Nom : Denolle

Prénom : Quentin

Classe : CENT2

Module : Projet informatique

Responsable du module : Jean-Marie Guyader

Titre du document :

Date et heure d'envoi :

Nombre de mots :

☑ En adressant ce document à l'enseignant, je certifie que ce travail est le mien et que j'ai pris connaissance des règles relatives au référencement et au plagiat.

Sommaire

1) Introduction

L'objectif de ce projet informatique est de créer un programme en python capable de résoudre un labyrinthe à partir d'un fichier texte est d'une image en png. Pour cela nous devons utiliser le module turtle, qui est un module graphique permettant d'afficher des images, de tracer des lignes, … Notre programme doit respecter certaines contraintes (cahier des charges) :

A partir d'un fichier texte, on doit stocker la description du labyrinthe dans une liste de liste. Dans cette liste les cases 0 représentent des murs, la case 1 la case de départ, la 3 la case d'arrivé, et les cases déjà visités devront avoir la -1.

Dans une liste listPath, on doit stocker les cases visitées, et qui ne mène pas à une impasse (seulement les cases qui mène à la solution ou vers un chemin indéfini)

On doit utiliser le module turtle pour afficher l'image du labyrinthe, et le curseur en déplacement. Le tracé du curseur doit être noir et rouge lorsqu'une impasse est découverte.

Le nom du labyrinthe doit être rentré par l'utilisateur.

De plus le projet est divisé en 3 parties, la première consiste à faire une simple résolution de labyrinthe et une représentation graphique à partir de fichiers texte et images existantes. La seconde partie consiste à faire le même travail mais seulement avec le fichier texte. Et enfin la dernière partie consiste à générer un labyrinthe aléatoirement, et le résoudre sans documents ou ressources.

2) Explication du code

1) Résolution du labyrinthe

Dans un premier temps je commence par importer les modules qui vont ma servir. Ici j'ai seulement besoin du module turtle.

```
#Importation des bibliothèques
import turtle
```

Une fois que j'ai importé les modules dont j'aurai besoin, j'ai besoin de savoir quel labyrinthe le programme devra résoudre. Pour cela j'ai créé une fonction Get_chemin qui me permet d'obtenir les informations dont j'ai besoin pour résoudre le labyrinthe.

```
#Fonctions
def Get_chemin():
    '''Cette fonction permet de demander à l'utilisateur quel labyrinthe il veut lancer,
    puis d'ouvrir et prendre les données du fichier texte correspondant.
    Les informations sont placés dans une liste de liste appelée chemin.
    '''
    chemin = []
    num_lab =input("Quel est le numero du labyrinthes que vous souhaitez executer ?")
    f = open(f'Data\Maze{num_lab}.txt', 'r')
    while True:
        ligne = f.readline()
        lst_ligne = list(ligne.strip())
        if(ligne == ''):
            break
        chemin.append(lst_ligne[:])
    return chemin, num_lab
```

Je commence par créer une liste vide (chemin) qui me permettra de stocker les informations contenues dans le fichier texte correspondant.

Je demande ensuite à l'utilisateur quel labyrinthe doit résoudre le programme.

Une fois que l'on sait quel labyrinthe résoudre, on ouvre le fichier texte correspond et l'on ajoute chaque ligne du fichier (transformé en liste) à la liste chemin.

La fonction retourne à la fin la liste chemin remplie et le numéro de labyrinthe à résoudre.

Une fois que l'on a réussi à obtenir le labyrinthe à résoudre, il faut trouver le point de départ, pour cela j'ai créé une fonction Get_start_point.

```
def Get_start_point(chemin):
    '''Cette fonction permet de trouver les coordonnées du point de départ
    à partir de la liste chemin.
    '''
    start_point = ['']
    for ligne in range(len(chemin)):
        if start_point[0] != '':
            break
        for collone in range(len(chemin[ligne])):
            if chemin[ligne][collone] == '1':
                start_point = [ligne,collone]
                break
    return start_point
```

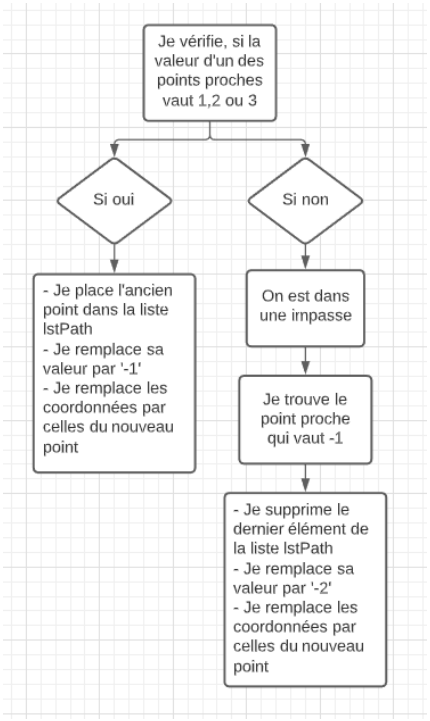
Au départ start_point est une liste vide.

Pour trouver start_point, on vérifie pour tous les éléments de toutes les lignes (listes) contenu dans la liste chemin si sa valeur est 1 (ce qui signifie que c'est le départ). Lorsque l'on a trouvé la position de l'élément qui vaut 1, on stocke ses coordonnées (sa ligne et sa colonne) dans la variable start_point (la première coordonnée correspond à la ligne et la seconde à sa colonne, c'est-à-dire sa position dans la ligne)

La fonction retourne donc la liste start_point contenant les coordonnées de la case départ.

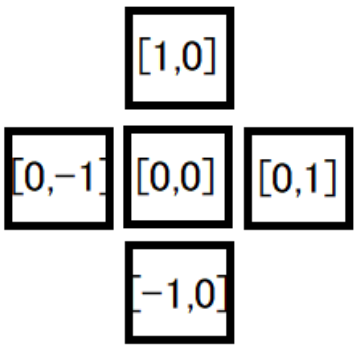
Maintenant que l'on a le point de départ, on peut commencer à résoudre le labyrinthe. Pour cela j'ai créé une fonction (Mouv_turtle) qui permet de déplacer le curseur d'une case.

```
def Mouv_turtle(chemin, point, listPath, origine):
    '''Cette fonction permet de déplacer le curseur.
    Elle permet de vérifier si des cases alentours sont libres, et si c'est le cas d'y aller.
    Si aucune case proche n'est libre elle permet de faire demi-tour, et de tracer un trait rouge
    Elle permet aussi de modifier la liste chemin et de remplir la liste listPath.
    '''
    points_proches = [[0,1],[1,0],[0,-1],[-1,0]]
    for i in range(len(points_proches)):
        if chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '1' \
        or chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '2' \
        or chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '3':
            listPath.append(point)
            chemin[point[0]][point[1]] = '-1'
            point = [point[0]+points_proches[i][0],point[1]+points_proches[i][1]]
            turtle.pencolor('black')
            turtle.goto(origine[1]+20*point[1],origine[0]+20*point[0])
            return chemin, point, listPath
    for i in range(len(points_proches)):
        if chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '-1':
            del listPath[-1]
            chemin[point[0]][point[1]] = '-2'
            point = [point[0]+points_proches[i][0],point[1]+points_proches[i][1]]
            turtle.pencolor('red')
            turtle.goto(origine[1]+20*point[1],origine[0]+20*point[0])
            return chemin, point, listPath
```



La fonction peut être divisé en deux parties (deux boucles for). La première consiste à déplacer le pion lorsque qu'il y a une case disponible à côté, la seconde partie permet de déplacer le curseur lorsqu'i n'y a plus de cases disponibles (impasse). Dans les deux parties j'utilise une liste points_proches qui contient elle-même des listes contenant des pseudos coordonnés qui additionné aux coordonnées de la position actuelle (stocké dans la variable point) vont permettre d'obtenir les points à côté de la

position actuelle. Respectivement la case de droite, puis la case au-dessus, la case de gauche et enfin la case en dessous.



