7 Application : générateur aléatoire de labyrinthes

À l'aide de toutes les notions que vous avez vues ce semestre, vous allez créer dans ce chapitre un programme Python capable de générer aléatoirement des labyrinthes et les afficher sous forme d'image.

7.1 Pré-requis : le module PIL

Le module PIL, acronyme de *Python Image Library*, est un module dédié à la création, la modification et le traitement d'images aux formats classiques (PNG, JPEG, etc.). C'est lui que nous allons utiliser pour créer un rendu graphique des labyrinthes que nous génèrerons. Plus précisément, nous allons réaliser l'importation suivante :

```
from PIL import Image, ImageDraw
```

Ceci nous permettra d'utiliser les deux *classes* suivantes (vous pouvez penser une classe comme un nouveau type d'objet permettant de représenter un concept plus riche que ceux déjà existants : entier, flottant, booléen, etc.) :

— la classe `Image` qui représente une image, c'est-à-dire un quadrillage formé de *pixels* de couleur,

— la classe `ImageDraw.Draw` qui représente un objet capable de dessiner sur une image.

Métaphoriquement, un objet de type `Image` sera une feuille de papier, et un objet de type `ImageDraw.Draw` sera un dessinateur capable de dessiner des formes géométriques simples sur la feuille de papier qu'on lui aura donnée.

7.1.1 Utilisation de la classe Image

On peut créer une nouvelle image vierge en utilisant la fonction `Image.new(mode, size, color)`, qui prend les arguments suivants :

— `mode` (obligatoire) : une chaîne de caractères qui représente le *mode de couleurs* qui sera utilisé. Sans rentrer dans les détails techniques, nous utiliserons toujours le mode `"RGB"` dans lequel on associe à chaque pixel un triplet (r, v, b) dont les composantes représentent respectivement les quantités de rouge, de vert et de bleu formant ensemble la couleur voulue,

— `size` (obligatoire) : un tuple formé de deux entiers (w, h) représentant respectivement la largeur (*width*) et la hauteur (*height*) de l'image créée,

— `color` (facultatif) : un tuple (r, v, b) représentant la couleur de remplissage de l'image (c'est-à-dire la couleur donnée initialement à chacun des pixels qui composent l'image). Si on ne donne pas ce paramètre, alors c'est la couleur noire qui sera utilisée.

Remarque. Le modèle RGB (pour *red, green, blue*) repose sur la synthèse *additive* des couleurs, c'est-à-dire que :

— une absence de couleur donne du noir,

— une superposition de rouge, vert et bleu purs donne du blanc,

— du rouge et du vert donnent du jaune,

— du rouge et du bleu donnent du magenta,

— du vert et du bleu donnent du cyan.

Dans PIL (comme dans beaucoup d'autres contextes en informatique), les quantités de rouge, de vert et de bleu sont décrites par des entiers compris entre 0 et 255 inclus. Autrement dit, les couleurs listées ci-

dessus seront représentées respectivement par les tuples $(0, 0, 0)$, $(255, 255, 255)$, $(255, 255, 0)$, $(255, 0, 255)$ et $(0, 255, 255)$.

Une fois un objet `img` de type `Image` créé, on peut appeler à partir de celui-ci les méthodes :

— `img.show()` pour afficher l'image dans une fenêtre externe,

— `img.save(name)` pour sauvegarder l'image sous le nom indiqué dans la chaîne de caractères `name`.

À tester. Exécutez le code suivant.

```
from PIL import Image

img = Image.new("RGB", (400, 300), (0, 0, 255))
img.show()
img.save("mon_image.png")
```

1. Vérifiez qu'un fichier appelé `mon_image.png` a été créé dans le répertoire de travail courant (en général le dossier contenant le script Python que vous venez d'exécuter).
2. Modifiez les valeurs contenues dans le triplet $(0, 0, 255)$ pour tester plusieurs couleurs.

7.1.2 Utilisation de la classe ImageDraw.Draw

On crée un objet de type `ImageDraw.Draw` en appelant simplement la fonction `ImageDraw.Draw(img)` où `img` est un objet de type `Image` que vous aurez créé au préalable. Ceci vous renvoie un objet capable de dessiner sur l'image `img` à l'aide de différentes méthodes. En voici deux d'entres elles (mais il en existe bien d'autres !) :

1. `line(points, fill, width)` permet de dessiner une ligne brisée, avec :
   — `points` : une liste de tuples `[(x1, y1), (x2, y2), ...]` contenant les coordonnées des points que vous souhaitez relier entre eux,
   — `fill` : un tuple (r, v, b) représentant la couleur de trait à utiliser,
   — `width` : l'épaisseur de trait à utiliser (en pixels).
2. `rectangle(corners, fill, outline, width)` permet de dessiner un rectangle, avec :
   — `corners` (obligatoire) : une liste `[(x1, y1), (x2, y2)]` contenant les coordonnées (sous forme de tuples) des deux coins opposés du rectangle à dessiner,
   — `fill` (facultatif) : un tuple (r, v, b) représentant la couleur de remplissage à utiliser (ou, par défaut, `None` pour ne pas remplir le rectangle),
   — `outline` (facultatif) : un tuple (r, v, b) représentant la couleur de trait à utiliser (ou, par défaut, `None` pour ne pas dessiner le bord du rectangle),
   — `width` (facultatif) : l'épaisseur de trait à utiliser (en pixels). Vaut 1 par défaut.

