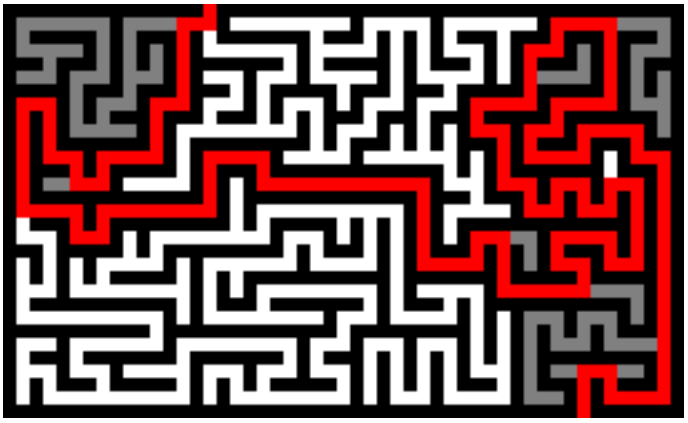
Notes de cours
Semaine 5

Cours Turing

1 Résolution d'un labyrinthe en Python

La semaine dernière nous avons vu comment générer des labyrinthes sous forme d'images en Python. Cette semaine nous allons voir comment résoudre ces mêmes labyrinthes, c'est-à-dire trouver un chemin de l'entrée à la sortie du labyrinthe. Ci-dessous est présenté un exemple de ce que l'on cherche à obtenir aujourd'hui. Le chemin de l'entrée à la sortie est affiché en rouge, les zones explorées sans succès en gris.



Pour cela, nous allons voir deux approches de résolution :

1. le parcours en profondeur, et
2. le parcours en largeur.

1.1 Bases communes

Avant de plonger dans la découverte des différentes approches, il nous faut poser quelques bases communes aux deux approches.

1.1.1 Nœud

Tout d'abord, intéressons-nous au concept de nœud. La semaine dernière, nous avions le concept de pièces reliées par des couloirs, avec une distinction entre les pièces et les couloirs. Dans l'étape du jour, nous n'aurons pas besoin de faire cette différence, et pour cela nous

utiliserons juste le concept de *nœud*. Les couloirs et les pièces de l'étape précédentes seront les nœuds de l'étape du jour. Les murs de nos labyrinthes ne seront pas considérés comme des nœuds.

En termes d'implémentation, chaque nœud correspondra à exactement un pixel de l'image du labyrinthe. Au départ, tous les nœuds du labyrinthe seront représentés par les pixels blancs de l'image. On identifiera donc un nœud grâce à ses coordonnées x et y.

1.1.2 Couleur des nœuds

La couleur du pixel associé à un nœud nous servira à indiquer si le nœud a déjà été visité ou encore s'il fait partie du chemin jusqu'à la sortie. Nous utiliserons un pixel gris pour indiquer que le nœud a été visité lors de la recherche mais qu'il ne mène pas à la sortie. Au final, nous utiliserons la couleur rouge pour indiquer les nœuds sur le chemin vers la sortie. La couleur blanche sera réservée aux nœuds qui n'ont pas été visités.

```
def est_libre(image, noeud):
    return image.getpixel(noeud) == (255, 255, 255)
```

1.1.3 Voisins et voisins non-visités

On appelle les nœuds situés directement au-dessus, au-dessous, à gauche ou à droite d'un nœud ses voisins. Notez que, contrairement à l'étape précédente, les voisins sont directement adjacents et ne sont pas séparés par un couloir.

Nos deux méthodes de résolution auront besoin d'une fonction pour déterminer les voisins *non-visités* d'un nœud. Les voisins non-visités sont représentés par des pixels blancs dans l'image du labyrinthe. Il sera donc intéressant d'implémenter une fonction pour donner la liste des voisins non-visités d'un nœud. Attention à la gestion des bordures de l'image.

```
def voisins_non_visites(image, noeud):
    voisins = []
    # À compléter
```

1.1.4 Nœud de départ

Dans les labyrinthes que l'on cherchera à résoudre, il y aura toujours un unique nœud de départ situé sur le haut du labyrinthe. Il sera intéressant d'implémenter une fonction pour trouver les coordonnées x et y du nœud de départ à partir de l'image du labyrinthe.

```
def noeud_depart(image):
    # À compléter
```

1.1.5 Dessiner un chemin

Lorsque nous aurons trouvé un chemin du départ à l'arrivée dans le labyrinthe, il faudra indiquer ce chemin en rouge dans l'image. Pour cela, vous pourrez utiliser la fonction suivante :

```
def dessiner_chemin(image, chemin):
    for noeud in chemin:
        image.putpixel(noeud, (255, 0, 0))
```

1.2 Parcours en profondeur

Le parcours en profondeur (*depth-first search* en anglais) est une méthode de parcours des labyrinthes et plus généralement de parcours dans ce que l'on appelle des *arbres*. Nous aurons l'occasion de parler des arbres en plus de détails tout prochainement dans la suite du cours.