Dans la première partie, on vérifie pour chaque point à côté de la position actuelle si leurs valeurs sont 1, 2 ou 3. Pour la première case correspondant à un de ces critères, on va ajouter la position actuelle à la liste listPath (qui contient chaque coordonnée de la solution), on change aussi la valeur de la position en -1 (ce qui signifie que l'on a déjà visité cette case), puis on modifie les coordonnées contenues dans la variable point par les nouvelles coordonnées et on déplace le curseur à la nouvelle position.

Dans la seconde partie (si la première partie n'a pas pu être réalisé, et qu'il n'y a donc plus de case de libre), on doit revenir sur nos pas car l'on est dans une impasse. On vérifie donc pour chaque point à côté de la position actuelle si sa valeur est -1 (une case où l'on est déjà passé). Dans ce cas on supprime le dernier élément de listPath. On remplace ensuite la valeur de la position actuelle par -2 (ce qui veut dire que l'on a déjà visité cette case et qu'elle mène à une impasse). Et enfin on modifie les coordonnées contenues dans la variable point et on déplace le curseur à la nouvelle position tout en traçant un trait rouge (impasse).

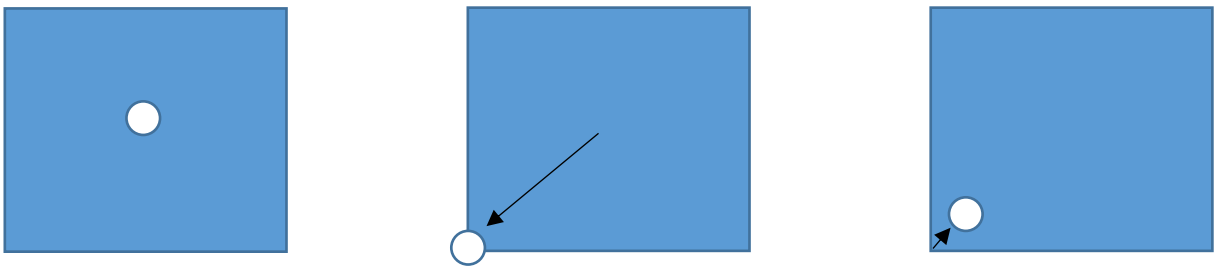
Une fois que l'on a obtenu le point de départ (appel de la fonction Get_start_point) et que l'on arrive à déplacer le curseur (fonction Mouv_turtle), il suffit de répéter la fonction Mouv_turtle jusqu'à ce que la valeur de la position soit 3 (la case d'arrivé). Pour cela j'utilise une boucle while.

```
#Initialisation des variables
listPath = []

#Programme principale
chemin, num_lab = Get_chemin()
point = Get_start_point(chemin)

while chemin[point[0]][point[1]] != '3':
    chemin, point, listPath, Impasse_Path = Mouv_turtle(chemin, point, listPath, Impasse_Path, c
```

Pour afficher une représentation graphique, j'utilise le module turtle que j'ai importé en début de programme. Après avoir récupéré le point de départ je configure la fenêtre graphique.

```
#Initialisation des variables
listPath = []

#Programme principale
chemin, num_lab = Get_chemin()
point = Get_start_point(chemin)


''' Creéation du graphique '''
origine = [-(20*len(chemin))/2 + 10, -(20*len(chemin[0]))/2 + 10]
turtle.setup(20*(len(chemin[0])+2), 20*(len(chemin)+2))
turtle.bgpic(f'Data\Maze{num_lab}.png')
turtle.title(f'Réalisation du labyrinthe {num_lab}')
turtle.up()
turtle.goto(origine[1]+20*point[1], origine[0]+20*point[0])
turtle.down()

while chemin[point[0]][point[1]] != '3':
    chemin, point, listPath = Mouv_turtle(chemin, point, listPath, origine)

turtle.exitonclick()
```

Je commence par définir un point d'origine car je préfère que l'origine du repère soit en bas à gauche plutôt qu'au milieu de l'image comme c'est le cas initialement. Pour cela chaque coordonnée du point origine doit être de : – la moitié du graphique. De plus je veux que le point origine soit placé au centre de la case je rajoute donc 10 pour chaque coordonnée.



Ensuite je règle la taille de la fenêtre graphique. Je prends donc la taille du labyrinthe auquel j'additionne deux car dans l'image le labyrinthe est entouré d'une bande blanche de 20 pixels (soit une case de chaque côté), je multiplie ensuite par 20 car c'est la taille d'une case.

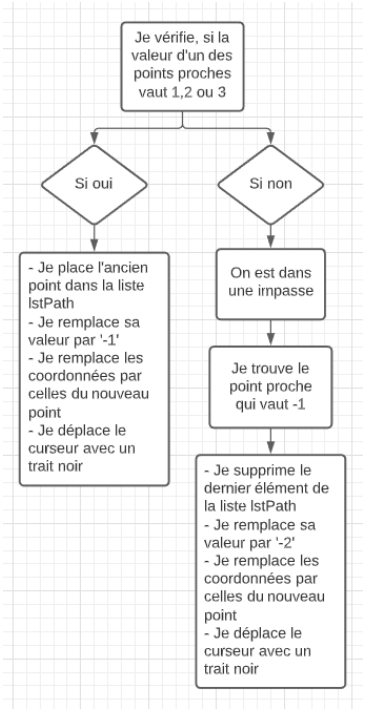
Je définie ensuite l'image de fond qui correspond au labyrinthe sélectionné et qui est placé dans un fichier Data. Je donne aussi un titre à la fenêtre (ici Réalisation du labyrinthe sélectionné).

Puis je déplace le curseur au point de départ, j'utilise pour cela les coordonnées du point d'origine auxquelles j'ajoute 20 x les coordonnées (en case) du point de départ. Pour que le tracé ne soit pas visible, je commence par lever le curseur avant de le déplacer, puis le rebaisse une fois le déplacement terminé.

A la fin du programme j'utilise la commande exitonclick qui permet de fermer la fenêtre lorsque l'utilisateur fait un click.

