Terminale - Spécialité NSI

Projet - Générateur de labyrinthes

Présentation du problème

Un labyrinthe est un réseau de chemins, d'embranchements, dont on peine généralement à trouver la sortie. Un labyrinthe peut donc constituer un jeu dont le but est de trouver un chemin allant du départ du labyrinthe jusqu'à sa sortie. Un labyrinthe **parfait** est un labyrinthe où il n'existe qu'un seul chemin depuis l'entrée vers la sortie et où chaque cellule (pièce) est reliée à toutes les autres, et ce, de manière unique.

A première vue, il pourrait sembler compliqué de construire un labyrinthe parfait (de n'importe quelle taille). Pourtant, il existe une méthode très simple, utilisant **les arbres binaires**, permettant d'automatiser cette tâche en générant un labyrinthe parfait de n'importe quelle taille aléatoirement.

Le but de ce projet va donc être, à partir de la présentation de cette méthode, de coder, en python, un générateur aléatoire de labyrinthes qu'on pourra également afficher et résoudre automatiquement en exploitant les arbres binaires avec lesquels vous avez déjà travaillé.

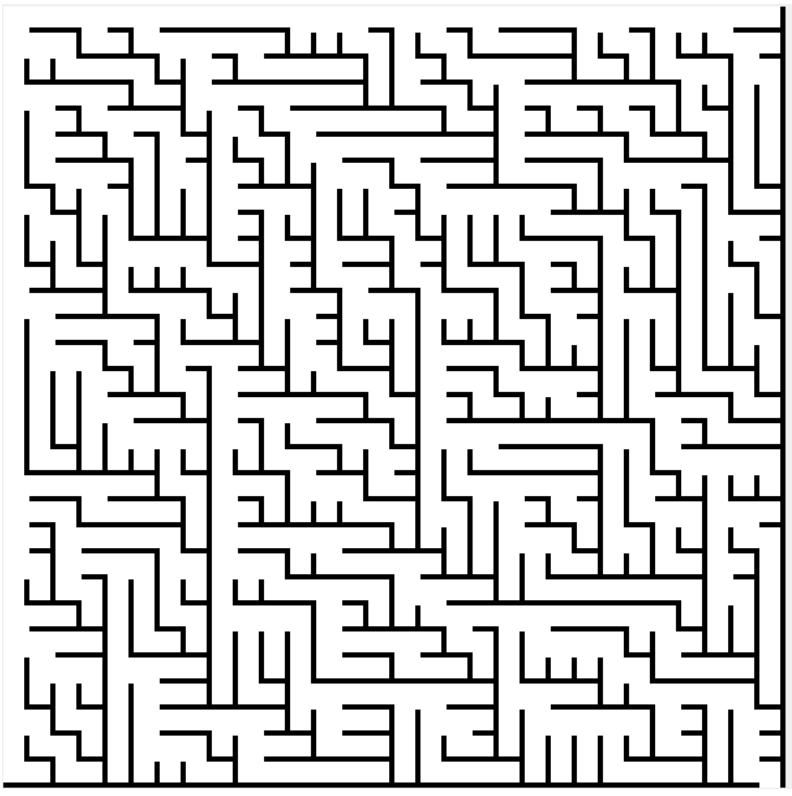

FIGURE 1 – Labyrinthe 30x30

Méthode

De manière générale

Sans encore parler d'arbre, la méthode pour construire le labyrinthe est la suivante :

- On a un labyrinthe de dimensions H x L (Hauteur, Largeur). On a donc une grille qui contient (H * L) cases à traiter (pièces du labyrinthe).
- Chaque cases a des coordonnées (x,y) dans le labyrinthe (la case en haut à gauche (0,0), la deuxième cases de la ligne du haut (1,0), la dernière case en bas à droite (H-1, L-1), etc...
- Au départ, il n'existe aucun passage entre les pièces (on peut imaginer quatre murs tout autour de la case).
- On traite chaque case de la grille une à une :
  - On choisit **aléatoirement** une des cases voisines : à gauche de la case, ou au-dessus, et on ouvre un passage avec (on brise le mur entre la case et la case voisine choisie).
  - Si la case n'a pas de voisin à gauche, on choisit forcément le voisin du haut (toutes les cases situées sur la première colonne de la grille)
  - Si la case n'a pas de voisin en haut, on choisit forcément le voisin de gauche (toutes les cases situées sur la première ligne)
  - Si la case n'a pas de voisin à gauche ou en haut, on ne fait rien (c'est la case (0,0), le départ du labyrinthe)
- Quand toutes les cases ont été traitées, on a construit un labyrinthe parfait !

