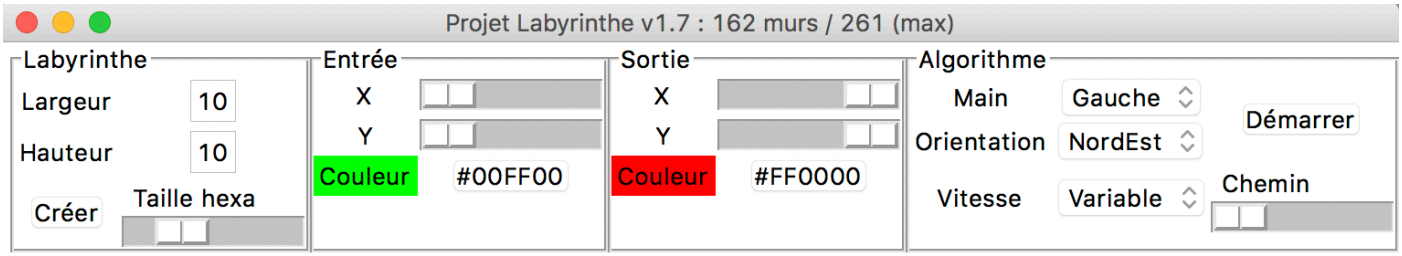
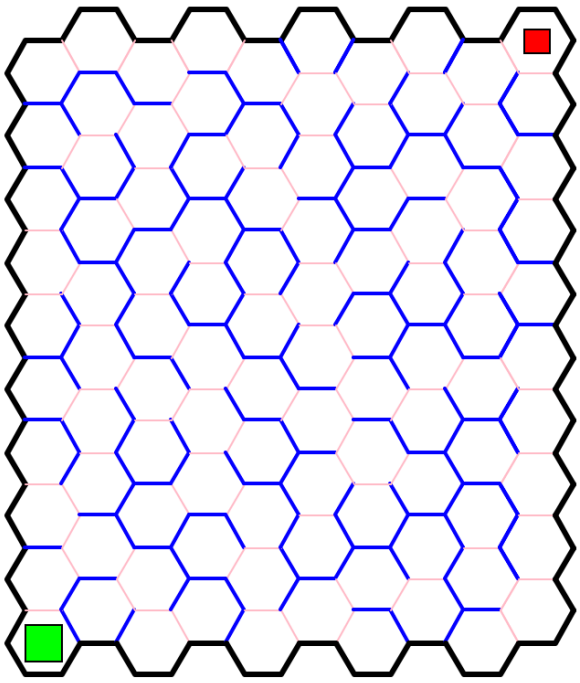
TERMINALE S ISN
Lucas Schneider
Maxence Decourtis
Maé de Montalier

PROJET ISN : LABYROBOT





I – PRESENTATION DU PROJET

Notre équipe LabyRobot s'est lancée dans la réalisation d'un projet en apparence basique mais qui en réalité demande de nombreuses connaissances. Nous sommes fiers de vous présenter notre projet final : un générateur automatique de labyrinthe parfait composé d'îlots avec un robot capable de trouver à chaque fois la sortie.

Pourquoi ce projet ?

Tout d'abord, nous avons réfléchi ensemble à propos de notre futur projet. Assez rapidement, l'idée de la programmation d'un labyrinthe généré aléatoirement et automatiquement avec un robot capable de la résoudre nous est apparue comme la meilleure idée. Il nous a fallu alors peu de temps pour déterminer ce projet et ainsi, nous lancer dans sa réalisation.

Nous avons choisi ce projet pour de multiples raisons. Premièrement, étant tous les trois des étudiants en Terminale Scientifique, les algorithmes sont des suites finis d'instructions avec lesquels nous avons déjà travaillé que ce soit en mathématique ou hors cadre scolaire. Il se trouve que la programmation du projet met en jeu plusieurs algorithmes. Connaissant alors déjà un peu le fonctionnement de ces derniers, nous avons désiré en savoir plus pour pouvoir les mettre en pratique dans notre projet. La curiosité et l'excitation d'apprendre de nouvelles choses nous ont alors poussé à nous lancer dans ce travail.

Dans un second temps, nous avions auparavant lu sur internet des articles mentionnant ce type de projet. L'idée de base nous semblait intéressante mais nous avons vraiment choisi de créer ce programme car nous voulions apporter quelque chose de plus. Nous avons alors réalisé un générateur de labyrinthe qui comporte des hexagones et non des simples cases carrées. Cependant, lors des premières recherches sur le code nécessaire à la création de ce labyrinthe, nous nous sommes rendus compte que le programme était plus dur que dans un labyrinthe classique. Mais attiré par la difficulté et l'envie de se surpasser, nous nous sommes acharnés pour que notre idée puisse devenir réalité.

Dans un dernier temps, ce projet nous parut très intéressant à réaliser car il allait nous octroyer de nombreuses connaissances de programmation qui pourront être utilisées à nouveau dans notre future carrière professionnelle. La quantité de travail que nécessitait ce code ne nous a pas effrayé car nous étions très déterminés et le résultat final en valait le coup. C'était alors pour nous un vrai apport autant personnel que professionnel car nous savions que le projet n'allait pas être simple et qu'ainsi, la cohésion au sein de notre groupe ne pouvait être que renforcé. L'aspect répartition des tâches et travail de groupe nous a beaucoup plu et à travers ce travail, nous savions qu'ils seraient essentiels.

Ainsi, les enjeux du projet étaient multiples :

- Trouver l'algorithme de résolution le plus simple et le plus rapide pour le robot.
- Comprendre et utiliser de nouvelles fonctions de programmation.
- Obtenir un "jeu" facile à prendre en main par l'utilisateur au travers d'une interface clair.
- Générer un labyrinthe aléatoirement à chaque fois
- Se répartir équitablement les tâches et travailler ensemble

1. Positionnement du projet par rapport à des solutions existentielles

Notre projet étant un labyrinthe parfait hexagonal automatiquement généré avec un avec un algorithme de résolution, nous avons recherché si des projets tels que le nôtre avait déjà été réalisé auparavant. Après quelques recherches nous avons surtout trouvé des labyrinthes carrés parfaits générés automatiquement, en effet il existe de nombreux algorithmes permettant la génération de labyrinthes parfaits, et de nombreux algorithmes de résolution. Cependant nous n'avons pas trouvé nous n'avons pas trouvé de tels algorithmes pour des labyrinthes à bases hexagonales.

2. Cahier des charges de l'équipe

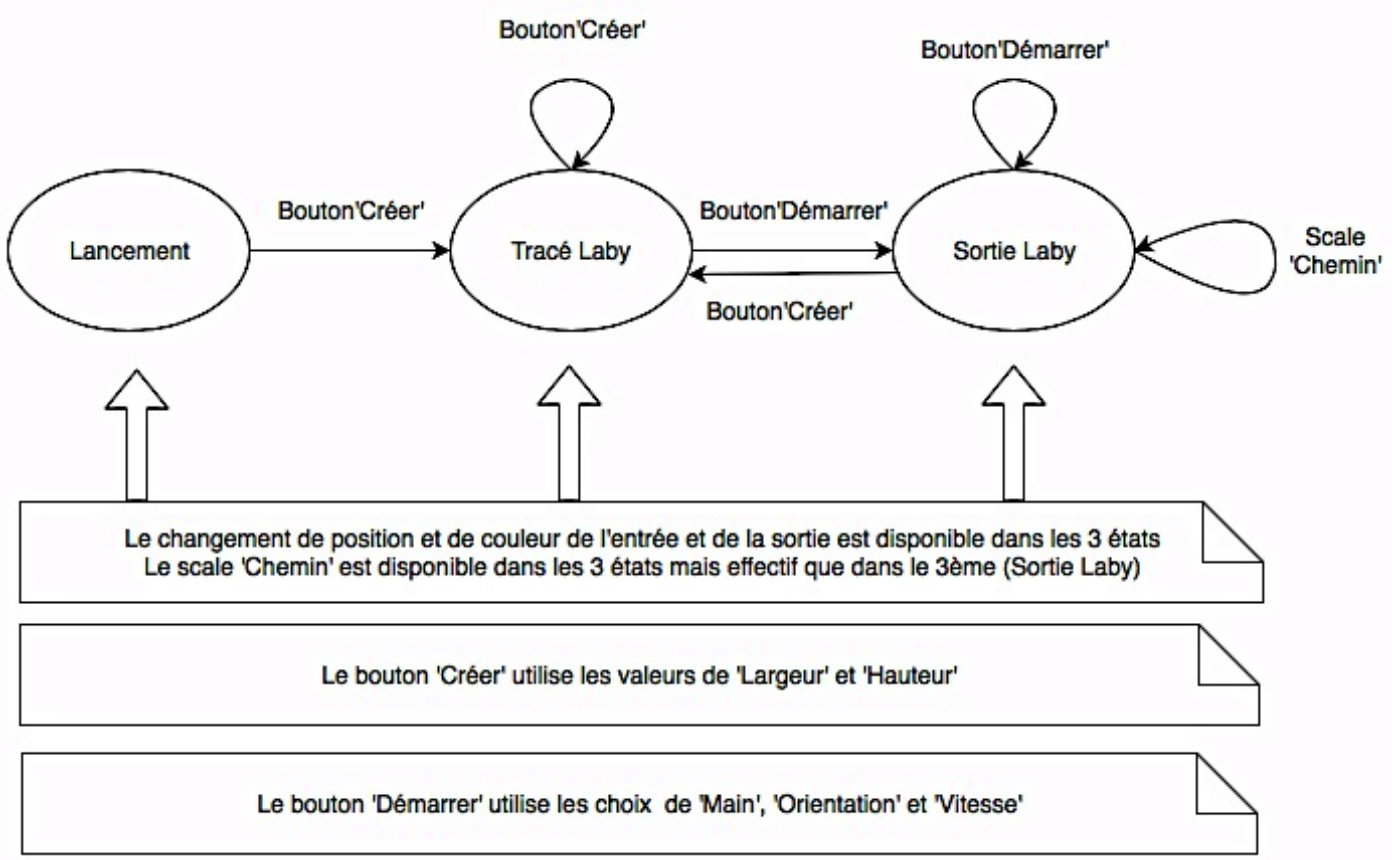
- génération automatique de labyrinthe / parfait
- cases hexagonales
- taille configurable
- définition de la position entrée et sortie
- interface utilisateur conviviale (graphique)
- pouvoir rejouer le tracé
- visualisation du déplacement (tracé pas pratique ca marche avec les numéros)
- algorithme de suivi de mur (deux options main gauche et main droite)

3. Moyens mis en œuvre (langage / matériel)

Notre programme nécessite l'utilisation de Turtle. En effet, ce module graphique permet de, comme son nom l'indique, déplacer une tortue sur un écran. Ce dernier fonctionne avec le langage de programmation Python. On a alors pu utiliser certaines de nos connaissances sur la langage pour les appliquer dans le code. De plus, nous avons eu recours à TKinter pour la création de l'interface du labyrinthe. Encore une fois, c'est le langage Python qui est utilisé car c'est un module de base intégré dans Python. Ainsi, la totalité de notre code repose sur le langage Python.