Une fois que le curseur est placé sur le point de départ et que la fenêtre est réglée, j'utilise la fonction Mouv_turtle qui me permet de déplacer le curseur d'une case.

```python
def Mouv_turtle(chemin, point, listPath, origine):
    '''Cette fonction permet de déplacer le curseur.
    Elle permet de vérifier si des cases alentours sont libres, et si c'est le cas d'y aller.
    Si aucune case proche n'est libre elle permet de faire demi-tour, et de tracer un trait rouge
    Elle permet aussi de modifier la liste chemin et de remplir la liste listPath.
    '''
    points_proches = [[0,1],[1,0],[0,-1],[-1,0]]
    for i in range(len(points_proches)):
        if chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '1' \
        or chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '2' \
        or chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '3':
            listPath.append(point)
            chemin[point[0]][point[1]] = '-1'
            point = [point[0]+points_proches[i][0],point[1]+points_proches[i][1]]
            turtle.pencolor('black')
            turtle.goto(origine[1]+20*point[1],origine[0]+20*point[0])
            return chemin, point, listPath
    for i in range(len(points_proches)):
        if chemin[point[0]+points_proches[i][0]][point[1]+points_proches[i][1]] == '-1':
            del listPath[-1]
            chemin[point[0]][point[1]] = '-2'
            point = [point[0]+points_proches[i][0],point[1]+points_proches[i][1]]
            turtle.pencolor('red')
            turtle.goto(origine[1]+20*point[1],origine[0]+20*point[0])
            return chemin, point, listPath
```



Il s'agit donc de la même fonction que dans la première partie à laquelle on a ajouté la fonction goto qui permet de déplacer le curseur. On déplace le curseur vers la case correspondante en additionnant les coordonnées de l'origine et 20 x les coordonnées de la nouvelle position (identique que le déplacement vers la case de départ). De plus lorsque le curseur fait demi-tour (impasse) le trait tracé est rouge, sinon il est noir.

2) Représentation graphique

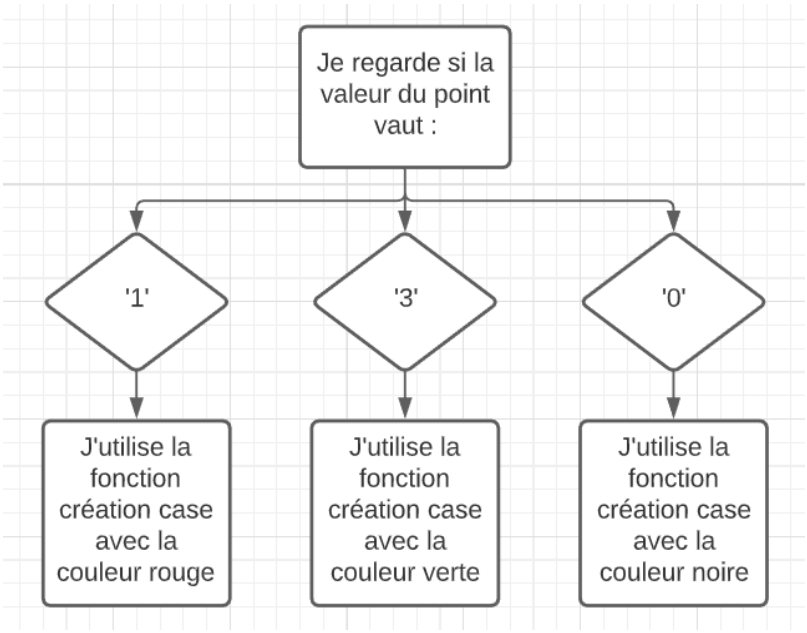
Pour la seconde partie je dois faire le même travail, mais je ne dois plus utiliser l'image de fond correspondant à l'image du labyrinthe. Je dois donc créer la représentation du labyrinthe avant de la résoudre. Je commence donc par créer une fonction qui me permet de dessiner une case (légèrement arrondie) et que j'appelle creation_case.

```
def creation_case(point, origine, color):
    turtle.pencolor(color)
    turtle.pensize(20)
    turtle.speed('fastest')
    turtle.up()
    turtle.goto(origine[1]+20*point[1], origine[0]+20*point[0])
    turtle.down()
    turtle.forward(1)
    turtle.left(90)
    turtle.backward(1)
    turtle.left(90)
    turtle.forward(1)
    turtle.left(90)
    turtle.backward(1)
    turtle.left(90)
    turtle.up()
    return
```

Je commence donc par changer la couleur du tracé puis la taille, et enfin la vitesse. Je déplace le curseur au point voulu (avec le curseur levé pour ne pas faire de tracé), puis je lui fait faire un tour complet, ce qui me permet d'obtenir une case.

Ensuite je créé une autre fonction qui me permet de déterminer la couleur du tracé, et j'utilise la fonction précédente.

```
def creation_map(point, origine):
    if chemin[point[0]][point[1]] == '1':
        creation_case(point, origine, 'red')
    elif chemin[point[0]][point[1]] == '3':
        creation_case(point, origine, 'green')
    elif chemin[point[0]][point[1]] == '0':
        creation_case(point, origine, 'black')
    return
```



Je vérifie donc la valeur d'un certain point si c'est 1 la couleur est rouge (départ), si c'est 3 la couleur est verte (arrivé) et si c'est 2 la couleur est noire (mur)

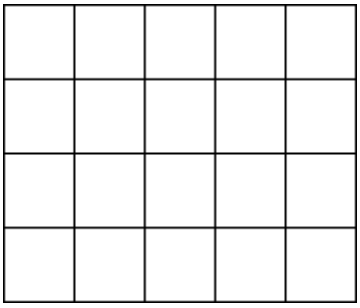
Enfin j'utilise des boucles qui me permettent d'appeler cette fonction pour chaque point du labyrinthe.

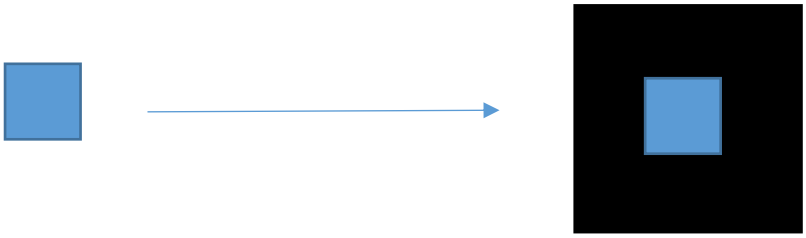
```
''' Creéation du graphique '''
origine = [-(20*len(chemin))/2 + 10, -(20*len(chemin[0]))/2 + 10]
turtle.setup(20*(len(chemin[0])+2), 20*(len(chemin)+2))
turtle.title(f'Réalisation du labyrinthe {num_lab}')

for i in range(len(chemin)):
    for j in range(len(chemin)):
        creation_map([i,j], origine)
turtle.up()
turtle.goto(origine[1]+20*point[1], origine[0]+20*point[0])
turtle.down()
turtle.pensize(1)
turtle.speed('normal')
```

Après les boucles for, je remet le curseur au départ et remet des valeurs pour la résolution du labyrinthe (taille du tracé et vitesse)

3) Génération aléatoire d'un labyrinthe

La génération aléatoire d'un labyrinthe est la partie où j'ai rencontré le plus de difficultés, et qui ne fonctionne pas toujours . Je n'arrivais pas à trouver une méthode pour créer un labyrinthe aléatoirement et qui soit solvable. J'ai donc recherche sur internet une méthode pour créer un labyrinthe, et je suis tombé sur une page wikipédia (https://fr.wikipedia.org/wiki/Mod%C3%A9lisation_math%C3%A9matique_de_labyrinthe#G%C3%A9n%C3%A9ration). Je me suis donc inspiré de la méthode de fusion des chemins, mais le problème est que dans nos labyrinthe les murs sont des cases on non des traits. J'ai donc décidé de qu'une case serai entouré de mur.