Attention. Pour PIL, l'origine du système de coordonnées se trouve dans le coin supérieur gauche de l'image. C'est-à-dire que dans une image de taille 800×600 par exemple, le coin en haut à gauche a pour coordonnées $(0, 0)$ tandis que le coin en bas à droite a pour coordonnées $(800, 600)$.

À tester. Essayez le code suivant.

```
from PIL import Image, ImageDraw

img = Image.new("RGB", (400, 300), (192, 192, 192))
draw = ImageDraw.Draw(img)

draw.rectangle([(10, 10), (190, 290)], fill=(255, 0, 0))
draw.line([(0, 150), (400, 150)], (0, 0, 0), 4)
draw.rectangle([(210, 10), (390, 290)], fill=(0, 0, 255))
draw.line([(20, 20), (380, 20), (380, 280), (20, 280)], (255, 255, 0), 2)

img.show()
```

Exercice 7.1. À l'aide de PIL, reproduisez (approximativement) les drapeaux officiels des pays suivants :

1. Maurice
2. Suisse
3. Islande
4. Royaume-Uni

7.2 Cahier des charges

7.2.1 Modélisation d'un labyrinthe

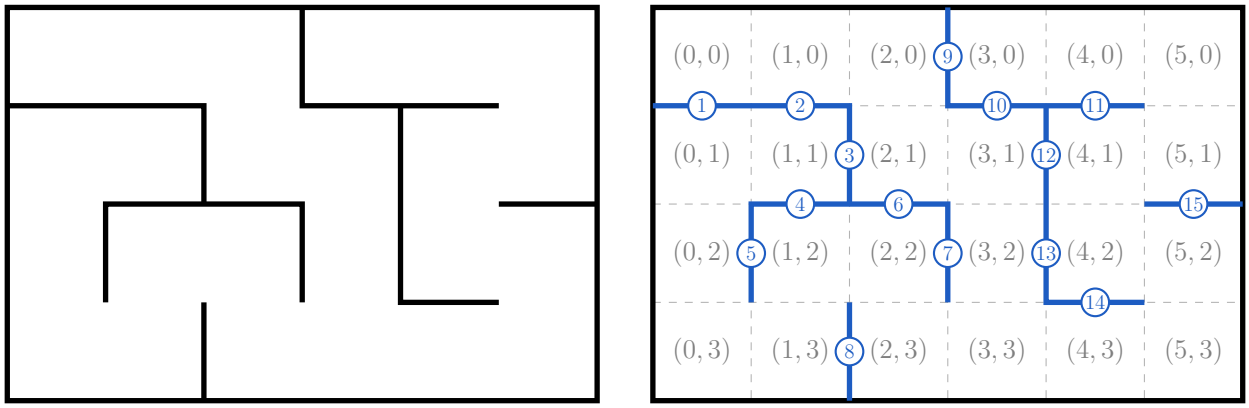


FIGURE 7.1 – Un exemple de labyrinthe de taille 6×4

Imaginons une grille formée de W colonnes et de H lignes. On peut repérer chacune des cases de cette grille par ses coordonnées (x, y) avec $0 \leq x < W$ et $0 \leq y < H$, en prenant pour convention la case en haut à gauche pour origine. Construire un labyrinthe à partir de cette grille revient donc à ajouter des murs de largeur 1 unité (horizontaux ou verticaux) entre deux cases adjacentes de la grille. Par exemple, le labyrinthe illustré sur la figure 7.1 a été obtenu en dessinant 15 murs dans une grille de taille 6×4. On peut associer à chaque mur un tuple (x_1, y_1, x_2, y_2) contenant les coordonnées des deux cases adjacentes séparées par ce mur. Par exemple, toujours dans le cas du labyrinthe de la figure 7.1 :

— le mur n° 1 sépare les cases $(0, 0)$ et $(0, 1)$, on le représentera par le tuple $(0, 0, 0, 1)$,

— le mur n° 3 sépare les cases $(1, 1)$ et $(2, 1)$, on le représentera par le tuple $(1, 1, 2, 1)$,

— le mur n° 14 sépare les cases $(4, 2)$ et $(4, 3)$, on le représentera par le tuple $(4, 2, 4, 3)$.