4. Structure globale du projet

Il y a 3 états dans lesquels l'utilisateur peut réaliser des actions différentes : l'état 'Lancement' obtenu au lancement du programme, l'état 'Tracé Laby' et l'état 'Sortie Laby' :



`

5. Tableau présentant les tâches et les avancées de notre équipe

	AVANCÉES et TÂCHES		
	LUCAS	MAXENCE	MAE
12/01/18	Choix sujet : labyrinthe parfait généré automatiquement avec un robot le qui le résout par lui même. Faire des recherches sur labyrinthes car sujet complexe et ambitieux.		
19/01/18	Idée d'un labyrinthe avec cellules en diagonales pour originalités. Trouver un programme pour l'interface graphique.		

<table>
<tr><td></td><td colspan="3"></td></tr>
<tr><td>26/01/18</td><td colspan="3">Il faut s'occuper de :
- L'interface graphique
- Comment générer un labyrinthe ?
- Quels outils informatiques utiliser ?
- Modélisation du robot (affichage son chemin)
- Quelle case d'entrée ? Quelle case de sortie ?
- Algorithme du robot → aléatoire ou plusieurs mode de sortie

Test pour algorithme du robot : peut-il sortir en se collant au coté gauche ou au côté droit ? → vrai à gauche et vrai à droite</td></tr>
<tr><td rowspan="2">02/02/18</td><td colspan="3">Test des labyrinthes parfait et imparfait.
Mise en commun des premières lignes de programmation et des résultats des tâches → Utilisation du logiciel Turtle pour la génération du labyrinthe, et de TK inter pour l'interface graphique.
Répartition globale des tâches :</td></tr>
<tr><td>Trouver comment générer le labyrinthe avec Turtle.</td><td>Trouver les ressources nécessaires pour la réalisation du projet.</td><td>Comprendre les techniques de TK inter pour l'interface.</td></tr>
<tr><td>09/02/18</td><td colspan="3">Cours annulé, Skype pour parler des avancements et des tâches à faire.</td></tr>
<tr><td>16/02/18</td><td>Algorithme du labyrinthe terminé par Lucas.
Et mise en forme de programme.</td><td>Avancer sur le déplacement du robot.
Savoir ce qu'il y manque dans le programme.</td><td>Début de l'interface graphique fait.</td></tr>
<tr><td rowspan="2">Pendant les vacances de février</td><td>Codage de la partie génération du labyrinthe en hexagone avec l'initialisation de Turtle</td><td colspan="2">Trouver les idées et faire des recherches pour le code de la génération du labyrinthe → idées ensemble, Lucas lui qui codait le plus + petite aide de son père, ingénieur en informatique.</td></tr>
<tr><td colspan="3">On s'est vus mardi, vendredi et samedi de la première semaine, Maé en Skype. Traçage de la grille d'hexagone centrée sur la fenêtre Turtle.
Réglages de quelques problèmes lorsque le labyrinthe contenait un nombre pair ou impair d'hexagones en largeur au niveau des murs extérieurs.
Aide avec fonction help.turtle, → beaucoup de nouvelles idées.</td></tr>
<tr><td>09/03/18</td><td>Commencer l'algorithme du robot avec l'algorithme de Pledge : sortir du labyrinthe en tournant tout le temps à gauche ou tout le temps à droite.</td><td>Faire les finitions du codage du programme du labyrinthe.</td><td>Continuer de coder l'interface graphique avec Lucas.</td></tr>
</table>

<table>
<tr><td>16/03/18</td><td colspan="2">Début programmation de l'algorithme de Pledge.
Continuer programmation génération automatique du labyrinthe.
Continuer à avancer maintenant tous ensemble sur l'interface graphique.</td></tr>
<tr><td rowspan="2">23/03/18</td><td colspan="2">Pas cours d'ISN, réunion chez Maxence.
Abandon de l'algorithme de Pledge trop compliqué, autre algorithme car un seul type de labyrinthe : le labyrinthe parfait.</td></tr>
<tr><td>Utilisation et programmation d'un nouvel algorithme « suivre le mur », le robot va seulement suivre soit le mur de gauche ou soit le mur de droite plus simple à coder.
Nouvelle idée → slider : suivre résolution laby</td><td>Avancée au niveau de l'interface graphique (la doc de TK inter sur internet très pratique)</td></tr>
<tr><td>30-31/03/18</td><td colspan="2">Pas cours d'ISN, réunion chez Lucas pendant le week-end pour réalisation de l'interface graphique. Petite aide du père de Lucas l'organiser et comprendre les fonctionnalités de chaque fonction.</td></tr>
<tr><td rowspan="2">06/04/18</td><td colspan="2">Mise au point avec Mr Jules, → il faut avancer vite.
Idée d'un fichier audio.</td></tr>
<tr><td>Réglage d'un défaut.
Avancer codage algorithme résolution.
Interface graphique.</td><td>Grande avancée dans l'interface graphique de la page du labyrinthe, nouvelles idées et nouvelles mises en formes.</td></tr>
<tr><td>13/04/18</td><td colspan="2">Mise au point des avancées de la semaine.
Objectif : terminer code pour rentrée pour finaliser petits détails.</td></tr>
<tr><td rowspan="2">Pendant les vacances de Pâques</td><td colspan="2">Finir l'interface et l'algorithme de programmation du robot.</td></tr>
<tr><td>Programmer l'algorithme de gauche et droite.
Test vérification de l'algorithme « suivi du mur.
Faille : robot bloqué en main gauche.</td><td>Programmer slider pour chemin robot (aide de Lucas) + option pour l'utilisateur : dessiner parois labyrinthe avec couleur.
Programmer boutons slider et boutons main droite et main gauche.</td></tr>
<tr><td>04/05/18</td><td colspan="2">Retour des vacances → quasiment terminé juste détails. Accords sur modifs à faire pour finaliser. Répartition tâches pour rédaction dossier.
Changer nom certaines fonctions car pas explicites. Ajout d'une fonction pour changer vitesse tracé chemin du robot.</td></tr>
</table>

Dossier ISN

Quelques fonctions :

Voilà quelques exemples de fonctions sur lesquelles j'ai travaillé au cours de mon projet ISN. Ici sont présentés le fonctionnement global des fonctions, à quoi elles servent et leurs différentes entrées et sorties.

fonction init() :

Entrée :
- -Arguments : largeur et hauteur du labyrinthe
- -Variables globales lues : voir initTurtle() et initDonnees()

Sorties :
- -Return : non
- -Variables globales écrites : voir initTurtle() et initDonnees()

La fonction appelle initTurtle() pour initialiser le module 'Turtle' puis initDonnees(largeur, hauteur) pour dimensionner et initialiser également les variables globales TabLaby, Largeur, Hauteur, Sortie, ainsi que initCentrage() qui initialise les variables globales DemiLargeurTotale et DemiHauteurTotale.

fonction traceHexa() :

Entrée :
- -Arguments : abscisse et ordonné en pixels du centre de l'hexagone, types de mur des côtés
- -Variables globales lues : DemiTailleHexa

Sorties :
- -Return : non
- -Variables globales écrites : non

Cette fonction trace un hexagone en fonction de son centre et des types de ses côtés.
Elle utilise les paramètres définissant la couleur et l'épaisseur en fonction du type (EPAISSEUR_LABY et COULEUR_LABY).

fonction centreHexa() :

Entrée :
- -Arguments : index en largeur (i) et index en hauteur (j) de l'hexagone choisi
- -Variables globales lues : DemiTailleHexa, DemiLargeurTotale, DemiHauteurTotale

Sorties :
- -Return : coordonnées en pixel du centre de l'hexagone
- -Variables globales écrites : non

Celle-ci calcule les coordonnées en pixels correspondant à l'hexagone (i, j), dans le système de coordonnées de Turtle.

fonction creeLaby() :

Entrée :
- -Arguments : aucun
- -Variables globales lues : Largeur, Hauteur, TabLaby

Sorties :
- -Return : non
- -Variables globales écrites : MursInit, MursMax, MursPlus, MursMoins

Cette fonction est un algorithme servant à générer un labyrinthe parfait :

Description général de l'algorithme :
Au début de l'algorithme créant le labyrinthe tous les murs sont de type internes hormis ceux des côtés de la grille. On va commencer par affecter une valeur croissante à chaque cellule (dans TabLaby[i][j][6]), et tant que les valeurs des

cellules ne sont pas toutes nulles, on applique les étapes suivantes :

- supprimer de manière aléatoire un mur interne (qui existe pour optimiser le temps de création) séparant deux hexagones de valeurs différentes (très importants pour ne pas détruire des murs qu'il faut garder)
- affecter la valeur minimale des deux cellules concernées à toutes les cellules ayant la valeur maximale Au début de l'algorithme créant le labyrinthe il y a des murs internes partout

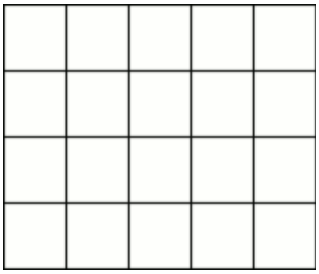
Puis, on calcul le temps nécessaire : fonction time (du module time), enregistre le temps à l'entrée de la routine et à la fin puis on fait la différence avec le temps préalablement enregistré arrondi au centième de seconde :
tps = round(time.time() - tps, 2)

Description détaillée :
Au commencement, on va affecter une valeur croissante (en partant de zéro) à chaque hexagone (i, j) en utilisant le tableau "TabLaby[i][j][6]" et une double boucle imbriquée suivant la largeur et la hauteur du labyrinthe. On initialise ensuite le nombre max de murs et le nombre de murs (MursMax) quand le labyrinthe parfait est créé (MursInit) et on initialise le nombre de mur pouvant être détruits (nbMursCibles) — équivalents à deux fois le nombre de MursMax car chaque mur interne est compté deux fois —.
On utilise après une boucle "while" : tant que les valeurs des cellules ne sont pas toutes nulles — en effet, une fois que le labyrinthe parfait est créé il y a un chemin qui passe par toutes les cellules donc elles sont toutes jointes et donc ont toutes la même valeur (zéro) —.
A l'aide de la fonction random.randrange, on tire au sort un numéro de mur parmi les nbMursCibles murs cibles. On recherche alors le mur cible dans la liste des murs (on reboucle dans largeur, hauteur et 6) et on regarde si c'est un mur interne. Enfin, on regarde le numéro de la cellule voisine et si les deux numéros sont différents, les deux cellules ne sont pas déjà reliées. Sinon ce n'est pas un mur cible.
C'est nbMurs la variable devant être égale à nbMursCible (on teste tous les murs en i, j, k et si c'est un mur cible nbMurs incrémente).
Une fois qu'on est sur le mur cible (nbMurs = nbMursCibles) on a trouvé le mur à détruire :

- on affecte ce mur ainsi que son voisin à la valeur ouverture
- on stocke les valeurs minimales et maximale des deux cellules
- on affecte la plus petite à toutes les valeurs ayant la plus grande (en utilisant deux indices m et n pas i et j qui sont déjà utilisés)

On doit alors recalculer le nombre de murs cibles car il a changé :
→ on parcours tous les murs (Largeur, Hauteur, 6)
→ on regarde si c'est un mur interne et si sa valeur est différente de son voisin
A ce moment, la fonction contient six boucles imbriquées !