Le labyrinthe aura donc une taille trois fois plus grande que la taille initiale. J'ai donc commencé par importer le module random qui me permet de choisir des nombres aléatoirement et qui me servira par la suite. J'ai ensuite commencé à créer le labyrinthe avec une liste de liste.

```
import random

# Création du labyrinthe
lab = []
ligne = []
taille_lab = random.randint(3,10)
ligne_externe = []

for i in range(3*taille_lab):
    ligne.append('0')
for j in range(3*taille_lab):
    lab.append(ligne)
```

J'utilise donc une boucle for pour ajouter dans la liste ligne (qui correspond à une ligne) le nombre de case voulu (3 fois la taille initiale du labyrinthe, qui est elle-même choisi aléatoirement, ici entre 3 et 10 mais on peut changer), qui sont initialement des murs. J'utilise une seconde boucle for pour ajouter le nombre de lignes voulus dans la liste lab qui correspond donc aux informations contenues dans le fichier texte.

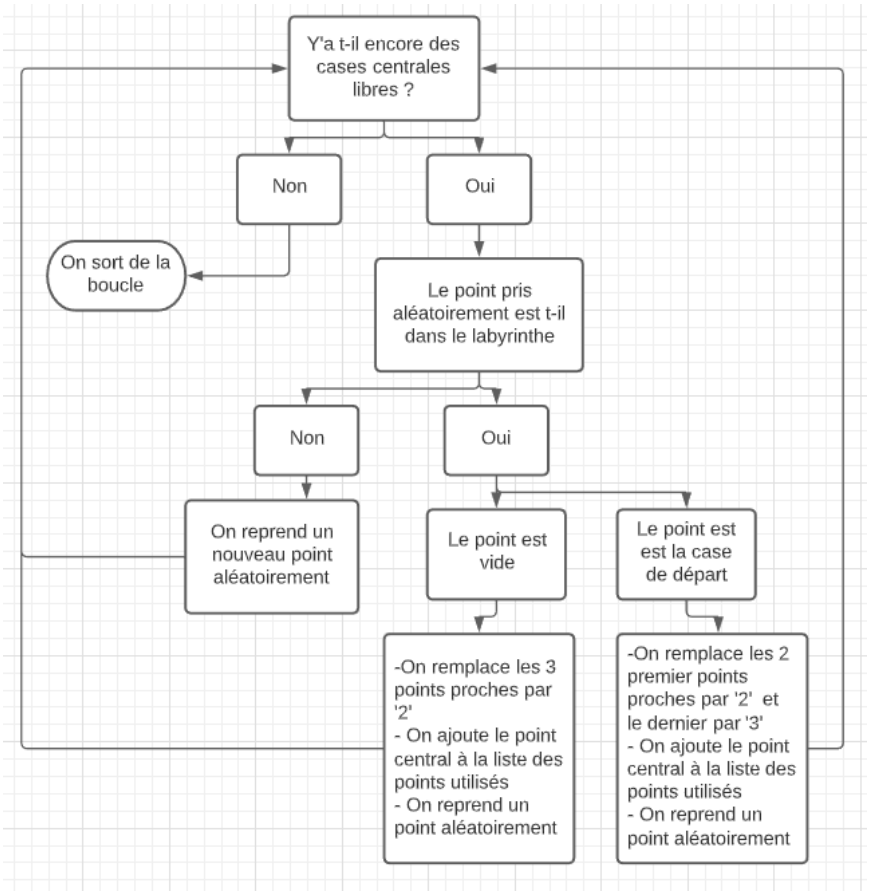
Une fois que le labyrinthe est « créé », je place le point de départ et le point d'arrivé.

```
# Positionnement du point de départ et d'arrivé
c_start = 3*random.randint(0,taille_lab-1)+1
l_start = 3*random.randint(0,taille_lab-1)+1
c_end = 3*random.randint(0,taille_lab-1)+1
l_end = 3*random.randint(0,taille_lab-1)+1
while c_end == c_start and l_end == l_start:
    c_start = 3*random.randint(0,taille_lab-1)+1
    l_start = 3*random.randint(0,taille_lab-1)+1
    c_end = 3*random.randint(0,taille_lab-1)+1
    l_end = 3*random.randint(0,taille_lab-1)+1


lab[l_start]=lab[l_start][:c_start]+['1']+lab[l_start][c_start+1:]
lab[l_end]=lab[l_end][:c_end]+['4']+lab[l_end][c_end+1:]
```

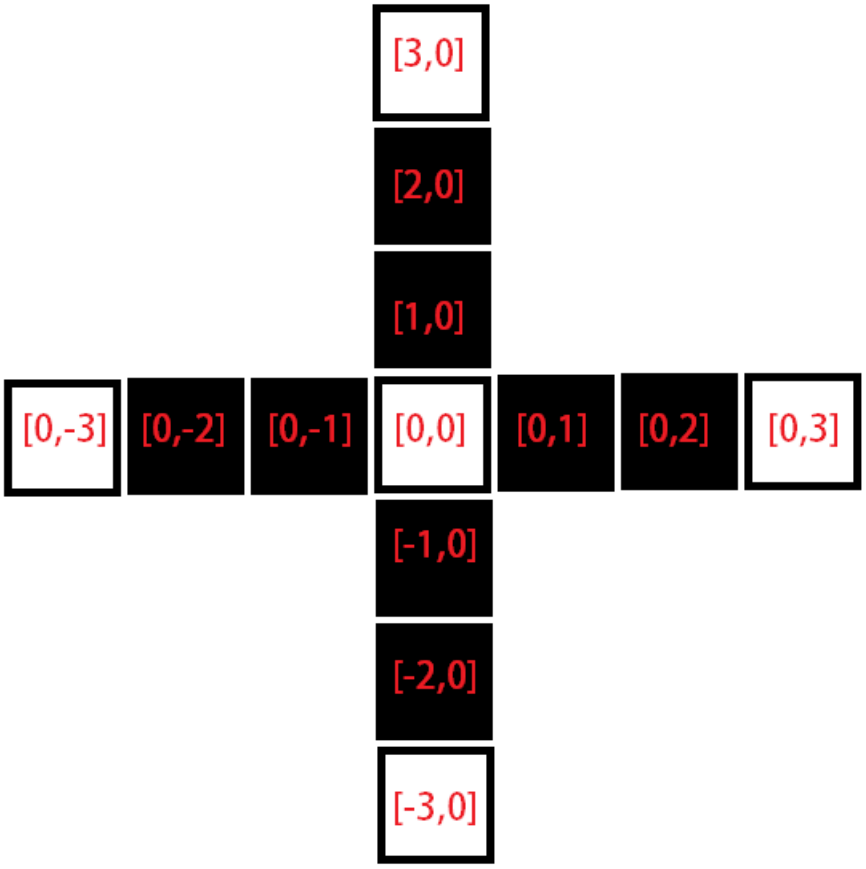
J'utilise donc le module random que je multiplie par 3 et que j'additionne par 1 pour que la case de départ ou la case d'arrivé se trouve au centre d'un carré de 3 cases de côté. Le c correspond à la colonne et le l à la ligne. Je vérifie aussi que le point de départ et le point d'arrivé ne soit pas au même endroit, puis je remplace les lignes nécessaires (pour la case d'arrivé je mets 4, mais je le change plus tard en 3).