Pour éviter toute ambiguïté, on pourra éventuellement imposer la convention $x_1 \leq x_2$ et $y_1 \leq y_2$ (sans quoi le mur n° 1 aurait pu être représenté aussi par le tuple $(0, 1, 0, 0)$, le mur n° 3 par $(1, 2, 1, 1)$, etc.).

Finalement, on représentera un labyrinthe par un **dictionnaire** possédant les entrées suivantes :

— à la clé `"width"` on associera un **entier** donnant la largeur du labyrinthe (c'est-à-dire le nombre de colonnes de la grille),

— à la clé `"height"` on associera un **entier** donnant la hauteur du labyrinthe (c'est-à-dire le nombre de lignes de la grille),

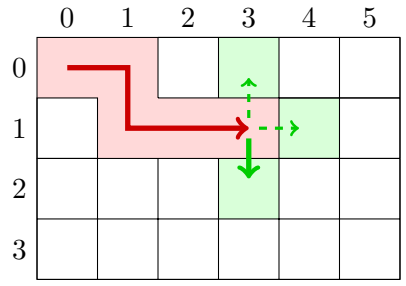
— à la clé `"walls"` on associera un **ensemble** contenant tous les murs du labyrinthes, représentés par des tuples (x_1, y_1, x_2, y_2) conformément à la convention présentée plus haut.

Par exemple, le labyrinthe de la figure 7.1 sera décrit par le dictionnaire suivant :

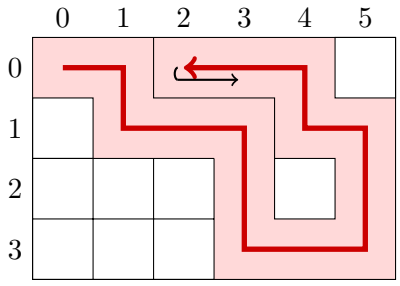
```
maze = {
    "width": 6,
    "height": 4,
    "walls": {(0, 0, 0, 1), (1, 0, 1, 1), (1, 1, 2, 1), (1, 1, 1, 2),
              (0, 2, 1, 2), (2, 1, 2, 2), (2, 2, 3, 2), (1, 3, 2, 3),
              (2, 0, 3, 0), (3, 0, 3, 1), (4, 0, 4, 1), (3, 1, 4, 1),
              (3, 2, 4, 2), (4, 2, 4, 3), (5, 1, 5, 2)}
}
```

7.2.2 Algorithme utilisé

Pour générer aléatoirement un labyrinthe, on utilisera l'algorithme suivant, reposant sur un principe de *retour sur trace* (ou *backtracking* en anglais). On commence par générer une grille complètement fermée, c'est-à-dire dans laquelle deux cases voisines quelconques sont toujours séparées par un mur. Puis, en partant initialement d'une case quelconque (par exemple la case de coordonnées $(0, 0)$), on se déplace dans la grille case par case en suivant les règles ci-dessous.



S'il reste des cases voisines qui n'ont pas encore été visitées, alors on se déplace vers l'une d'entre elles (au hasard) en supprimant au passage le mur franchi.



Sinon, on revient sur nos pas d'une case.

Lorsqu'on arrive à la situation où on doit revenir sur nos pas alors qu'on est déjà revenu jusqu'au point de départ, c'est que l'algorithme est terminé. Une figure interactive est disponible sur Moodle pour mieux visualiser cet algorithme.

7.2.3 Fonctions à implémenter

Votre programme devra implémenter toutes les fonctions suivantes, en respectant scrupuleusement les noms, paramètres, arguments par défaut et valeurs de retours imposées.

— **`new_closed_maze`** – Renvoie un dictionnaire décrivant un labyrinthe complètement « fermé », c'est-à-dire dans lequel *toutes* les paires de cases adjacentes possibles sont séparées par un mur.

Arguments :

— `width` (obligatoire) : la largeur du labyrinthe à créer (en nombre de cases),

— `height` (obligatoire) : la hauteur du labyrinthe à créer (en nombre de cases).

— **`maze_to_img`** – Renvoie un objet de type `Image` créé à l'aide de la bibliothèque PIL, donnant une représentation visuelle du labyrinthe (comme sur la figure 7.1, à gauche).

Arguments :

— `maze` (obligatoire) : un dictionnaire décrivant le labyrinthe à dessiner,

— `cell_size` (facultatif, par défaut 32) : la taille en pixels d'une case de la grille,

— `line_width` (facultatif, par défaut 2) : la largeur de trait à utiliser pour le tracé des murs,

— `bg` (facultatif, par défaut $(0, 0, 0)$) : la couleur de fond à utiliser,

— `fg` (facultatif, par défaut $(255, 255, 255)$) : la couleur de trait à utiliser,

— `margin` (facultatif, par défaut 8) : l'espace (en pixels) à laisser autour du labyrinthe.