Voici un petit gif animé très sympathique qui m'a permis de comprendre le fonctionnement de l'algorithme générant un labyrinthe parfait. Il y a cependant une erreur dans cette animation, saurez-vous la trouver ?



fonction voisin() :
Entrée :
-Arguments : index en largeur (i) et index en hauteur (j) et l'index du côté (k)
-Variables globales lues : Largeur
Sorties :
-Return : index en largeur, en hauteur et du côté pour la paroi voisine (-1 si inexistante)
-Variables globales écrites : non

Cette fonction nous a permis de calculer le triplet définissant la paroi voisine de celle d'entrée. Elle nous a été indispensable pour plusieurs fonctions du projet.

fonction demarrerCmd() :
Entrée :
-Arguments : aucun
-Variables globales lues : Sens, Orientation, TabLaby, Entree, Sortie, Vitesse

Sorties :
-Return : non
-Variables globales écrites : Chemin, Orientation

Cette fonction est appelée lors du clic de l' utilisateur sur le bouton 'Démarrer' de l'interface Tk. Elle permet la résolution du labyrinthe en effectuant l'algorithme de suivi de mur (main droite ou gauche) :
On va d'abord vérifier que la Turtle n'est pas déjà positionnée sur la sortie auquel cas il n'y a pas d'algorithme à effectuer. Ensuite on va "placer une main sur le mur" en fonction de la direction choisie (toujours suivre le mur de droite ou de gauche). On avance ensuite jusqu'à la cellule suivante, toujours en suivant un mur spécifique, et c'est la que c'est un peu "tricky" (pour les bilingues), tenez vous bien : pour suivre un mur (droit ou gauche oui je sais je me répète mais c'est pour ceux qui suivent pas) on commence envisage les possibilité de sortie de l'hexagone d'une certaine manière : à chaque mur de l'hexa (0, 1, 2 ,3 ,4 ,5), on associe une valeur — correspondant à l'orientation + Sens + 3, le tout modulo 6 — qui représente le mur par lequel il va falloir passer pour continuer de garder une main sur le mur. Ensuite on regarde si le type de ce mur est une ouverture, et si c'est le cas on peut emprunter ce chemin pour passer à la case suivante. En revanche, si ce mur est un mur interne, on examine le mur suivant dans le sens trigonométrique ou dans le sens inverse — en fonction de l'algorithme de suivi de mur que l'on choisit, je vais pas redire droite ou gauche vous allez plus me supporter —. Une fois qu'on arrive enfin à accéder à la cellule suivante, on répète simplement l'opération jusqu'à atteindre la sortie (le petit carrée rouge la vous voyez ce que c'est ?), ou jusqu'à avoir parcouru deux fois le nombre total de cellules du labyrinthe, où dans ce cas il n'est pas possible d'atteindre la sortie. De plus, à la fin de l'algorithme, un message apparaît renvoyant le temps et l'état de l'algorithme : s'il a échoué ou non.

fonction avance() :
Entrée :
-Arguments : position initiale de la Turtle (index i, j de l'hexagone dans lequel on est)
-Variables globales lues : Orientation
Sorties :
-Return : nombre de cellules traversées (1)
-Variables globales écrites : Chemin

Cette fonction appelle la fonction voisin(), pour déterminer les index i et j du nouvel hexagone, en utilisant l'orientation courante, puis utilise centreHexa() pour déterminer les coordonnées en pixels du nouvel hexagone pour pouvoir y aller avec la fonction turtle.goto() qui réalise un tracé animé.

creerLabyCmd() :
Entrées :
-Arguments : non
-Variables globales lues : non
Sorties :
-Return : non
-Variables globales écrites : voir init(), creeLaby(), traceGrilleHexa()

Cette fonction est appelée lors du clic de l'utilisateur sur le bouton 'Créer' de l'interface Tk. Elle valide les valeurs de largeur et de hauteur demandées pour le labyrinthe, puis appelle les fonctions init(), creeLaby() et traceGrilleHexa(). Elle positionne aussi les widget Tk 'Scale' pour la sortie à la bonne valeur par défaut (coin supérieur droit), et rend actif le bouton Tk 'Démarrer'.

traceGrilleHexa() :
Entrées :
-Arguments : non
-Variables globales lues : Largeur, Hauteur, TabLaby, Entree, Sortie, CoulEntree, CoulSortie
Sorties :
-Return : non
-Variables globales écrites : Orientation, Chemin

Cette fonction trace le labyrinthe ainsi que les symboles de l'entrée et de la sortie. Pour optimiser le temps de tracé, les hexagones d'index en abscisse paire sont entièrement tracés, ainsi que les hexagones de la dernière ligne et de la dernière colonne. Pour les autres hexagones il suffit de tracer le quatrième côté (côté n°4, orientation sud) !
Toujours pour de meilleures performances l'animation 'turtle' est désactivée lors de ce tracé.

clicSouris() :
Entrées :
-Arguments : événement bouton déclencheur de 'ButtonRelease' (coordonnées, widget)
-Variables globales lues : LargeurCanvas, HauteurCanvas, InitTurtle, LargeurCible, HauteurCible, TabLaby
Sorties :
-Return : non
-Variables globales écrites : LargeurCanvas, HauteurCanvas

Cette fonction gère l'événement Tk 'ButtonRelease' (relâché de bouton souris). Elle est utilisée pour valider le changement de taille de la fenêtre à la souris par l'utilisateur, ainsi que le clic de la souris pour changer l'état d'une paroi du labyrinthe (le clic bascule la paroi la plus proche entre les types mur interne (MUR_INT) et ouverture (OUVERTURE)
Si il y a un labyrinthe tracé lorsque la fenêtre change de taille, il faut pour forcer le tracé du labyrinthe supprimer le module Turtle puis le réinitialiser.

retailleFenetre() :
Entrées :
-Arguments : événement bouton déclencheur de 'Configure' (largeur, hauteur, widget)
-Variables globales lues : LargeurCanvas, HauteurCanvas
Sorties :
-Return : non
-Variables globales écrites : LargeurCible, HauteurCible

Cette fonction stocke la nouvelle taille (largeur / hauteur) du Canvas (zone dans laquelle le labyrinthe est ou sera tracé) lors d'une modification par l'utilisateur de la taille de la fenêtre. Ces valeurs sont ensuite utilisées par “clicSouris” lorsque l'utilisateur à fini de retailler la fenêtre.

coordHexa() :
Entrée :
-Arguments : abscisse (en pixels, origine au centre de l'écran) et ordonnée (en pixels, origine au centre de l'écran, orientée vers le haut)
-Variables globales lues : InitTurtle, DemiLargeurTotale, DemiHauteurTotale, DemiTailleHexa
Sortie :
-Return : index du rang en largeur et hauteur de l'hexagone
-Variables globales écrites : non

Cette fonction est utilisée pour convertir une position de clic sur l'écran en index d'hexagone (permet de savoir sur quel hexagone l'utilisateur a cliqué)

tailleCmd() :
Entrée :
-Arguments : valeur du scale “Taille hexa”
-Variables globales lues : InitTurtle, DemiLargeurTotale, DemiHauteurTotale, DemiTailleHexa
Sortie :
-Return : index du rang en largeur et hauteur de l'hexagone
-Variables globales écrites : DemiTailleHexa

Cette fonction modifie la variable globale DemiTailleHexa, puis appelle la fonction permettant de recentrer le labyrinthe, et enfin appelle la fonction de tracé : “traceGrilleHexa”, qui retrace la grille.
Celle-ci est appelée de la manière suivante :

```
# Troisième ligne : scale de changement de taille
    scaTaille = Scale(labyFrame, from_=CELLULE_MIN, to=CELLULE_MAX, showvalue=False, orient='h',
label='Taille hexa', command=tailleCmd)
    scaTaille.set(CELLULE_INI)
    scaTaille.grid(row=2, column=1)
```

Pour définir la taille du scale, on utilise “from_”, qui permet de définir la valeur minimale du scale, et “to” qui définit la valeur maximale. “showvalue = False” permet de ne pas afficher la valeur courante du scale, et orient = 'h' permet d'avoir un scale horizontal. La valeur initiale du scale est positionnée par “set”, et “grid” va rendre visible et positionner le scale sur la grille.

Ma démarche et mon travail :

Dès le commencement du programme, j'ai su que mon code contiendrait bon nombre d'algorithmes et de fonctions. J'ai donc entreprit de positionner tous mes paramètres, constantes et variables globales au début du programme pour ainsi faciliter “l'accès” au code et pouvoir modifier des parties plus librement. Pour savoir par quel bout prendre le code, j'ai d'abord réfléchi aux fonctions communes à plusieurs algorithmes et donc qui me serviront tout au long du projet, telle que le tableau TabLaby. Ensuite dans la résolution du labyrinthe, j'ai d'abord réalisé plusieurs maquettes sur papier pour envisager les différents cas qui s'imposaient à moi. Puis, pour pouvoir mieux observer les problèmes et les aborder un par un, je me suis employé sur des cas simples, c'est-à-dire des petits labyrinthes composés de quatre à dix cases maximum. Puis lorsque le problème me semblait réglé je faisais tourner le programme sur des cas plus complexes avec des paramètres plus grands, et ainsi vérifier par des généralisations en résolvant plusieurs fois le labyrinthe.
Pour ce qui est de la partie interface, j'envisageais déjà de la séparer en deux parties : un bandeau contenant les différentes interactions avec l'utilisateur, là où celui-ci pourra appliquer des changements sur le labyrinthe, et un bandeau graphique dans lequel sera dessiné le labyrinthe (bandeau qu'on appelle le Canvas).

Réalisation finale et comment s'intègre mon travail dans l'équipe :

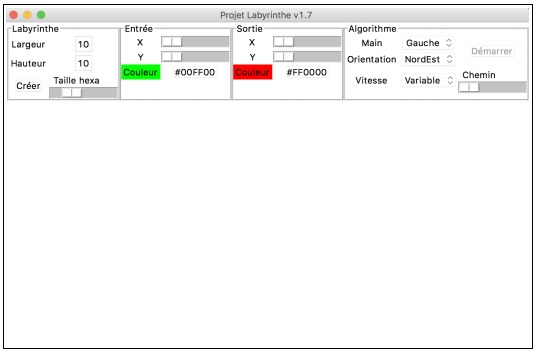
Lors de la séparation des taches, mon rôle était au début axé principalement sur la résolution, c'est à dire le travail avec le module Turtle. Lorsque l'on a commencé la partie consistant l'affichage, j'ai du travaillé aussi sur l'interface. J'ai donc travaillé sur les différents algorithmes utilisés pour la création et de résolution du labyrinthe, sur les différentes fonctions constituants l'ossature du projet tel l'initialisation de Turtle et des donnés, le traçage d'un hexagone, de la grille du labyrinthe, la détermination d'un mur voisin, le changement du type de mur, le “retaillage” de la fenêtre de ou d'autres fonctions plus mineurs comme celles déterminant la position de l'entrée et de la sortie du labyrinthe, l'orientation de la Turtle, des fonctions de Tk etc...