Une fois que j'ai le point de départ et le point d'arrivé il me reste à compléter le labyrinthe en ajoutant des nouveaux points (nouvelles cases libres)

```
# Positionnement de nouveaux points
points_proches=[[[0,1],[0,2],[0,3]],[[1,0],[2,0],[3,0]],[[0,-1],[0,-2],[0,-3]],[[-1,0],[-2,0],[-3,0]]]
pts_utilises = [[l_start,c_start]]

rdm_pts_utilises = pts_utilises[random.randint(0,len(pts_utilises)-1)]
rdm_points_proches = points_proches[random.randint(0,len(points_proches)-1)]

while len(pts_utilises) < taille_lab**2:
    if rdm_points_proches[2][0]+rdm_pts_utilises[0] > len(lab)-1 or rdm_points_proches[2][1]+rdm_pts_utilises[1] > len(lab)-1
        rdm_points_proches = points_proches[random.randint(0,len(points_proches)-1)]
        rdm_pts_utilises = pts_utilises[random.randint(0,len(pts_utilises)-1)]
    else:
        if lab[rdm_points_proches[2][0]+rdm_pts_utilises[0]][rdm_points_proches[2][1]+rdm_pts_utilises[1]] == '0':
            lab[rdm_points_proches[2][0]+rdm_pts_utilises[0]] = lab[rdm_points_proches[2][0]+rdm_pts_utilises[0]][:rdm_points
            lab[rdm_points_proches[1][0]+rdm_pts_utilises[0]] = lab[rdm_points_proches[1][0]+rdm_pts_utilises[0]][:rdm_points
            lab[rdm_points_proches[0][0]+rdm_pts_utilises[0]] = lab[rdm_points_proches[0][0]+rdm_pts_utilises[0]][:rdm_points
            pts_utilises.append([(rdm_points_proches[2][0]+rdm_pts_utilises[0]),(rdm_points_proches[2][1]+rdm_pts_utilises[1]
            rdm_points_proches = points_proches[random.randint(0,len(points_proches)-1)]
            rdm_pts_utilises = pts_utilises[random.randint(0,len(pts_utilises)-1)]
        elif lab[rdm_points_proches[2][0]+rdm_pts_utilises[0]][rdm_points_proches[2][1]+rdm_pts_utilises[1]] == '4':
            lab[rdm_points_proches[2][0]+rdm_pts_utilises[0]] = lab[rdm_points_proches[2][0]+rdm_pts_utilises[0]][:rdm_points
            lab[rdm_points_proches[1][0]+rdm_pts_utilises[0]] = lab[rdm_points_proches[1][0]+rdm_pts_utilises[0]][:rdm_points
            lab[rdm_points_proches[0][0]+rdm_pts_utilises[0]] = lab[rdm_points_proches[0][0]+rdm_pts_utilises[0]][:rdm_points
            pts_utilises.append([(rdm_points_proches[2][0]+rdm_pts_utilises[0]),(rdm_points_proches[2][1]+rdm_pts_utilises[1]
            rdm_points_proches = points_proches[random.randint(0,len(points_proches)-1)]
            rdm_pts_utilises = pts_utilises[random.randint(0,len(pts_utilises)-1)]
        else:
            rdm_points_proches = points_proches[random.randint(0,len(points_proches)-1)]
            rdm_pts_utilises = pts_utilises[random.randint(0,len(pts_utilises)-1)]
```



Comme je l'ai dit précédemment j'utilise la méthode de fusion des chemins, pour cela j'ai créé deux listes, une liste pts_utilises qui contient les points qui sont déjà utilisés (seulement ceux au centre de carrés de 3 cases, et qui ne contient au départ que la case de départ), et une liste points_proches qui contient les coordonnées des déplacements (en ligne droites) de 3 cases possibles.



Je sélectionne ensuite aléatoirement (avec le module random) une case déjà utilisée et une direction.

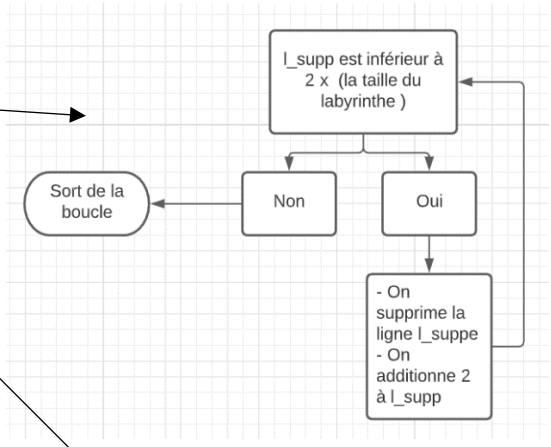
J'utilise une boucle qui s'arrête lorsque tous les points au centre des carrés de tailles 3*3 sont utilisés.

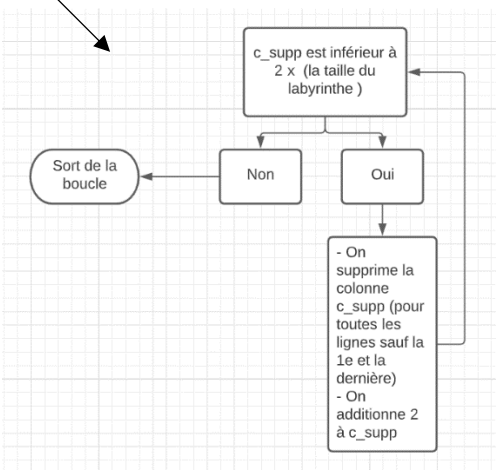
Je vérifie ensuite que le nouveau point dont les coordonnées sont les sommes de la direction et du point sélectionné, est contenu dans le labyrinthe (coordonnées supérieures à 0 et inférieures à la taille du labyrinthe). Si c'est le cas je retire aléatoirement une position et une direction.

Lorsque le nouveau point est bien dans le labyrinthe, je vérifie qu'il soit inutilisé (que ce soit un mur, ou alors la case d'arrivé qui n'est pas encore relié), puis si c'est le cas, je remplace les 3 cases qui permettent de passer du point initial au point actuel par '2' (une case libre, ou par '3' pour la case d'arrivé), j'ajoute le point aux points utilisés, puis je rechoisi aléatoirement une position et une direction. Si le nouveau point est utilisé je rechoisi aléatoirement une position et une direction.

Une fois que toutes les cases centrales sont remplies, je décide de supprimer quelques lignes et colonnes qui sont en doubles.

```python
#Suppression des lignes et collones doubles
l_supp = 2
c_supp = 2

while l_supp < 2*taille_lab:
    del lab[l_supp]
    l_supp +=2

while c_supp < 2*taille_lab:
    for i in range(len(lab)-2):
        del lab[i+1][c_supp]
    c_supp += 2
i=0
while i < len(lab):
    ligne_externe.append('0')
    i += 1

lab[0]=ligne_externe
lab[len(lab)-1] = ligne_externe

print(lab)
```





Je définis donc la première ligne et la première colonne à supprimer. Puis j'utilise des boucles qui me permettent de supprimer une ligne/colonne sur 3 (2^e, 5^e, 8^e, …). Pour modifier les colonnes, je ne touche pas aux lignes externes (première et dernière), je remplace juste par une ligne de '0' de la bonne taille. Je fais ça car quand j'utilise la boucle for il y avait trop d'éléments d'enlevés dans ces lignes et que de toutes façons elles ne contiennent que des '0'.

Pour ce qui est de la représentation graphique du labyrinthe je n'ai pas eu le temps de la faire.