Le parcours en profondeur est une méthode qui consiste à explorer les chemins jusqu'au bout avant de tenter d'explorer d'autres chemins. À chaque étape d'un chemin, à chaque nœud, on choisit arbitrairement un nœud voisin non-visité pour s'y déplacer. Au bout du chemin, soit on arrive à la sortie et on arrête le parcours, soit on tombe sur un cul-de-sac. Dans le cas d'un cul-de-sac, on rebrousse chemin jusqu'au dernier nœud avec des voisins encore non-visités. Notez que cette méthode d'exploration des labyrinthes est très similaire à celle employée la semaine dernière pour générer le labyrinthe.

En termes d'implémentation en Python, il vous faudra implémenter une fonction nommée `parcours_en_profondeur` qui prend deux arguments :

1. l'image du labyrinthe, et
2. la position de départ.

La fonction devra retourner, sous la forme d'une liste, les positions des nœuds sur le chemin du départ à l'arrivée. Les positions de départ et d'arrivée devront être contenues dans cette liste. L'image pourra être modifiée par la fonction, pour, par exemple, indiquer quel nœuds ont été visités.

```
def parcours_en_profondeur(image, pos_depart):
    x, y = pos_depart
    chemin = []
    # À compléter
    return chemin
```

Il est possible de l'implémenter à l'aide d'une boucle, auquel cas il faudra maintenir une liste Python pour contenir le chemin parcouru depuis le noeud de départ jusqu'à la cellule courante. Il est aussi possible d'implémenter cette fonction de manière récursive. Une fois le chemin retourné, vous pourrez utiliser la fonction `dessiner_chemin` afin de colorier ce chemin en rouge.

1.3 Parcours en largeur

Le parcours en largeur *breadth-first search* est une autre méthode de parcours qui explore les noeuds à des distances de plus en plus grandes du nœud de départ. Avant d'explorer un nœud à une distance $d + 1$ de la position de départ, tous les nœuds à distance d doivent avoir été explorés.

Les files Afin de réaliser un parcours en profondeur, on s'aide généralement d'une *file* (*queue* en anglais). Une file est une structure de données, une collection de valeurs, qui permet de réaliser efficacement deux opérations de base :

1. L'ajout d'un ou plusieurs éléments en fin de file.
2. Le retrait d'un élément en début de file.

Les files sont des structures de données dites FIFO (pour *first-in first-out*, littéralement *premier dedans, premier dehors*). Elles permettent de traiter les éléments dans l'ordre où ils sont insérés, comme une file d'attente au magasin ou à la cafétéria.

Notez que, à contrario, les listes Python permettent de faire efficacement des traitements dans l'ordre LIFO (pour *last-in, first-out*, *dernier dedans, premier dehors*). En effet, pour les listes Python, l'ajout et le retrait à la fin de la liste sont efficacement effectués. Pour les files, on peut ajouter efficacement à la fin et retirer efficacement au début.

En Python, il est possible d'utiliser très simplement les files à l'aide du module `collections`. Le module exporte une collection appelée `deque` (pour *double-ended queue*). Cette collection représente une file améliorée qui permet l'ajout et le retrait efficace des deux côtés de la file.

```
from collections import deque

ma_file = deque()  # Création d'une file
ma_file.append(1)  # Ajout d'un élément
ma_file.extend([2, 3, 4])  # Ajout de plusieurs éléments
elem = ma_file.popleft()  # Retrait du premier élément
```

Description du parcours en profondeur Le parcours en profondeur se base sur l'utilisation d'une file de nœuds. Initialement, la file ne contiendra que le nœud de départ. Puis, de manière répétée et tant que la file n'est pas vide, on retirera le premier élément de la file, on vérifiera que cet élément n'est pas le nœud d'arrivée, puis on ajoutera en fin de file tous les voisins non-visités du nœud. Si l'on tombe sur le nœud d'arrivée, le parcours se termine.

Cette manière de fonctionner, très simple, ne permet cependant pas directement de reconstruire un chemin du départ à l'arrivée. En effet, on perd trace du chemin qui mène à un nœud une fois ce nœud ajouté à la file. Pour reconstruire le chemin, il nous faudra nous aider d'une autre structure de données. Cette structure additionnelle nous permettra de noter, pour chaque nœud visité (autre que le nœud de départ), le nœud par lequel on était arrivé. Pour cela, la collection la plus adaptée sera un dictionnaire.