Vous pouvez essayer sur papier pour comprendre la méthode (sur un petit labyrinthe).

Avec les arbres

Vous l'aurez peut-être remarqué, mais un labyrinthe parfait généré de cette manière est une structure hiérarchique. En effet, on part du point de départ et on ne revient jamais en arrière. Chaque case peut avoir au maximum deux passages ouverts vers une autre case. On peut donc modéliser cela avec un arbre binaire : Chaque case est un noeud , la racine est la case de départ (0,0), un fils à gauche signifie un passage ouvert en bas, un fils à droite signifie un passage ouvert à droite.

Voici la méthode pour générer le labyrinthe avec un arbre binaire :

- On crée une liste à deux dimensions (c'est-à-dire, une liste qui contient une liste) qui contiendra tous les noeuds binaires (cases) du labyrinthe, indexées par leurs coordonnées. La liste contiendra donc un nombre de liste correspondant à la longueur du labyrinthe, et ces listes contiendront un nombre de Noeuds correspondant à la largeur du labyrinthe. Dans cette liste "l", l[x][y] donne la case de coordonnées (x,y)
- Initialement, la valeur de chaque NoeudBinaire correspond aussi à ses coordonnées sur la grille (stockées dans un tuple par exemple), et, ils n'ont aucun fils. Vous pouvez imaginer un "espace" de noeuds sans aucun lien, le but va être de piocher dans cet espace afin de créer les relations parent / fils progressivement.

— On parcourt la liste des noeuds (deux boucles, pour varier sur la longueur et la largeur). Pour chaque noeud binaire :
  — On récupère, dans la liste, les noeuds qui correspondent à la case située au-dessus et à gauche du noeud (donc, pour un noeud de coordonnées (x,y) on récupère le noeud situé à (x,y-1) et (x-1,y)) **s'ils existent** (certaines cases n'ont pas de voisins à gauche ou au-dessus).
  — Si on a récupéré deux voisins, on choisit un des deux aléatoirement, sinon, S'il n'y en a qu'un, on choisit celui qui reste. S'il n'y en a pas, on en choisit donc aucun.
  — Si c'est le voisin du dessus qui a été choisi, son fils gauche (qui était nul) devient le noeud traité dans cette itération.
  — Si c'est le voisin de gauche qui a été choisi, son fils droit (qui était nul) devient le noeud traité dans cette itération.
— Quand on sort des boucles, on a donc obtenu un arbre binaire qui stocke le labyrinthe (avec pour racine le noeud qui est en position (0,0) dans la liste).
— On garde une référence sur l'entrée du labyrinthe (premier noeud de la liste) mais on ne garde pas la liste. Elle est inutile, car on obtient déjà tout le labyrinthe à partir du noeud racine.

Avec cet arbre, on pourra par la suite dessiner le labyrinthe aisément. Si dans l'arbre, un noeud n'a pas de fils à gauche cela veut dire qu'il y a un mur en bas, s'il n'a pas de fils à droite, cela veut dire qu'il y a un mur à droite.

Développement

Le projet est divisé en différentes parties, avec une difficulté croissante. Vous devez traiter les parties dans l'ordre.

Récupérez le fichier **Labyrinthe.py** (et les fichiers annexes) qui contient le squelette de code que vous devrez compléter pour réaliser le projet.

Vous pouvez apporter des modifications à ce fichier comme bon vous semble, mais il est conseillé de conserver ce qui est déjà en place.

Partie 1 : Génération de labyrinthes

Le but de cette première partie est d'appliquer la méthode que nous avons vu précédemment pour générer un labyrinthe en utilisant des arbres binaires.

Vous devez compléter la méthode **genererLabyrinthe(self)** de la classe Labyrinthe.
Cette méthode contient déjà une liste "cases" qui contient la liste des noeuds binaires déjà initialisés. Vous devez parcourir cette liste et créer les liens entre les noeuds comme décrit précédemment. A la fin, la méthode crée deux attributs "entree" (qui correspond au noeud racine de l'arbre) et "sortie", afin de garder une référence vers ces deux cases.

Rappel : La bibliothèque **random** permet de gérer l'aléatoire (génération de nombres aléatoires, choix aléatoires...). Il peut être utile d'en consulter la documentation en ligne.

