un labyrinthe parfait ; à droite un labyrinthe imparfait.

Génération par exploration exhaustive

La première méthode que nous allons mettre en œuvre consiste à partir d'un labyrinthe dont toutes les cases sont isolées par des murs. Une case est choisie arbitrairement et marquée comme étant « visitée ». On détermine ensuite quelles sont les cases voisines et non visitées, on en choisit une au hasard que l'on marque comme ayant été visitée, et on ouvre le mur les séparant. On recommence ensuite avec la nouvelle case. Lorsqu'il n'y a plus de case voisine non visitée, on revient à la case précédente.
Lorsqu'on est revenu à la case de départ et qu'il n'y a plus de possibilité, le labyrinthe est terminé.
On observera que cette démarche est très voisine de la recherche d'une solution que nous avons mis en œuvre à la question précédente : on utilise un tableau `dejavu` pour marquer les cases visitées, et le parcours est géré par une pile.

Question 2. Rédiger une fonction `creation1(lab)` qui prend en argument un labyrinthe dont toutes les parois sont fermées et qui en ouvre un certain nombre jusqu'à obtenir un labyrinthe parfait.
On prendra garde au fait que pour ouvrir une paroi, il faut creuser des deux côtés. Par exemple, pour ouvrir la paroi située entre les cases (i, j) et $(i+1, j)$ il faudra faire passer les deux valeurs `lab.tab[i][j].S` et `lab.tab[i+1][j].N` à `True`.

Question 3. Déduire de la question précédente le nombre de murs internes dans un labyrinthe parfait de taille $p \times q$.

Génération par fusion des chemins

La seconde méthode démarre elle aussi d'une grille dans laquelle toutes les cases sont isolées ; il y a donc autant de composantes connexes que de cases, à savoir $p \times q$. Tant qu'il existe au moins deux composantes connexes distinctes, on choisi arbitrairement une paroi ; si celle-ci sépare deux composantes connexes distinctes, on ouvre la paroi, reliant ainsi ces deux composantes. À la fin du processus, il n'y a plus qu'une seule composante connexe, qui constitue un labyrinthe parfait.

Union-find
Ce second algorithme demande de gérer une suite de partitions d'un ensemble (ici l'ensemble des cases de la grille), partitions supportant deux opérations : `find(x)` qui donne un représentant unique de la classe à laquelle appartient l'élément x, et `union(x, y)` qui réunit les classes des éléments x et y.
Il existe une structure très efficace pour gérer ces partitions : la structure d'*union-find*, définie dans le fichier `laby.py`. Elle possède une fonction de création : si `lst` est la liste des singletons du départ (pour vous ce sera la liste des couples (i, j) pour $0 \leqslant i < p$ et $0 \leqslant j < q$), `partition = UnionFind(lst)` crée cette structure.
Une fois créée, on dispose de deux méthodes :

- `partition.find(x` renvoie un représentant unique de la classe de l'élément x) ;
- `partition.union(x, y)` fusionne les classes des éléments x et y.

Question 4. Rédiger une fonction `creation2(lab)` qui prend en argument un labyrinthe dont toutes les parois sont fermées et qui en ouvre un certain nombre jusqu'à obtenir un labyrinthe parfait, en suivant cette fois la méthode de fusion des chemins.

Remarque à destination des étudiants de l'option informatique. La génération des labyrinthes est équivalente à la recherche d'un arbre couvrant sur un graphe constitué d'une grille $p \times q$. La première méthode est adaptée de l'algorithme de Prim, la seconde de l'algorithme de Kruskal.

Comparaison des deux méthodes

Les labyrinthes générés par ces deux méthodes ne sont pas tout à fait équivalents du point de vue qualitatif : la première méthode donne lieu à quelques chemins assez longs avec peu de ramifications, ces dernières étant en général assez courtes. Au contraire, les labyrinthes générés par la seconde méthode présentent de nombreuses bifurcations.

Question 5. Corroborer cette observation en évaluant pour chacune des deux méthodes la longueur moyenne des solutions pour un labyrinthe de taille 50×50, en réalisant une centaine d'expériences.