3) Résultats des tests

Tous d'abord on peut tester la fonction qui nous permet d'obtenir les informations sur le labyrinthe à résoudre en lisant le fichier texte correspondant (la fonction Get_chemin).

Lorsque l'on demande le labyrinthe 1 on obtient les informations suivantes :

Indice	Type	Taille	Valeur	
0	list	21	['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', ...]	000000000000000000000
1	list	21	['1', '1', '2', '2', '0', '2', '2', '2', '2', '2', ...]	112202222202222222020
2	list	21	['0', '0', '0', '2', '0', '2', '0', '0', '0', '2', ...]	000202000202000002020
3	list	21	['0', '2', '0', '2', '2', '2', '0', '2', '2', '2', ...]	020222022222022202220
4	list	21	['0', '2', '0', '0', '0', '0', '0', '2', '0', '2', ...]	020000020200000200020
5	list	21	['0', '2', '0', '2', '2', '2', '0', '2', '0', '2', ...]	020222020222220202220
6	list	21	['0', '2', '0', '2', '0', '2', '0', '0', '0', '0', ...]	020202000000020202000
7	list	21	['0', '2', '2', '2', '0', '2', '2', '2', '2', '2', ...]	022202222222220202220
8	list	21	['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', ...]	000000000002000200020
9	list	21	['0', '2', '2', '2', '2', '2', '0', '2', '2', '2', ...]	022222022222022222020
10	list	21	['0', '2', '0', '0', '0', '2', '0', '2', '0', '0', ...]	020002020000020000020
11	list	21	['0', '2', '0', '2', '2', '2', '0', '2', '0', '2', ...]	020222020202222202220
12	list	21	['0', '2', '0', '0', '0', '0', '0', '2', '0', '2', ...]	020000020202020002000
13	list	21	['0', '2', '2', '2', '2', '2', '0', '2', '0', '2', ...]	022222020222022202020
14	list	21	['0', '0', '0', '2', '0', '2', '0', '2', '0', '0', ...]	000202020000000202020
15	list	21	['0', '2', '2', '2', '0', '2', '2', '2', '0', '2', ...]	022202220202222222020
16	list	21	['0', '2', '0', '0', '0', '0', '0', '0', '0', '2', ...]	020000000202000000020
17	list	21	['0', '2', '2', '2', '0', '2', '0', '2', '2', '2', ...]	022202022202222202220
18	list	21	['0', '0', '0', '2', '0', '2', '0', '2', '0', '2', ...]	000202020200000202020
19	list	21	['0', '2', '2', '2', '0', '2', '2', '2', '0', '2', ...]	022202220222222222023
20	list	21	['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', ...]	000000000000000000000

Le programme marche donc bien puisque la liste de listes contient les mêmes informations que le fichier texte correspondant avec le labyrinthe 1, et tous les autres labyrinthe, voici par exemple le labyrinthe 4 :

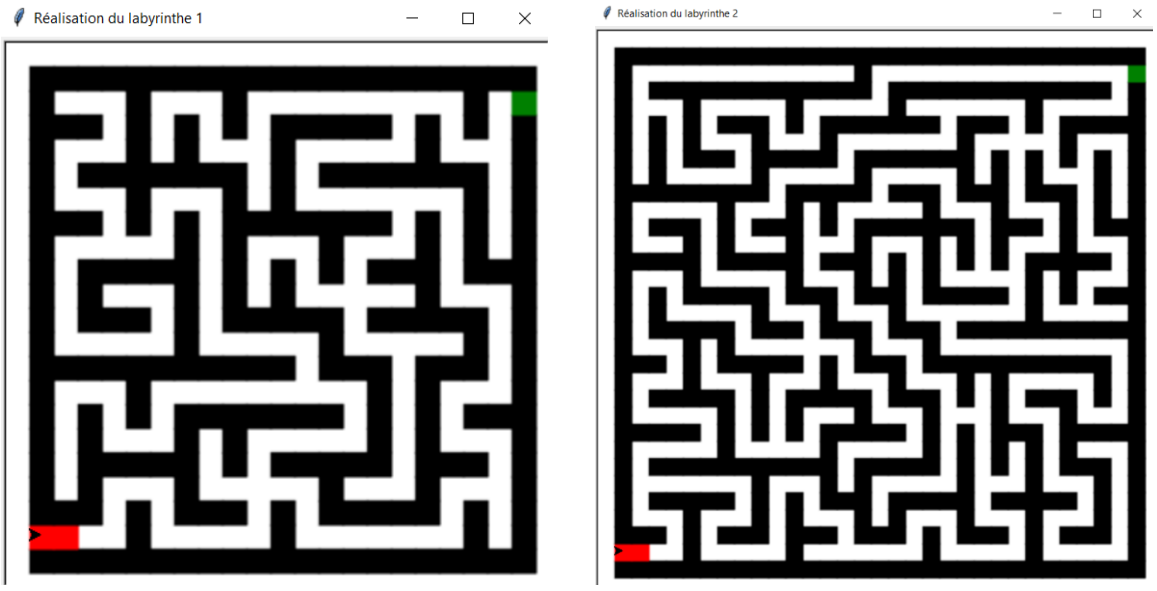
Indice	Type	Taille	Valeur	
0	list	11	['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', ...]	00000000000
1	list	11	['1', '1', '2', '2', '0', '2', '2', '2', '2', '2', ...]	11220222220
2	list	11	['0', '0', '0', '2', '0', '2', '0', '0', '0', '2', ...]	00020200020
3	list	11	['0', '2', '0', '2', '2', '2', '0', '2', '2', '2', ...]	02022202220
4	list	11	['0', '2', '0', '0', '0', '0', '0', '2', '0', '2', ...]	02000002020
5	list	11	['0', '2', '0', '2', '2', '2', '0', '2', '0', '2', ...]	02022202020
6	list	11	['0', '2', '0', '2', '0', '2', '0', '2', '0', '0', ...]	02020202000
7	list	11	['0', '2', '2', '2', '0', '2', '2', '2', '2', '2', ...]	02220222220
8	list	11	['0', '0', '0', '0', '0', '0', '0', '0', '2', '0', ...]	00000000200
9	list	11	['0', '2', '2', '2', '2', '2', '0', '2', '2', '2', ...]	02222202223
10	list	11	['0', '0', '0', '0', '0', '0', '0', '0', '0', '0', ...]	00000000000

On peut ensuite tester la fonction qui permet de trouver les coordonnées de la case de départ en rajoutant le code `print(f'Le départ du labyrinthe {num_lab}  est au point : {point}')` :

```
In [29]: runfile('C:/Users/quent/Documents/Cours/2
quent/Documents/Cours/2e année/Python/Projet')
Le départ du labyrinthe 1  est au point : [1, 0]

In [30]: runfile('C:/Users/quent/Documents/Cours/2
quent/Documents/Cours/2e année/Python/Projet')
Le départ du labyrinthe 2  est au point : [1, 0]

In [31]: runfile('C:/Users/quent/Documents/Cours/2
quent/Documents/Cours/2e année/Python/Projet')
Le départ du labyrinthe 3  est au point : [1, 0]

In [32]: runfile('C:/Users/quent/Documents/Cours/2
quent/Documents/Cours/2e année/Python/Projet')
Le départ du labyrinthe 4  est au point : [1, 0]
```