Enfin, je suis très satisfait du travail que l'on a fourni car on répond bien à nos attentes de début d'année. De plus, nous avons enrichi le programme de beaucoup d'idées qui nous sommes venues dans l'avancement du projet. En effet, par rapport au cahier des charges, le programme permet de :

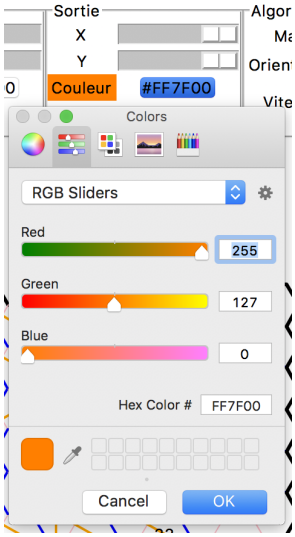
- modifier la vitesse d'animation du tracé de la résolution
- calculer le temps d’exécution
- afficher des impressions supplémentaires pour la mise-au-point
- changer les couleurs des symboles d'entrée et de sortie du labyrinthe
- avertir du type de fin de l'algorithme
- prendre en compte le changement de taille de la fenêtre
- supprimer ou ajouter des murs internes par un clic souris

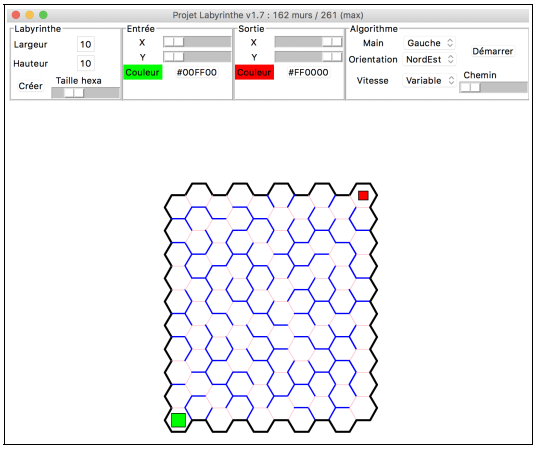
Fonctionnement global de Labyrobot

Au lancement du programme labyrinthe, on obtient automatiquement la fenêtre suivante. Elle est composée d'une zone de menu (en haut) et d'une zone graphique (Canvas Tk) dans laquelle sera tracé le labyrinthe. On remarque que la version et le titre du projet sont inscrits dans le nom de la fenêtre.



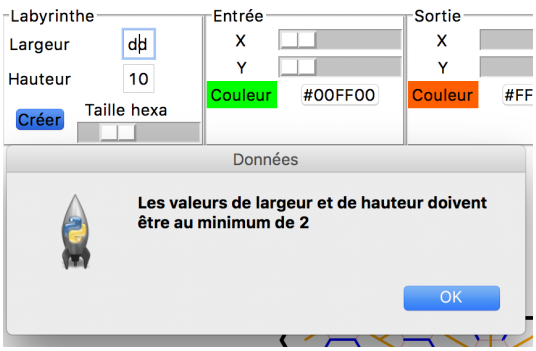
Ce menu propose les différente couleurs associable à l'entrée et à la sortie du labyrinthe. La couleur choisie est apparente dans le label “Couleur” de l'entrée et sortie, sa valeur en hexadécimal apparaît dans le texte du bouton en format (#RRGGBB).



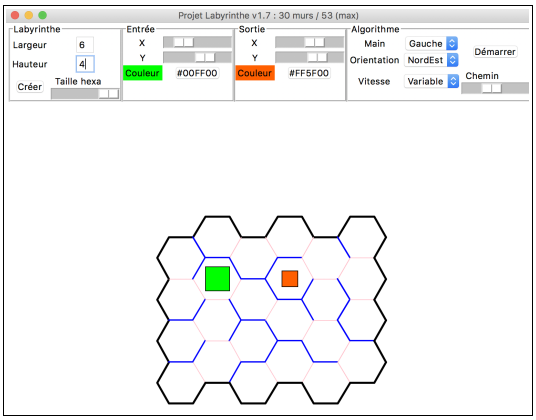


Après l'activation du bouton créer, le labyrinthe se génère automatiquement avec les largeurs et hauteurs choisies précédemment par l'utilisateur. Les valeurs de largeur et hauteur sont testées, et doivent être comprises dans un intervalle LARG_HAUT_MIN (2) et LARG_HAUT_MAX (60). Ces valeurs ont été choisies car en dessous de 2 il n'y a pas de résolution et 60 pour que le labyrinthe reste raisonnable. Dans la zone de titre de la fenêtre apparaît le nombre de murs interne et le nombre de côtés internes (ici 162 et 261)

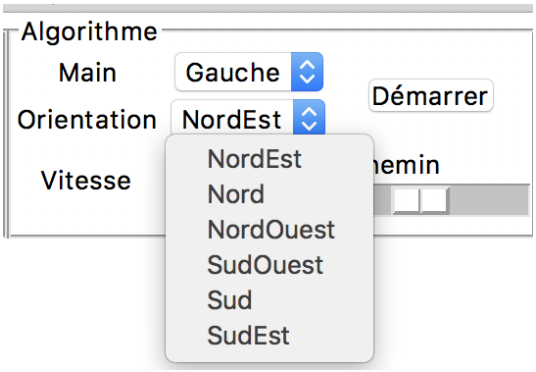
Si l'utilisateur rentre des valeurs n'importe comment, un message d'erreur est renvoyé :



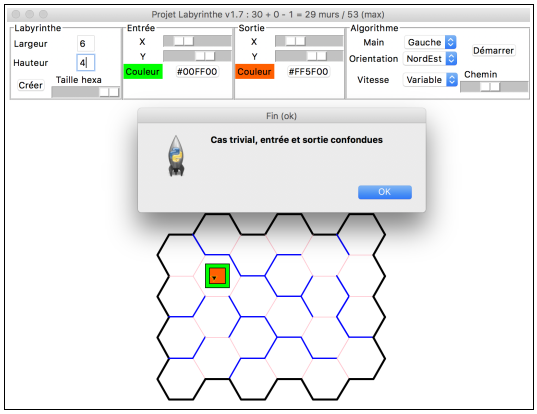
L'utilisateur peut d'ailleurs adapter dynamiquement la taille des cellules grâce au scale “Taille hexa”, comme dans l'exemple suivant, montrant également la possibilité de changer la position de l'entrée et de la sortie.



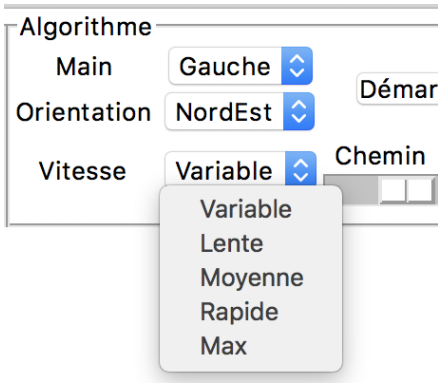
L'utilisateur a le choix dans l'orientation initiale de la Turtle à l'entrée du labyrinthe. De plus, il choisit aussi la main qui suivra le mur pour l'algorithme de résolution.

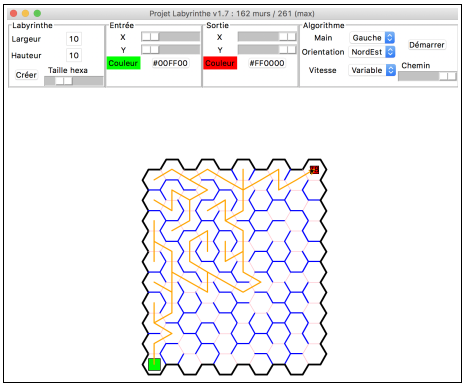


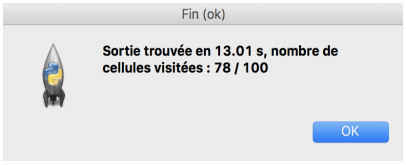
L'utilisateur clique sur le bouton démarrer — accessible qu'après avoir créé le labyrinthe — pour recherché la sortie. Si l'entrée est déjà positionnée au même endroit que la sortie, on se situe dans un cas appelé “trivial”, donc l'algorithme n'a rien à faire.



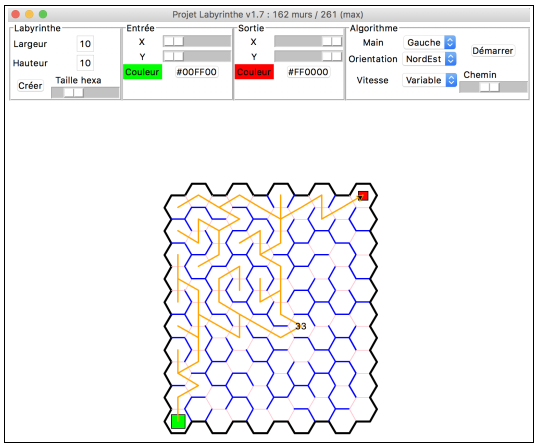
Dans un cas non-trivial, l'algorithme va rechercher la sortie à la vitesse choisie. Lorsque l'algorithme a trouvé la sortie du labyrinthe, une boîte de dialogue s'affiche avec un message indiquant le nombre d'hexagones parcourus et le temps d'exécution de celui-ci.



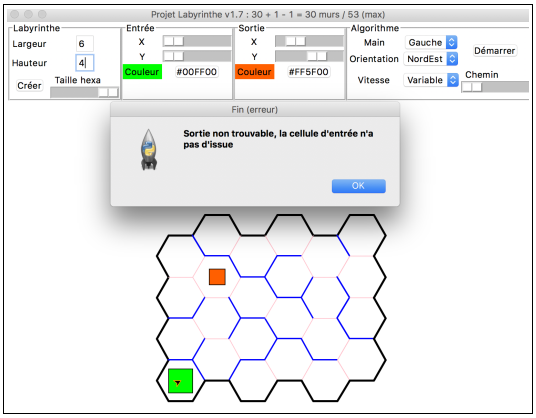




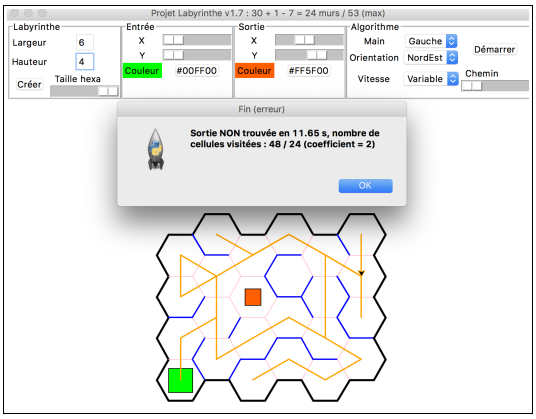
Le scale “chemin” permet alors à l'utilisateur de retracer le parcours effectué par la Turtle dont chaque case est numérotée séquentiellement entre l'entrée et la sortie.



De plus, l'utilisateur a la possibilité de changer le type d'un coté interne au labyrinthe entre ouverture et mur interne. Pour cela il suffit à l'utilisateur de cliquer sur le labyrinthe et le coté le plus proche du clic sera changé. Le bandeau de titre comporte alors, en plus des valeurs déjà affichées, le nombre de murs ajoutés et supprimés.
Dans le cas où l'utilisateur essaie de piéger la Turtle, on s'en aperçoit car cela l'empêche d'avancer.



Lorsque l'utilisateur, en modifiant des murs créer un labyrinthe imparfait dans lequel la sortie se trouve sur un ilot, l'algorithme de suivi de murs échoue et n'atteindra jamais la sortie. Un nombre maximum de cellules traversables est définit, empêchant ainsi l'algorithme de boucler indéfiniment.