A la suite de la classe, vous devez écrire un programme permettant de tester votre code. Le but est d'afficher l'arbre généré, avec la fonction **afficher_arbre**.
On peut imaginer cela de façon interactive, où le programme demande à l'utilisateur la longueur et la largeur du labyrinthe souhaité, l'initialise puis affiche l'arbre correspondant. Pour rappel, la racine de l'arbre correspond, après génération, à l'attribut "entree".
Dans vos tests, ne dépassez pas les dimensions 15x15, car l'arbre ne sera pas affichable au-delà.

Partie 2 : Affichage du labyrinthe

Le but de cette seconde partie est d'afficher le labyrinthe généré.

Vous devez compléter la méthode **afficher(self)** de la classe Labyrinthe.
Le module **AfficheurLabyrinthe** est à votre disposition pour vous aider. Ouvrez-le pour en lire la **documentation**.
Le but est de créer, successivement, des objets **CaseGraphiqueLabyrinthe** pour chaque noeud de l'arbre et d'afficher ou non le mur de droite ou du bas (des méthodes de cette classe permettent de gérer cela). Dès que la case est correctement configurée, il faut l'ajouter à la variable **"labyGraphique"** en spécifiant ses coordonnées dans la grille, via la bonne méthode.
Pour rappel, chaque Noeud contient, comme valeur, un tuple représentant ses coordonnées. Pour une case donnée, il y a un mur à droite si le noeud associé n'a pas de fils à droite, et il y a un mur en bas si le noeud associé n'a pas de fils à gauche.

Afin de parcourir l'arbre, il est fortement recommandé d'utiliser **un parcours en largeur** (donc, en utilisant la classe File). Le noeud "initial" placé dans la file sera **self.entree**.

A la suite de la classe, vous devez écrire un programme permettant de tester votre code. Le but est d'afficher le labyrinthe dans une fenêtre (comme l'image au début du document de projet) via la méthode **afficher(self)** que vous venez de coder.
On peut imaginer cela de façon interactive, où le programme demande à l'utilisateur la longueur et la largeur du labyrinthe souhaité, l'initialise puis affiche le labyrinthe.
Il n'y a pas de limite de taille pour votre labyrinthe ! (enfin, au-delà de 1000x1000, cela risque de ramer...)

Partie 3 : Résolution du labyrinthe

Le but de cette troisième partie est de résoudre le labyrinthe en calculant le chemin de l'entrée vers la sortie et en l'affichant (en rouge) sur le labyrinthe.

Vous devez compléter la méthode **solution_recursif(self, debut, fin, chemin)** de la classe Labyrinthe.
Cette méthode est **récursive** et permet de trouer un chemin allant de la case représentée par le noeud "début" vers la case allant représentée par le noeud "fin". La variable "chemin" est passée lors des appels récursifs pour garder en mémoire le chemin parcouru jusqu'ici (liste des cases parcourues). C'est cette variable qui sera retournée à la fin. Elle devra contenir la liste des coordonnées successives des cases du chemin sous forme de tuples. (Par exemple [(0,0), (1,0), (2,0), (2,1), etc...]).

Le code de cette méthode **est très proche** de celui utilisé pour trouver un **chemin dans un graphe** (car un arbre est un graphe !). N'hésitez pas à vous en inspirer.

Vous n'avez pas à modifier le code de la fonction **solution(self)**. Cette méthode permet d'appeler la fonction **solution_recursif(self, debut, fin, chemin)** en demandant de chercher un chemin allant du début du labyrinthe jusqu'à la sortie. Si votre code fonctionne, l'appel de cette méthode devrait vous donner le chemin solution du labyrinthe.

Enfin, afin d'afficher la solution sur le labyrinthe, créez une nouvelle méthode similaire à **afficher(self)** ou modifiez cette fonction directement. Le but est de colorier les cases en rouge (une méthode de **CaseGraphiqueLabyrinthe** permet de faire cela) si celles-ci appartiennent au chemin solution..