— **`get_valid_neighbors`** – Renvoie un ensemble contenant toutes les cellules voisines possibles d'une cellule donnée. Par exemple dans le labyrinthe de la figure 7.1, la cellule $(2, 1)$ possède quatre voisines, mais la cellule $(5, 2)$ ne possède que trois voisines et la cellule $(0, 3)$ seulement deux.

Arguments :

— `coord` (obligatoire) : les coordonnées de la cellule dont on cherche les voisines,

— `maze` (obligatoire) : le labyrinthe dans lequel on raisonne.

— **`new_random_maze`** – Renvoie un dictionnaire décrivant un labyrinthe généré aléatoirement à l'aide de l'algorithme de retour sur trace décrit précédemment.

Arguments :

— `width` (obligatoire) : la largeur du labyrinthe à créer (en nombre de cases),

— `height` (obligatoire) : la hauteur du labyrinthe à créer (en nombre de cases).

Le but est de pouvoir créer et afficher un labyrinthe avec une suite d'instructions de la forme :

```
maze = new_random_maze(32, 24)
img = maze_to_image(maze, bg=(0, 0, 0), fg=(255, 255, 255))
img.show()
```

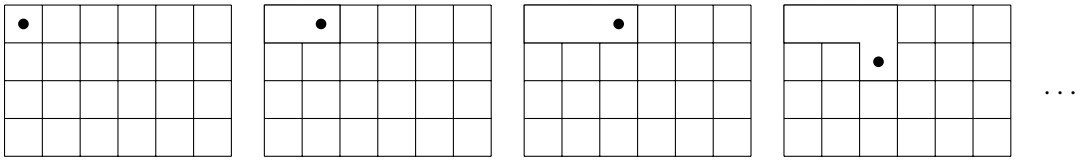
Les fonctions `new_closed_maze` et `get_valid_neighbors` sont seulement des fonctions utilitaires qui seront utilisées au sein de la fonction `new_random_maze`.

Exercice 7.2. Implémentez les fonctions décrites dans cette section en respectant scrupuleusement le cahiers des charges imposé, puis utilisez-les pour générer quelques labyrinthes et les afficher avec les options de votre choix (taille, couleur, …).

Exercices supplémentaires

Pour les plus rapides, voici quelques pistes que vous pouvez explorer pour pousser votre projet plus loin (libre à vous d'inventer les vôtres au gré de votre imagination).

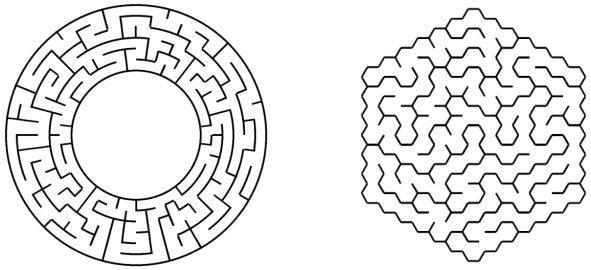
▷ **Version animée de l'algorithme.**
Complétez la fonction **`new_random_maze`** afin qu'à chaque étape de l'algorithme, elle enregistre une image du labyrinthe dans un dossier dédiée, de sorte que toutes les images créées forment ensuite une animation montrant le déroulement de l'algorithme.



Vous pourriez ensuite utiliser un outil externe tel que FFmpeg pour combiner toutes ces images en un seul fichier vidéo, mais on s'éloigne du cadre de ce cours de programmation.

▷ **D'autres algorithmes de génération aléatoire de labyrinthes.**
Créez des variations de la fonction **`new_random_maze`** qui reposent sur d'autres algorithmes que celui proposé dans ce sujet. La page Wikipédia anglaise Maze generation algorithm peut être un point de départ pour vos recherches.

▷ **D'autres formes de labyrinthes.**
Généralisez ce projet à des formes de labyrinthes plus originales, par exemple construits sur des grilles circulaire ou hexagonales.



▷ **Résolution du labyrinthe.**

— Créez une nouvelle fonction **`solve_maze`** qui prend en arguments un tuple `start` et un tuple `end`, et qui renvoie un chemin dans le labyrinthe reliant la case de coordonnées `start` à la case de coordonnées `end`. Le chemin sera renvoyé sous forme d'une liste contenant les coordonnées de cases successives depuis la case `start` jusqu'à la case `end`.

— Modifiez la fonction **`maze_to_img`** de sorte qu'elle prenne en nouvel argument facultatif un chemin à dessiner dans le labyrinthe.

Utilisez ensuite ces deux fonctions de sorte à afficher un labyrinthe généré aléatoirement ainsi que le trajet qui permet d'aller de la case en haut à gauche jusqu'à la case en bas à droite.