Retour d'expérience :

En commençant le codage du projet, il a fallu bien organiser la structuration du code. L'organisation du programme m'a demandé beaucoup de rigueur et d'attention mais m' a été très bénéfique dans la suite du codage et a permis de gagner beaucoup de temps. Je n'avais jamais codé de programmes contenant autant de lignes et de fonctions, de plus cette manière de coder était nouvelle et s'habituer à cette organisation m'a demandé du temps. En effet, j'ai fait la distinction entre les différents types de variables globales, qui sont les variables utilisables dans toutes les fonctions du code : les constantes, qui ne sont pas modifiables (comme racine de 3 sur 2), les paramètres qui sont modifiables avant le lancement du programme mais pas par l'interface utilisateur (par exemple la version du programme) et enfin les “variables globales tout court” qui sont modifiées au court du programme telle que la largeur ou hauteur du labyrinthe. Cela a demandé aussi une rigueur dans les règles d'écriture du code car pour les différencier, chacune est écrite d'une certaine manière : les constantes et paramètres tout en MAJUSCULES, et les variables globales avec seulement une majuscule en début. Finalement, cela m'a permis de me retrouver dans le programme, de coder plus rapidement, de mieux comprendre ce que je faisais et de pouvoir expliquer aisément aux autres les avancements du code.

Dans la réalisation du projet, on utilise deux modules : Tkinter et Turtle. Lorsque j'ai commencé la création de la grille d'hexagone j'étais assez confus entre les systèmes de coordonnées utilisés. En effet, TK, turtle et la grille d'hexagone possèdent chacun un type de coordonnées différent : Turtle place l'origine du repère au centre de la fenêtre (en pixels), Tk place l'origine du repère dans le coin haut-gauche de la fenêtre (en pixels également), et enfin la grille d'hexagone est centrée dans Canvas (en index i et j). Son origine est donc plus ou moins décalée vers le coin bas-gauche en fonction de la taille de la grille. De plus, au début j'ai eu pas mal de difficulté à faire le distinguo entre les coordonnées en pixels et les coordonnées de la grille ce qui m'a freiné pas mal dans l'avancé du projet notamment pour la réalisation de la fonction “ClicMur” où j'avais besoin des coordonnées en turtle alors que l'évènement “clic” me renvoyait les coordonnées en Tk. J'ai donc eu besoin de convertir les coordonnées de Tk en celle de Turtle...

Pour créer la grille composée d'hexagones qui constituera ensuite le labyrinthe, chaque hexagone comprend une coordonnée en abscisse en fonction d'un index 'i', d'une coordonnée en ordonnée en fonction de 'j', et 'k' qui décrit la valeur des 6 cotés et une valeur centrale qui nous servira plus tard dans la génération du labyrinthe. Pour pouvoir stocker les valeurs de chaque côtés de chaque hexagone. Pour trouver cet outil me permettant de structurer ces trois donnés, j'ai effectué des recherches sur Internet qui m'ont appris à initialiser ce type de tableau et à l'utiliser.

Pour utiliser les différents modules Tkinter et Turtle, il à fallu apprendre à les maîtriser ce qui certes m'aura beaucoup appris mais aussi demandé du temps à travailler dessus. Comprendre comment ils fonctionnaient mais surtout tous les appels des différentes fonctions qui permettent d'intégrer de nouveaux éléments au code ne fût pas une mince affaire. Pour cela j'ai eu recours à la doc de Turtle (qui est malheureusement qu'en anglais) et à mon père qui avec toute sa pédagogie à réussi à me faire comprendre le module Tk. Tkinter fut surement le plus compliqué à utiliser car on a besoin de comprendre les différents types d'objets, à quoi ils servent, leurs propriétés... De plus, il y a plusieurs moyens de définir une interface ce qui fait que j'ai maintes fois hésité et suis revenu sur ce que j'avais. Enfin, la mise au point n'est pas simple car le fait de faire une petite erreur l'interface (l'objet de l'interface) n'apparait pas, et rien n'indique où est fait l'erreur contrairement à une programmation ordinaire en python.

Travail de groupe :

Pour moi le travail que j'ai fourni pendant la réalisation de ce projet s'inscrit complètement dans une optique de travail de groupe, il n'y a pas la moindre fonctionnalités du projet qui n'a pas été décidée ensemble. En effet, étant je crois celui qui a le plus de ressources en programmation du groupe, je m'employais à coder les parties plus compliquées comme les différents algorithmes. Mais au début de chaque séance on faisait une mise en commun où j'expliquais à mes camarades tous les détails de ce que j'avais modifié (et de leurs modifications), et dans quel but. Ainsi chacun était au courant de ce que faisaient les autres, ce qui nous à permis d'être plus efficace et de décider ensemble des orientations que prenait le projet. Si je pense avoir apporté ma pierre à l'édifice, les pierres de Maé et Maxence m'ont été indispensables. Maé s'est révélée être une pro du rangement informatique et m'aidait à bien organiser le code, retrouver le type de variable, corrigeait les endroits où on ne respectait pas les règles d'écritures, et son talent pour la mode et le design fut d'une grande aide dans la réalisation de l'interface. Maxence a toujours su garder un esprit critique sur ce qu'on programmait et ses conseils nous réorientaient vers tel ou tel aspect, rendant le programme bien plus polyvalent qu'escompté. En plus de leurs avancées dans le projet, mes camarades ont à tout moment gardé le sens de l'humour et le sourire ce qui a rendu la progression bien plus agréable. Séance après séance, travailler en groupe nous a permis à chacun de faire des progrès en programmation grâce à l'expérience des autres.

Perspectives d'évolution :

Pouvoir modifier les murs permet de passer d'un labyrinthe parfait à un labyrinthe imparfait. Cela permet de tester l'algorithme de résolution dans d'autres conditions et observer ses limites. : si l'entrée ou la sortie sont sur un ilot, l'algorithme de suivi de murs échoue.
Effectivement, les améliorations que l'on pourrait apporter au projet seraient de :