Vous devriez obtenir quelque chose comme ça :

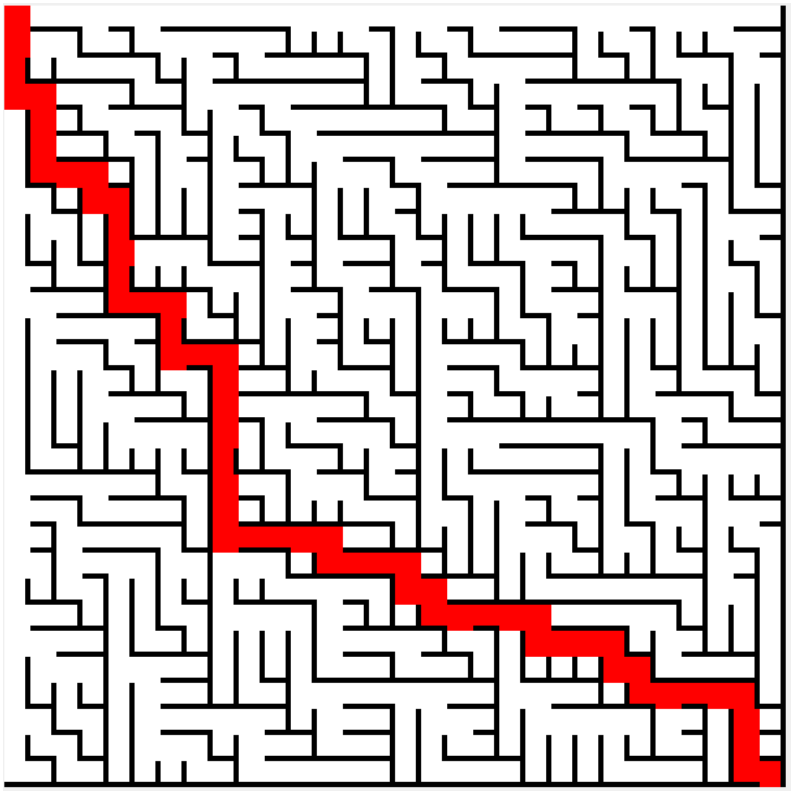

FIGURE 2 – Solution du labyrinthe 30x30

A la suite de la classe, vous devez écrire un programme permettant de tester votre code. Le but est d'afficher le labyrinthe dans une fenêtre (comme l'image au début du document de projet) via la méthode **afficher(self)** et également **la solution** (soir directement sur la fenêtre, soit après l'avoir fermée...) On peut imaginer cela de façon interactive, où le programme demande à l'utilisateur s'il veut voir la solution ou non après avoir affiché le labyrinthe initial, et ouvre une fenêtre avec la solution s'il dit oui.

Partie 4 : Bonus

Vous êtes libre sur cette partie ! Vous pouvez apporter toutes les évolutions que vous souhaitez (notamment, pour l'affichage par exemple, l'interface graphique...) et améliorer votre programme. On pourrait, par exemple, imaginer un système qui place l'entrée et la sortie du labyrinthe aléatoirement (pas seulement en haut à gauche et en bas à droite), un bouton qui permet de faire apparaître puis disparaître la solution sur la même fenêtre, etc...

Rendus

Vous devrez rendre (via pronote) :

- Le (ou les) fichier(s) python représentant la version la plus aboutie de votre projet.
- Un rapport de projet (par groupe) contenant :
  - Une partie présentant le déroulement du projet (un historique) qu'est-ce qui a été fait, dans quel ordre, comment vous avez travaillé (avec quels outils...).
  - La liste "fonctionnalités" de votre projet final (qu'est-ce que le projet permet de faire, en gros, quelles parties vous avez traitées).
  - Les éventuelles difficultés rencontrées et comment vous les avez résolues.
  - Un planning avec **la répartition du travail** (qui a fait quoi et quand).

Critères d'évaluation

Vous serez évalué sur différents points :

- Votre programme fonctionne (se lance sans erreurs).
- Respect du cahier des charges énoncé dans les différentes parties.
- Le niveau d'avancement du projet (les parties implémentées (et leur fonctionnement) / les ajouts bonus).
- Le projet fonctionne et permet de générer un labyrinthe sous forme d'arbre binaire.
- Le code contient des tests.
- Le projet ne contient pas (ou peu) de bugs.
- Le code utilise correctement la structure d'arbre binaire (pour la génération, le parcours, etc...).
- Les notions liées aux structures de données (arbres, file) et à la programmation objet sont bien exploitées.
- Exploitation des bibliothèques (random et AfficheurLabyrinthe)
- Les noms des variables sont significatives (on comprend ce que stocke la variable en lisant son nom).
- Le code est commenté et documenté par endroits, pour expliquer les lignes importantes et le fonctionnement du projet.
- Le rapport est bien présenté.
- Le rapport contient tous les points demandés et est suffisamment détaillé.
- Le travail dans le projet a été régulier et sérieux.
- Le travail a été réparti équitablement entre les membres du groupe.