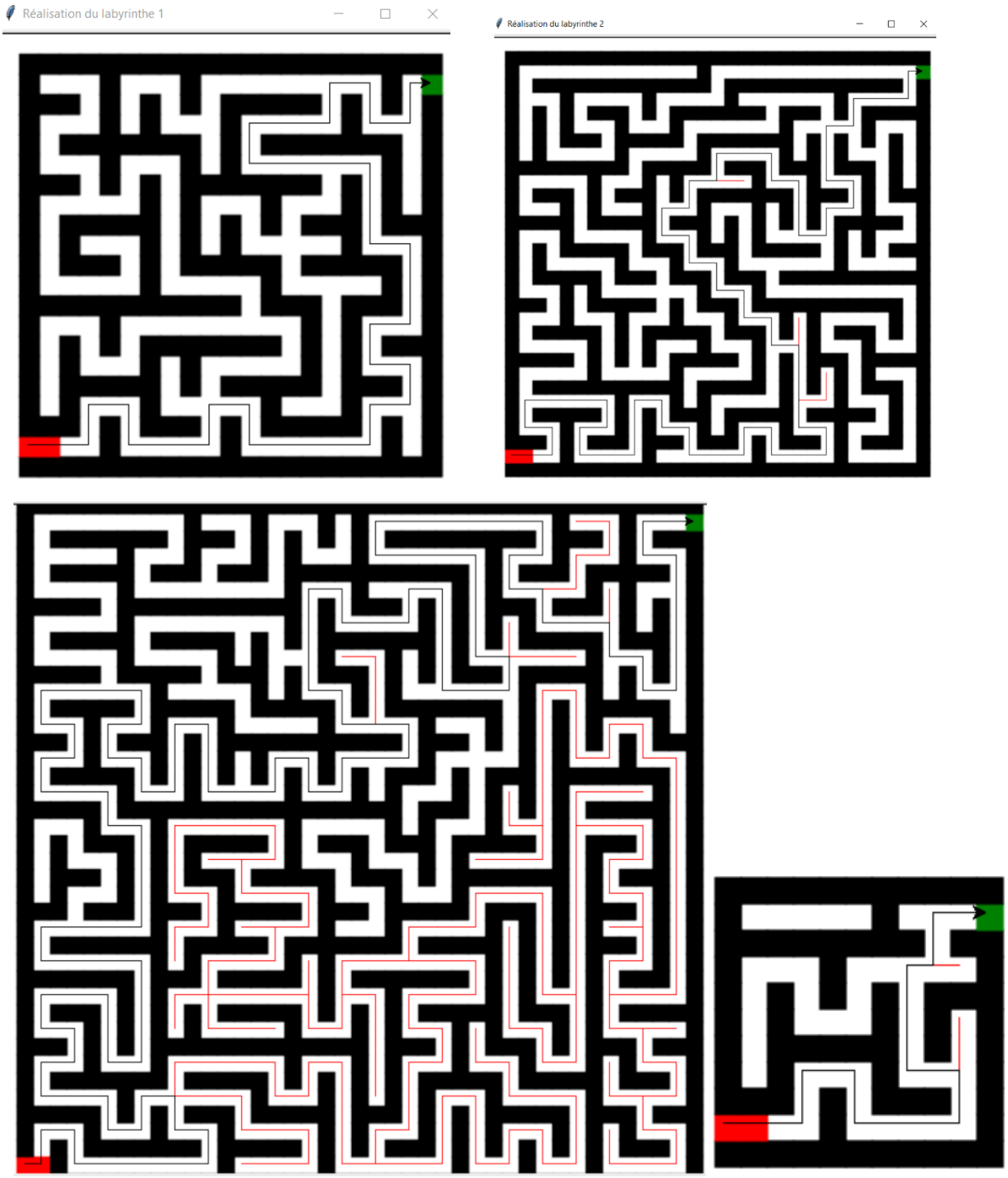
Or lorsque l'on compare au fichier texte on obtient bien les mêmes coordonnées. (la première coordonnée est la ligne et la seconde la colonne)

On peut ensuite vérifier que la fenêtre graphique s'ouvre et s'affiche correctement :



On voit donc que la fenêtre s'ouvre, le titre est correcte, l'image de fond correspond et est à la bonne taille (elle n'est pas coupée).

On peut ensuite vérifier le programme en intégralité, c'est-à-dire que le labyrinthe ce résout correctement :

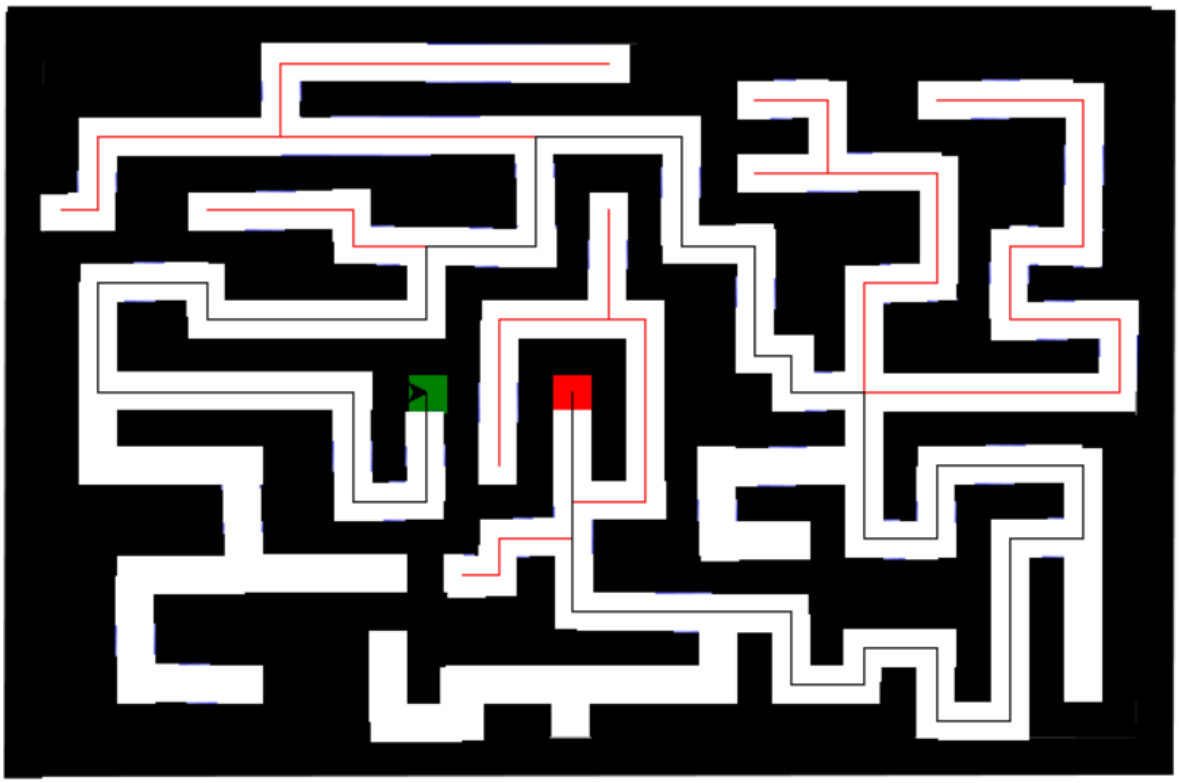


Le programme fonctionne donc correctement, il résout les labyrinthes, colore les impasses en rouge, …