- permettre un choix d'algorithme, en particulier celui de Pledge
- permettre le déplacement à la souris de l'entrée et de la sortie directement sur le labyrinthe.
- grouper les zones “Entrées” et “Sortie” en une seule avec le choix entre entrée et sortie, pour plus d'ergonomie

```python
# Pour fonctions mathématiques (sqrt, degrees, atan2)
import math
# Pour tracé graphique
import turtle
# Pour calcul de temps / performances
import time
# Pour nombre aléatoire
import random
# Pour Interface Homme-Machine Tk
from tkinter import Tk, Canvas, Label, LabelFrame, Entry, Scale, Button, Frame, StringVar, OptionMenu
from tkinter.messagebox import showinfo, showerror
from tkinter.colorchooser import askcolor
import tkinter.constants


# ------------------------------------------------------------------------------
# Constantes NON modifiables

# Racine de 3 sur 2 = sinus(60°)
RACINE3_2 = math.sqrt(3) / 2

# Nombre de données par hexagone : 1 pour chaque côté (6) + 1 pour les calculs = 7
NBR_HEXA = 7

# Valeur de la donnée d'un côté de l'hexagone pour un mur externe
MUR_EXT = 0

# Valeur de la donnée d'un côté de l'hexagone pour un mur interne (fermé)
MUR_INT = 1

# Valeur de la donnée d'un côté de l'hexagone pour une ouverture
OUVERTURE = 2

# Index du chemin dans les tableaux de couleur et d'épaisseur
CHEMIN = 3


# ------------------------------------------------------------------------------
# Paramètres modifiables

# Titre du projet
TITRE = "Projet Labyrinthe"

# Version
VERSION = " v1.7"

# Utilisation de l'interface TK (sinon on utilise seulement turtle)
TK = True

# Texte d'aide
AIDE = \
"""
Zone 'Labyrinthe' :
    Entrer la largeur et la hauteur souhaitées du labyrinthe
    puis cliquer sur 'Créer'
    L'ascenseur 'Taille hexa' permet de changer la taille des hexagones
Les zones 'Entrée' et 'Sortie' permettent de changer couleur et position de l'entrée et de la sortie
Zone 'Algorithme' :
    Le choix 'Main' définit quelle main est posée sur le mur
    Le choix 'Orientation' définit l'orientation initiale
    Le choix 'Vitesse' définit la vitesse de tracé du chemin
    Le bouton 'Démarrer' lance la recherche de sortie, il n'est actif que si un labyrinthe a été créé
    L'ascenseur 'Chemin' permet de visualiser la succession des hexagones traversés
""" \
if TK else \
"""
Commandes :
```

```
    testLaby(largeur, hauteur) : crée et trace un labyrinthe de dimensions (largeur, hauteur)
    traceGrilleHexa() : retrace le labyrinthe
    demarrerCmd() : lance la résolution du labyrinthe
    changeMur(i, j, k) : échange le type de mur entre MUR_INT et OUVERTURE
    tailleCmd(taille) : change la taille des hexagones et retrace le labyrinthe
    Sens : -1 pour main gauche, +1 pour main droite contre le mur
    Entree : position de l'entrée dans le labyrinthe (par défaut coin inférieur gauche [0, 0])
    Sortie : position de la sortie dans le labyrinthe (défaut coin supérieur droit [Largeur-1, Hauteur-1])
    CoulEntree : couleur de l'entrée (#RRGGBB)
    CoulSortie : couleur de la sortie (#RRGGBB)
    Vitesse : vitesse du tracé de chemin (-1 : variable, 0 : max, croissante de 1 à 10)
"""

# Largeurs et hauteurs minimales du labyrinthe (en cellules)
LARG_HAUT_MIN = 2

# Largeurs et hauteurs maximales du labyrinthe (en cellules)
LARG_HAUT_MAX = 60

# Largeur par défaut du labyrinthe (en cellules)
LARGEUR_DEF = 10

# Hauteur par défaut du labyrinthe (en cellules)
HAUTEUR_DEF = 10

# Largeur par défaut (en pixels) de la partie du tracé du labyrinthe
LARGEUR_FEN = 765

# Hauteur par défaut (en pixels) de la partie du tracé du labyrinthe
HAUTEUR_FEN = 600

# Hauteur de la zone de menus (en pixels)
HAUTEUR_MENU = 105

# Tag de l'entrée
TAG_ENTREE = "Entree_tag"

# Tag de la sortie
TAG_SORTIE = "Sortie_tag"

# Tag du chemin
TAG_CHEMIN = "Chemin_tag"

# Label pour la main gauche
MAIN_GAUCHE = "Gauche"

# Label pour la main droite
MAIN_DROITE = "Droite"

# Main (quelle main sera posée sur le mur)
MAIN = (MAIN_GAUCHE, MAIN_DROITE)

# Orientation initiale (correspond aux directions, ne pas changer l'ordre)
ORIENTATION = ("NordEst", "Nord", "NordOuest", "SudOuest", "Sud", "SudEst")

# Epaisseur des traits (mur externe, mur interne, ouverture, chemin)
EPAISSEUR_LABY = (3, 2, 1, 2)

# Couleurs du tracé en fonction du type (mur externe, mur interne, ouverture, chemin)
COULEUR_LABY = ('black', 'blue', 'pink', 'orange')

# Libellés des choix de vitesses
VITESSE_TXT = ('Variable', 'Lente', 'Moyenne', 'Rapide', 'Max')

# Valeurs des vitesses correspondantes aux libellés de VITESSE_TXT
# A part -1 qui indique une vitesse variable, les autres valeurs sont celles de turtle.speed()
VITESSE_VAL = (      -1,      3,       6,       9,     0)
```

```
135
136 # Index de la valeur par défaut dans la liste VITESSE_TXT (valeur maximale : len(VITESSE_TXT) - 1)
137 VITESSE_DEF = 0
138
139 # Taille minimale des cellules (en pixels)
140 CELLULE_MIN = 15
141
142 # Taille initiale des cellules (en pixels)
143 CELLULE_INI = 20
144
145 # Taille maximale des cellules (en pixels)
146 CELLULE_MAX = 35
147
148 # Coefficient multiplicatif du nombre total de cellules pour déterminer le nombre de cellules visitables
149 MULT_CELLULES = 2
150
151
152 # ------------------------------------------------------------------------
153 # Variables globales
154
155 # Etat de l'initialisation de Turtle
156 InitTurtle = False
157
158 # Demi-largeur d'un hexagone (en pixels)
159 DemiTailleHexa = CELLULE_INI
160
161 # Tableau des états des côtés (6 valeurs) et de la cellule (1 valeur) : total 7 données par cellule
162 TabLaby = [[[0] * NBR_HEXA] * 1] * 1
163
164 # Largeur du labyrinthe (en cellules)
165 Largeur = 0
166
167 # Hauteur du labyrinthe (en cellules)
168 Hauteur = 0
169
170 # Demi largeur et hauteur totales
171 DemiLargeurTotale = DemiHauteurTotale = 0
172
173 # Impressions additionnelles pour le debug
174 Debug = False
175
176 # Position de l'entrée du labyrinthe (coin bas gauche au départ)
177 Entree = [0, 0]
178
179 # Position de la sortie du labyrinthe (coin haut droite à la création du labyrinthe)
180 Sortie = [0, 0]
181
182 # Couleur entrée (#RRGGBB)
183 CoulEntree = "#00FF00"
184
185 # Couleur sortie (#RRGGBB)
186 CoulSortie = "#FF0000"
187
188 # Orientation de la 'turtle' : 0 = 30°, 1 = 90°, 2 = 150°, 3 = 210°, 4 = 270°, 5=330°
189 Orientation = 0
190
191 # Vitesse de la turtle
192 Vitesse = VITESSE_VAL[VITESSE_DEF]
193
194 # Nouvelle orientation testée : +1 si main droite, -1 si main gauche
195 # La valeur initiale doit être cohérente avec MAIN[0]
196 # Si c'est main gauche, on tourne vers la droite pour mettre la main gauche au mur
197 # Si c'est main droite, on tourne vers la gauche pour mettre la main droite au mur
198 Sens = -1 if MAIN[0] == MAIN_GAUCHE else 1
199
200 # Chemin trouvé (liste des cellules traversées)
201 Chemin = []
```

```
202
203 # Widget canvas pour le tracé du labyrinthe sous Tk
204 Wcanvas = 0
205
206 # Widget de base de la fenêtre
207 Wroot = 0
208
209 # Largeurs et hauteurs de la partie graphique (canvas)
210 LargeurCanvas = HauteurCanvas = LargeurCible = HauteurCible = 0
211
212 # Nombre de murs internes (initial, ajoutés, retirés, maximum)
213 MursInit = MursPlus = MursMoins = MursMax = 0
214
215
216 # ------------------------------------------------------------------------
217 # Fonctions
218
219 # Affichage des informations
220 def info() :
221     print(TITRE)
222     print(VERSION)
223     print("Pour plus d'informations :\n>>> aide()\n")
224     return
225
226
227 # Affichage du texte d'aide
228 def aide() :
229     print(AIDE)
230     return
231
232
233 # Création interface Tk
234 # a_largeur : largeur du labyrinthe
235 # a_hauteur : hauteur du labyrinthe
236 def creeInterface(a_largeur = LARGEUR_FEN, a_hauteur = HAUTEUR_FEN) :
237     # Variables globales de type entier
238     global LargeurCanvas, HauteurCanvas, LargeurCible, HauteurCible
239     # Variables globales de type Widget Tk
240     global Wroot, Wcanvas
241     # Variables globales de type StringVar Tk
242     global VarLargeur, VarHauteur, VarOrient
243     # Variables globales de type Scale Tk
244     global ScaEntreeX, ScaEntreeY, ScaSortieX, ScaSortieY, ScaChemin
245     # Variables globales de type Label Tk
246     global LblCoulEntree, LblCoulSortie
247     # Variables globales de type Button Tk
248     global BtnCoulEntree, BtnCoulSortie, BtnDemarrer
249     # Fenêtre principale
250     Wroot = Tk()
251     # Titre de la fenêtre
252     majTitre()
253     # Initialisation de la largeur et hauteur du graphique (et de la sauvegarde utilisée pour resize)
254     LargeurCible = LargeurCanvas = a_largeur
255     HauteurCible = HauteurCanvas = a_hauteur
256     # Définition de la taille de la fenêtre principale
257     Wroot.geometry(str(a_largeur)+"x"+str(a_hauteur+HAUTEUR_MENU)+"-10+10")
258     # Fonction appelée pour le changement de taille de la fenêtre
259     Wroot.bind('<Configure>', retailleFenetre)
260     # Fonction appelée pour le lacher du bouton souris (indique la fin du resize)
261     Wroot.bind('<ButtonRelease>', clicSouris)
262     # Frame des données
263     dataFrame = Frame(Wroot)
264     # Partie 'Labyrinthe'
265     labyFrame = LabelFrame(dataFrame, text='Labyrinthe')
266     # Première ligne : largeur du labyrinthe
267     Label(labyFrame, text='Largeur').grid(row=0, column=0)
268     VarLargeur = StringVar(Wroot)
```

```
    VarLargeur.set(LARGEUR_DEF)
    Entry(labyFrame, textvariable=VarLargeur, width=2).grid(row=0, column=1)
    # Deuxième ligne : hauteur du labyrinthe
    Label(labyFrame, text='Hauteur').grid(row=1, column=0)
    VarHauteur = StringVar(Wroot)
    VarHauteur.set(HAUTEUR_DEF)
    Entry(labyFrame, textvariable=VarHauteur, width=2).grid(row=1, column=1)
    # Troisième ligne : bouton 'Créer'
    Button(labyFrame, text='Créer', command=creerLabyCmd).grid(row=2, column=0, columnspan=1)
    # Troisième ligne : scale de changement de taille
    scaTaille = Scale(labyFrame, from_=CELLULE_MIN, to=CELLULE_MAX, showvalue=False, orient='h', label='Taille
-   hexa', command=tailleCmd)
    scaTaille.set(CELLULE_INI)
    scaTaille.grid(row=2, column=1)
    # Fin de la partie labyFrame
    labyFrame.grid(row=0, column=0, sticky=tkinter.N+tkinter.S)
    # Partie 'Entrée'
    entreeFrame = LabelFrame(dataFrame, text='Entrée')
    # Abscisse
    Label(entreeFrame, text="X").grid(row=0, column=0)
    ScaEntreeX = Scale(entreeFrame, to=LARGEUR_DEF-1, showvalue=False, orient='h', command=xEntreeCmd)
    ScaEntreeX.grid(row=0, column=1)
    # Ordonnée
    Label(entreeFrame, text="Y").grid(row=1, column=0)
    ScaEntreeY = Scale(entreeFrame, to=HAUTEUR_DEF-1, showvalue=False, orient='h', command=yEntreeCmd)
    ScaEntreeY.grid(row=1, column=1)
    # Label Couleur
    LblCoulEntree = Label(entreeFrame, text='Couleur', bg=CoulEntree)
    LblCoulEntree.grid(row=2, column=0)
    # Bouton Couleur
    BtnCoulEntree = Button(entreeFrame, text=CoulEntree, bg=CoulEntree, command=coulEntreeCmd)
    BtnCoulEntree.grid(row=2, column=1)
    # Fin de la partie entreeFrame
    entreeFrame.grid(row=0, column=1, sticky=tkinter.N+tkinter.S)
    # Partie 'Sortie'
    sortieFrame = LabelFrame(dataFrame, text='Sortie')
    # Abscisse
    Label(sortieFrame, text="X").grid(row=0, column=0)
    ScaSortieX = Scale(sortieFrame, to=LARGEUR_DEF-1, showvalue=False, orient='h', command=xSortieCmd)
    ScaSortieX.grid(row=0, column=1)
    # Ordonnée
    Label(sortieFrame, text="Y").grid(row=1, column=0)