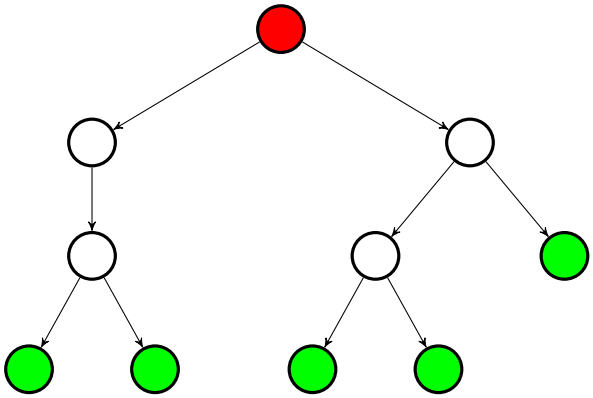
À chaque fois que l'on ajoute un nœud à la file, on ajoutera aussi au dictionnaire le nœud par lequel on y accède. Une fois le nœud d'arrivée trouvé, il suffira de consulter ce dictionnaire afin de remonter le chemin inverse.

Comparaisons avec la recherche en profondeur La recherche en profondeur et la recherche en largeur sont deux approches très similaires. D'ailleurs, en changeant un unique appel de méthode, vous devriez être capable de transformer votre implémentation de la recherche en largeur en une recherche en profondeur !

La recherche en largeur a cependant l'avantage de retourner le plus court chemin même en présence de boucles dans les labyrinthes. Ce cas de figure ne se présente pas encore, mais nous l'étudierons la semaine prochaine !

2 Les arbres

Les labyrinthes que vous avez générés la semaine passée et que nous avons résolus aujourd'hui sont des exemples d'un concept plus abstrait que l'on appelle les *arbres*. Un arbre est une structure de données formée de nœuds connectés de manière hiérarchique. Voici la représentation graphique d'un arbre.



Il existe un vocabulaire assez riche pour parler des différents éléments d'un arbre. Ci-dessous, nous allons présenter brièvement quelques-uns de ces termes.

Nœuds Les arbres sont composés de *nœuds*. Dans le schéma ci-dessus, ils sont représentés par des cercles. Dans le cas des labyrinthes, les nœuds de l'arbre correspondent aux différentes zones où il est possible de se déplacer.

Racine L'arbre a une unique *racine* (en rouge dans le schéma), un unique nœud qui est le point d'entrée dans l'arbre. Dans le cas des labyrinthes, il s'agissait du nœud de départ.

Arêtes Les nœuds d'un arbre peuvent être reliés à d'autres nœuds par le biais d'une *arête*. Les arêtes vont toujours d'un nœud à un autre nœud plus bas dans l'arbre, et ainsi ne forment jamais de *cycles* (de boucles).

Parent et enfants On appelle les nœuds directement en dessous d'un nœud et reliés à celui-ci par une arête les *enfants* du nœud. Selon la même métaphore, on appelle le *parent* d'un nœud le nœud dont il est l'enfant. Chaque nœud, à part la racine, a un unique parent.

Feuilles Certains nœuds n'ont pas d'enfants, on appelle ces nœuds des *feuilles* (en vert dans le schéma). Dans le contexte des labyrinthes, le nœud à la sortie est une feuille, tout comme les nœuds qui sont dans un cul-de-sac.

Étiquettes Les arbres peuvent aussi contenir des valeurs. Ces valeurs peuvent être liées aux nœuds comme aux arêtes. On appelle ces valeurs liées aux éléments de l'arbre des *étiquettes*.

Applications des arbres

Les arbres ont de nombreuses applications en informatique. Ils permettent de représenter des données hiérarchiques comme l'on retrouve par exemple dans les systèmes de fichiers (avec dossiers, sous-dossiers, etc.) ou l'organisation de grandes entreprises. Les arbres sont aussi au cœur d'algorithmes, comme par exemple l'algorithme de Huffman pour la compression de données. Ils se retrouvent aussi dans le domaine de l'intelligence artificielle sous la forme d'arbres de décision. Comme nous le verrons aussi à la toute fin du cours, les arbres peuvent aussi servir à représenter les expressions d'un langage de programmation et même la structure d'un programme.

Les deux méthodes de parcours que l'on a vues aujourd'hui sont applicables à n'importe quel arbre, et même à des structures plus générales appelées *graphes*. Nous aurons l'occasion d'aborder brièvement les graphes la prochaine fois, et nous traiterons le sujet plus en profondeur dans la suite du cours.