Pour pousser le programme j'ai aussi créé un labyrinthe qui n'est pas carré et où le départ et l'arrivé ne sont pas à la même place que pour les autres labyrinthes.

```
In [33]: runfile('C:/Users/quent/Documents/Cours/2e
quent/Documents/Cours/2e année/Python/Projet')
Le départ du labyrinthe 5  est au point : [10, 15]
```

Réalisation du labyrinthe 5

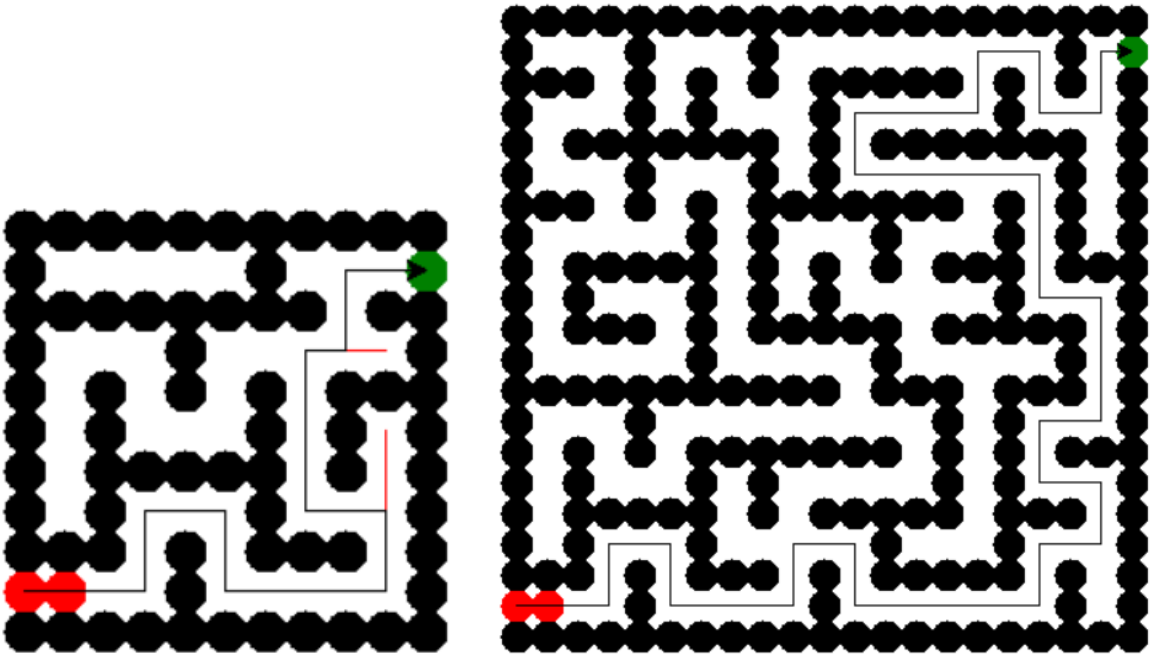


Maze5 - Bloc-notes

Fichier Edition Format Affichage Aide

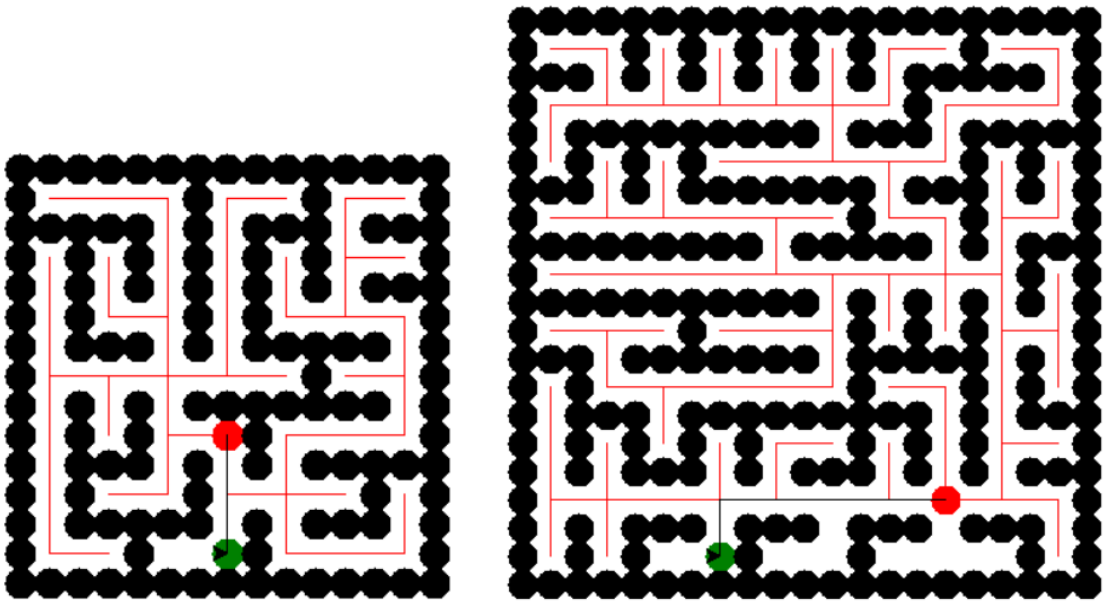
```
00000000000000000000000000000000
00000000002220020000000002220000
00000002002022222222022202020000
00022222002000000002020222020000
00020000000000022222220000020000
00022222222022020000000000020000
00000020000002220002220222022200
00000020022200022202000202000200
00222220020202020202222202222200
00200000020202020200000200000000
00222222220302010200022222222220
00200000000002000200220200000020
00200222222202222200200200022220
00222200000200002000200222020000
00000000022222202022200002022200
02200222220000202020000002000200
00200000000000200020222222000200
00222222222222222220002000000200
00000002000000000000222002222200
00000002222222222000000000000000
00000000000000000000000000000000
```

On peut maintenant tester la partie qui nous permet de générer le labyrinthe, pour ça j'utilise le programme de la deuxième partie :



On voit que le programme fonctionne correctement, que les labyrinthes sont bien générés (à gauche le 4e et à droite le 1e) et résolus.

Maintenant on peut tester la génération automatique de labyrinthe avec le 3e code :



On peut voir que le programme ne marche pas totalement car le labyrinthe ne peut pas être résolus. Il faut juste fusionner les deux chemin, mais je ne pense pas avoir le temps.

4) Conclusion

Pour conclure j'ai réussi à coder un programme capable de répondre à la partie 1 et 2, c'est-à-dire un programme capable d'afficher et de résoudre un labyrinthe à partir d'un fichier texte. J'ai aussi réussi un coder un programme qui répond à la problématique de la partie 3, c'est-à-dire générer aléatoirement un labyrinthe, le créer et le résoudre. Il y a cependant quelques points qui peuvent être amélioré comme le problème Terminator qui apparait 1 coup sur 2, la demande utilisateur du labyrinthe à résoudre qui n'est pas optimal et qui peut entrainer un bug en cas de mauvaise entré de l'utilisateur (j'ai préféré me concentrer sur la partie 3), le temps de génération du labyrinthe qui est assez long. On pourrait aussi apporter quelques améliorations comme plusieurs départs possibles ou encore le curseur qui pointe toujours vers sa destination (rotation).