    ScaSortieY = Scale(sortieFrame, to=HAUTEUR_DEF-1, showvalue=False, orient='h', command=ySortieCmd)
    ScaSortieY.grid(row=1, column=1)
    # Label Couleur
    LblCoulSortie = Label(sortieFrame, text='Couleur', bg=CoulSortie)
    LblCoulSortie.grid(row=2, column=0)
    # Bouton Couleur
    BtnCoulSortie = Button(sortieFrame, text=CoulSortie, bg=CoulSortie, command=coulSortieCmd)
    BtnCoulSortie.grid(row=2, column=1)
    # Fin de la partie sortieFrame
    sortieFrame.grid(row=0, column=2, sticky=tkinter.N+tkinter.S)
    # Partie 'Algo'
    algoFrame = LabelFrame(dataFrame, text='Algorithme')
    # Main
    Label(algoFrame, text='Main').grid(row=0, column=0)
    varMain = StringVar(Wroot)
    varMain.set(MAIN[0])
    OptionMenu(algoFrame, varMain, *MAIN, command=mainCmd).grid(row=0, column=1)
    # Orientation
    Label(algoFrame, text='Orientation').grid(row=1, column=0)
    VarOrient = StringVar(Wroot)
    VarOrient.set(ORIENTATION[0])
    OptionMenu(algoFrame, VarOrient, *ORIENTATION, command=orientationCmd).grid(row=1, column=1)
    # Vitesse
    Label(algoFrame, text='Vitesse').grid(row=2, column=0)
    VarVitesse = StringVar(Wroot)
```

```
    VarVitesse.set(VITESSE_TXT[VITESSE_DEF])
    OptionMenu(algoFrame, VarVitesse, *VITESSE_TXT, command=vitesseCmd).grid(row=2, column=1)
    # Bouton 'Démarrer'
    BtnDemarrer = Button(algoFrame, text='Démarrer', command=demarrerCmd, state='disabled')
    BtnDemarrer.grid(row=0, column=2, rowspan=2)
    # Scale 'Chemin'
    ScaChemin = Scale(algoFrame, showvalue=False, orient='h', label='Chemin', command=cheminCmd)
    ScaChemin.grid(row=2, column=2)
    # Fin de la partie algoFrame
    algoFrame.grid(row=0, column=3, sticky=tkinter.N+tkinter.S)
    # Fin de la partie dataFrame et affichage
    dataFrame.grid(row=0, column=0)
    # Fenêtre graphique (canvas)
    Wcanvas = Canvas(Wroot, background='white', width=a_largeur, height=a_hauteur)
    # Fin de la partie Wcanvas et affichage
    Wcanvas.grid(row=1, column=0)
    return


# Changement du choix de la main collée au mur (gauche => -1, droite => +1)
# a_val : valeur du menu choisie
def vitesseCmd(a_val) :
    global Vitesse
    if Debug : print('vitesseCmd :', a_val, '/ index :', VITESSE_TXT.index(a_val), '/ vitesse :',
    VITESSE_VAL[VITESSE_TXT.index(a_val)])
    Vitesse = VITESSE_VAL[VITESSE_TXT.index(a_val)]
    return


# Modification de la taille des hexagones
# a_val : valeur du scale (demi taille des hexagones)
def tailleCmd(a_val) :
    global DemiTailleHexa
    DemiTailleHexa = int(a_val)
    # Recalcul et retracé du labyrinthe s'il existe
    if InitTurtle :
        initCentrage()
        traceGrilleHexa()
    return


# Gestion du changement de taille de la fenêtre (événement 'Configure')
# a_evt : événement déclencheur
def retailleFenetre(a_evt) :
    global LargeurCible, HauteurCible
    # On sauve les valeurs des événements 'configure' de l'utilisateur (générés par Wroot)
    if a_evt.widget == Wroot :
        if a_evt.width != LargeurCanvas or a_evt.height != HauteurCanvas + HAUTEUR_MENU :
            if Debug : print('retailleFenetre', a_evt.width, a_evt.height)
            if a_evt.width > 1 :
                LargeurCible = a_evt.width
                HauteurCible = a_evt.height - HAUTEUR_MENU
    return


# Fin du changement de taille de la fenêtre (événement 'ButtonRelease')
# a_evt : événement déclencheur
def clicSouris(a_evt) :
    global LargeurCanvas, HauteurCanvas, InitTurtle
    if Debug : print('clicSouris', a_evt.widget, a_evt.x, a_evt.y)
    # Si la fenêtre a changé de taille
    if (a_evt.widget == Wroot or a_evt.widget == Wcanvas) and (LargeurCible != LargeurCanvas or HauteurCible !=
    HauteurCanvas) :
        # Sauvegarde de la nouvelle taille
        LargeurCanvas = LargeurCible
        HauteurCanvas = HauteurCible
        # Modification du Wcanvas pour la nouvelle taille
```

```
400         Wcanvas.configure(width=LargeurCanvas, height=HauteurCanvas)
401         if Debug : print('clicSouris', InitTurtle, LargeurCible, HauteurCible)
402         # Retracé du labyrinthe s'il existe
403         if InitTurtle == True :
404             global turtle
405             # Suppression de la variable turtle existante
406             del turtle
407             # On recrée le module turtle perdu par la commande précédente
408             import turtle
409             import importlib
410             # On force le rechargement de turtle
411             importlib.reload(turtle)
412             # On réinitialise turtle
413             InitTurtle = False
414             initTurtle()
415             # On retrace la grille
416             traceGrilleHexa()
417     # Autre cas (button release sur le canvas)
418     elif a_evt.widget == Wcanvas :
419         # Position avec origine au centre (ordonnées vers le haut)
420         x = a_evt.x - LargeurCanvas / 2
421         y = HauteurCanvas / 2 - a_evt.y
422         # Gestion du clic sur un mur
423         clicMur(x, y)
424     return
425 
426 
427 # Change le type de côté entre OUVERTURE et MUR_INT
428 # a_x : abscisse (origine au centre de l'écran)
429 # a_y : ordonnée (origine au centre de l'écran, orientée vers le haut)
430 def clicMur(a_x, a_y) :
431     # Récupération de la position en fraction d'hexagone
432     (i, j) = coordHexa(a_x, a_y)
433     # Coordonnées de l'hexagone
434     i = round(i)
435     j = round(j)
436     # On ne prend que les hexagones dans le labyrinthe
437     if 0 <= i and i < Largeur and 0 <= j and j < Hauteur :
438         # Coordonnées du centre le plus proche
439         c_x, c_y = centreHexa(i, j)
440         # Angle du clic par rapport au centre le plus proche
441         angle = math.degrees(math.atan2(a_y - c_y, a_x - c_x))
442         # Index du mur le plus proche
443         k = int(angle % 360 / 60)
444         if Debug : print('clicMur', i, j, k)
445         # Changement du type de mur
446         changeMur(i, j, k)
447         if Debug : turtle.goto(c_x, c_y)
448     return
449 
450 
451 # Change le type de côté entre OUVERTURE et MUR_INT
452 # a_i : abscisse de l'hexagone
453 # a_j : ordonnée de l'hexagone
454 # a_k : numéro du mur
455 def changeMur(a_i, a_j, a_k) :
456     global MursPlus, MursMoins
457     # Validité des arguments
458     if 0 <= a_i and a_i < Largeur and 0 <= a_j and a_j < Hauteur and 0 <= a_k and a_k < 6 :
459         # Changement du type de mur
460         typeMur = TabLaby[a_i][a_j][a_k]
461         # On ne change pas le type de mur externe
462         if typeMur != MUR_EXT :
463             # Paroi voisine à changer aussi
464             (v_i, v_j, v_k) = voisin(a_i, a_j, a_k)
465             # Nouveau type de côté (on échange MUR_INT et OUVERTURE)
466             if typeMur == OUVERTURE :
```

```
467                 nouveauType = MUR_INT
468                 # On ajoute un mur
469                 MursPlus += 1
470             else :
471                 nouveauType = OUVERTURE
472                 # On retire un mur
473                 MursMoins += 1
474             # On affecte le nouveau type de mur pour les 2 parois
475             TabLaby[a_i][a_j][a_k] = TabLaby[v_i][v_j][v_k] = nouveauType
476             # Retracé du labyrinthe
477             traceGrilleHexa()
478             # Mise à jour du titre
479             majTitre()
480     return
481
482
483 # Mise à jour du titre de la fenêtre
484 def majTitre() :
485     # Titre de la fenêtre après ajout / suppression de murs (on les indique)
486     if MursPlus > 0 or MursMoins > 0 :
487         titre = TITRE + VERSION + ' : ' + str(MursInit) + ' + ' + str(MursPlus) + ' - ' + str(MursMoins) + ' = ' + str(MursInit +
-         MursPlus - MursMoins) + ' murs / ' + str(MursMax) + ' (max)'
488     # Titre de la fenêtre avant création du labyrinthe
489     elif MursInit == 0 :
490         titre = TITRE + VERSION
491     # Titre de la fenêtre après création du labyrinthe mais avant ajout / suppression de murs
492     else :
493         titre = TITRE + VERSION + ' : ' + str(MursInit) + ' murs / ' + str(MursMax) + ' (max)'
494     # Affichage du titre avec Tk
495     if Wroot != 0 :
496         Wroot.title(titre)
497     # Affichage du titre avec turtle (sans Tk)
498     else :
499         turtle.title(titre)
500     return
501
502
503 # Numéro d'un hexagone en fonction de la position (a_x, a_y)
504 # a_x : abscisse (origine au centre de l'écran)
505 # a_y : ordonnée (origine au centre de l'écran, orientée vers le haut)
506 # Return : numéro de l'hexagone (i, j)
507 def coordHexa(a_x, a_y) :
508     if InitTurtle :
509         i = (a_x + DemiLargeurTotale) / DemiTailleHexa / 1.5
510         j = (((a_y + DemiHauteurTotale) / DemiTailleHexa / RACINE3_2) - round(i) % 2) / 2
511         if Debug : print('coordHexa', i, j)
512     else :
513         i = j = 0
514     return (i, j)
515
516
517 # Replay de l'algorithme de sortie du labyrinthe
518 # a_val : valeur du scale
519 def cheminCmd(a_val) :
520     if len(Chemin) > 0 :
521         # Récupération de la valeur du scale (de 0 à nombre de cellules - 1 traversées)
522         pos = int(a_val)
523         # Affichage d'un symbole (la valeur du scale) sur le centre de la cellule (en noir, plus visible)
524         traceSymbole(Chemin[pos][0], Chemin[pos][1], TAG_CHEMIN, 'black', str(pos+1))
525     return
526
527
528 # Lancement de l'algorithme de sortie du labyrinthe (bouton 'Démarrer')
529 # 1 : on vérifie qu'on est pas déjà sur la sortie
530 # 2 : on vérifie qu'on peut changer de cellule
531 # 3 : on place la main choisie sur le mur en fonction de la direction choisie
532 # 4 : on avance en suivant toujours le mur jusqu'à la sortie ou un nombre maximum de cellules visité
```

```
533 def demarrerCmd() :
534     global Orientation
535     # On retrace le labyrinthe (pour effacer les chemins turtle précédents s'il y en a)
536     traceGrilleHexa()
537     # Initialisation pour calcul du temps de l'algorithme
538     tps = time.time()
539     # Orientation de la 'turtle'
540     orientation()
541     # On lève le stylo
542     turtle.penup()
543     # On va au centre de la cellule
544     (x, y) = centreHexa(Entree[0], Entree[1])
545     turtle.goto(x, y)
546     # Affichage de la 'turtle'
547     turtle.showturtle()
548     # On trace !
549     turtle.pendown()
550     # Couleur du chemin
551     turtle.pencolor(COULEUR_LABY[CHEMIN])
552     # Epaisseur du chemin
553     turtle.pensize(EPAISSEUR_LABY[CHEMIN])
554     # Position de départ (attention pos = Entree ne marche pas)
555     pos = [Entree[0], Entree[1]]
556     # Nombre de cellules visitées
557     cellules = 1
558     # Nombre maximum de cellules visitables
559     maxCellules = Largeur * Hauteur * MULT_CELLULES
560     # Mise à zéro du chemin
561     Chemin.clear()
562     # Première cellule
563     Chemin.append((Entree[0], Entree[1]))
564     # Si on a Tk
565     if Wcanvas != 0 :
566         # On efface le symbole (texte) du chemin
567         Wcanvas.delete(TAG_CHEMIN)
568     # Cas trivial : on est déjà à la sortie
569     if Entree == Sortie :
570         message = 'Cas trivial, entrée et sortie confondues'
571         erreur = False
572     else :
573         # Il faut pouvoir avancer
574         prison = TabLaby[pos[0]][pos[1]][0] != OUVERTURE and TabLaby[pos[0]][pos[1]][1] != OUVERTURE and \
575                  TabLaby[pos[0]][pos[1]][2] != OUVERTURE and TabLaby[pos[0]][pos[1]][3] != OUVERTURE and \
576                  TabLaby[pos[0]][pos[1]][4] != OUVERTURE and TabLaby[pos[0]][pos[1]][5] != OUVERTURE
577         if not prison :
578             # Vitesse initiale (la plus lente en cas de vitesse variable)
579             turtle.speed(1 if Vitesse < 0 else Vitesse)
580             # Tant qu'on a pas trouvé la sortie on avance en suivant le mur (maximum maxCellules visitées)
581             while pos != Sortie and cellules < maxCellules :
582                 if Debug : print('demarrerCmd', cellules, Orientation)
583                 # On change l'orientation dans le sens de 'Main' jusqu'au mur libre
584                 for i in range(6) :
585                     # Si on ne peut pas avancer on change de direction (pour le départ on met la main sur le mur)
586                     if TabLaby[pos[0]][pos[1]][Orientation] != OUVERTURE or cellules == 1 and TabLaby[pos[0]][pos[1]]
-                    [Orientation-Sens] == OUVERTURE :
587                         # Nouvelle orientation
588                         Orientation = (Orientation + Sens) % 6
589                         # Visualisation de l'orientation
590                         orientation()
591                     # On peut avancer dans cette direction
592                     else :
593                         # On avance dans la cellule suivante
594                         cellules += avance(pos)
595                         # Nouvelle orientation
596                         # Main gauche sur le mur : 0 => 2, 1 => 3, 2 => 4, 3 => 5, 4 => 0, 5 => 1
597                         # Main droite sur le mur : 0 => 4, 1 => 5, 2 => 0, 3 => 1, 4 => 2, 5 => 3
598                         Orientation = (Orientation + Sens + 3) % 6
```

```
                    # Si vitesse variable : augmentation progressive de la vitesse de tracé du chemin
                    if Vitesse < 0 :
                        turtle.speed(cellules / 2)
                    # On a trouvé un mur libre, on ne continue pas à chercher
                    break
            # Calcul du temps écoulé
            tps = round(time.time() - tps, 2)
            # On a trouvé la sortie !
            if pos == Sortie :
                message = 'Sortie trouvée en ' + str(tps) + ' s, nombre de cellules visitées : ' + str(cellules) + ' / ' + str(Largeur
                * Hauteur)
                erreur = False
            # Sortie par nombre max atteint (erreur)
            else :
                message = 'Sortie NON trouvée en ' + str(tps) + ' s, nombre de cellules visitées : ' + str(cellules) + ' / ' +
                str(Largeur * Hauteur) + ' (coefficient = ' + str(MULT_CELLULES) + ')'
                erreur = True
        # Cas de la prison (erreur)
        else :
            message = "Sortie non trouvable, la cellule d'entrée n'a pas d'issue"
            erreur = True
    # Message de fin
    print(message)
    # Si on a Tk
    if TK :
        # Le scale du chemin doit aller de 0 à cellules - 1
        ScaChemin.configure(to=cellules-1)
        # Le chemin est positionné à la sortie
        ScaChemin.set(cellules-1)
        # Affichage dans la dernière cellule
        cheminCmd(cellules-1)
        # Affichage de la boîte de fin
        if erreur :
            showerror('Fin (erreur)', message)
        else :
            showinfo('Fin (ok)', message)
    return


# Changement du choix de la main collée au mur (gauche => -1, droite => +1)
# a_val : valeur du menu choisie
def mainCmd(a_val) :
    global Sens
    # Sur main gauche collée au mur, Sens = -1 (on tourne vers la droite, sens montre)
    if a_val == MAIN_GAUCHE :
        Sens = -1
    # Sur main droite collée au mur, Sens = +1 (on tourne vers la gauche, sens trigo)
    else :
        Sens = 1
    return


# Changement du choix d'orientation de départ
# a_val : valeur du menu choisie
def orientationCmd(a_val) :
    # L'index 0 correspond à Nord-Est : 30 degrés, le 5 à Sud-Est : 330 °
    changeOrientation(ORIENTATION.index(a_val))
    return


# Changement de la position en abscisse de l'entrée du labyrinthe
# a_val : valeur du scale
def xEntreeCmd(a_val) :
    global Entree
    # Récupération de l'abscisse de l'entrée
    Entree[0] = int(a_val)
    # On trace l'entrée avec la nouvelle valeur
```

```
664     traceEntree()
665     return
666
667
668 # Changement de la position en ordonnée de l'entrée du labyrinthe
669 # a_val : valeur du scale
670 def yEntreeCmd(a_val) :
671     global Entree
672     # Récupération de l'ordonnée de l'entrée
673     Entree[1] = int(a_val)
674     # On trace l'entrée avec la nouvelle valeur
675     traceEntree()
676     return
677
678
679 # Changement de la position en abscisse de la sortie du labyrinthe
680 # a_val : valeur du scale
681 def xSortieCmd(a_val) :
682     global Sortie
683     # Récupération de l'abscisse de la sortie
684     Sortie[0] = int(a_val)
685     # On trace la sortie avec la nouvelle valeur
686     traceSortie()
687     return
688
689
690 # Changement de la position en ordonnée de la sortie du labyrinthe
691 # a_val : valeur du scale
692 def ySortieCmd(a_val) :
693     global Sortie
694     # Récupération de l'ordonnée de la sortie
695     Sortie[1] = int(a_val)
696     # On trace la sortie avec la nouvelle valeur
697     traceSortie()
698     return
699
700
701 # Lancement de la création du labyrinthe (bouton 'Créer')
702 def creerLabyCmd() :
703     try :
704         # Récupération de la largeur choisie
705         largeur = int(VarLargeur.get())
706         # Récupération de la hauteur choisie
707         hauteur = int(VarHauteur.get())
708     except :
709         largeur = hauteur = 0
710     # Vérification de la validité des valeurs saisies
711     if largeur < LARG_HAUT_MIN or hauteur < LARG_HAUT_MIN :
712         # Valeurs erronées
713         showerror('Données', 'Les valeurs de largeur et de hauteur doivent être au minimum de ' + str(LARG_HAUT_MIN))
714     elif largeur > LARG_HAUT_MAX or hauteur > LARG_HAUT_MAX :
715         # Valeurs erronées
716         showerror('Données', 'Les valeurs de largeur et de hauteur doivent être au maximum de ' +
-         str(LARG_HAUT_MAX))
717     else :
718         # Le scale en X doit aller de 0 à largeur - 1
719         ScaEntreeX.configure(to=largeur-1)
720         # Le scale en Y doit aller de 0 à hauteur - 1
721         ScaEntreeY.configure(to=hauteur-1)
722         # Le scale en X doit aller de 0 à largeur - 1
723         ScaSortieX.configure(to=largeur-1)
724         # Le scale en Y doit aller de 0 à hauteur - 1
725         ScaSortieY.configure(to=hauteur-1)
726         # Initialisation du labyrinthe
727         init(largeur, hauteur)
728         # Création du labyrinthe parfait aléatoirement
729         creeLaby()
```

```
730         # Tracé du labyrinthe
731         traceGrilleHexa()
732         # Valeur par défaut de la sortie : coin supérieur droit
733         ScaSortieX.set(largeur-1)
734         ScaSortieY.set(hauteur-1)
735         # Le bouton 'démarrer' est maintenant actif
736         BtnDemarrer.configure(state='normal')
737     return
738 
739 
740 # Callback du bouton pour la couleur entrée
741 def coulEntreeCmd() :
742     global CoulEntree
743     # Boîte de dialogue de choix de couleur
744     coul = askcolor(CoulEntree)
745     # Si une couleur est choisie
746     if coul[1] != None :
747         # Récupération de la couleur choisie
748         CoulEntree = coul[1]
749         # On utilise la couleur choisie comme couleur de fond du label
750         LblCoulEntree.configure(bg=coul[1])
751         # On utilise la couleur choisie comme label et couleur de fond du bouton
752         BtnCoulEntree.configure(text=coul[1].upper(), bg=coul[1])
753         # On retrace l'entrée avec la bonne couleur
754         traceEntree()
755     return
756 
757 
758 # Callback du bouton pour la couleur sortie
759 def coulSortieCmd() :
760     global CoulSortie
761     # Boîte de dialogue de choix de couleur
762     coul = askcolor(CoulSortie)
763     # Si une couleur est choisie
764     if coul[1] != None :
765         # Récupération de la couleur choisie
766         CoulSortie = coul[1]
767         # On utilise la couleur choisie comme couleur de fond du label
768         LblCoulSortie.configure(bg=coul[1])
769         # On utilise la couleur choisie comme label et couleur de fond du bouton
770         BtnCoulSortie.configure(text=coul[1].upper(), bg=coul[1])
771         # On retrace la sortie avec la bonne couleur
772         traceSortie()
773     return
774 
775 
776 # Changement du choix d'orientation de départ
777 # a_val : valeur de l'orientation (0 à 5)
778 def changeOrientation(a_val) :
779     global Orientation
780     # L'index 0 correspond à 30 degrés, le 5 à 330 °
781     Orientation = a_val
782     # Positionne la turtle en fonction de l'orientation
783     if InitTurtle :
784         orientation()
785     return
786 
787 
788 # Positionne l'orientation de la 'turtle'
789 def orientation() :
790     # Orientation de la turtle
791     turtle.setheading(30 + Orientation * 60)
792     return
793 
794 
795 # Initialisation de turtle
796 def initTurtle() :
```

```
    global InitTurtle
    # Initialisation de Turtle si pas déjà fait
    if not InitTurtle :
        # Tracé avec Tk : on trace dans le canvas
        if TK :
            from turtle import RawPen
            global turtle
            turtle = RawPen(Wcanvas)
            if Debug : print('initTurtle', turtle.getscreen().screensize())
        # Tracé sans Tk : création de la fenêtre et titre
        else :
            majTitre()
            # Action sur clic souris
            turtle.onscreenclick(clicMur)
        # Cache le pointeur de Turtle
        turtle.hideturtle()
        # Vitesse maximale du tracé
        turtle.speed(0)
        # Orientation initiale : Nord-Est
        orientation()
        # Turtle est maintenant initialisé
        InitTurtle = True
    # Turtle est déjà initialisé : on nettoie la fenêtre
    else :
        turtle.clear()
    return


# Tracé d'un hexagone entier
# a_x : abscisse du centre de l'hexagone (en pixels)
# a_y : ordonnée du centre de l'hexagone (en pixels)
# a_type : type de côté (MUR_EXT, MUR_INT, OUVERTURE)
def traceHexa(a_x, a_y, a_type) :
    # On lève le stylo
    turtle.up()
    # On va au coin droit de l'hexagone
    turtle.goto(a_x + DemiTailleHexa,     a_y)
    # On abaisse le stylo
    turtle.down()
    # On trace le premier segment (dans le sens trigo) : mur 0
    turtle.pensize(EPAISSEUR_LABY[a_type[0]])
    turtle.pencolor(COULEUR_LABY[a_type[0]])
    turtle.goto(a_x + DemiTailleHexa / 2, a_y + DemiTailleHexa * RACINE3_2)
    # On trace le deuxième segment (dans le sens trigo) : mur 1
    turtle.pensize(EPAISSEUR_LABY[a_type[1]])
    turtle.pencolor(COULEUR_LABY[a_type[1]])
    turtle.goto(a_x - DemiTailleHexa / 2, a_y + DemiTailleHexa * RACINE3_2)
    # On trace le troisième segment (dans le sens trigo) : mur 2
    turtle.pensize(EPAISSEUR_LABY[a_type[2]])
    turtle.pencolor(COULEUR_LABY[a_type[2]])
    turtle.goto(a_x - DemiTailleHexa,     a_y)
    # On trace le quatrième segment (dans le sens trigo) : mur 3
    turtle.pensize(EPAISSEUR_LABY[a_type[3]])
    turtle.pencolor(COULEUR_LABY[a_type[3]])
    turtle.goto(a_x - DemiTailleHexa / 2, a_y - DemiTailleHexa * RACINE3_2)
    # On trace le cinquième segment (dans le sens trigo) : mur 4
    turtle.pensize(EPAISSEUR_LABY[a_type[4]])
    turtle.pencolor(COULEUR_LABY[a_type[4]])
    turtle.goto(a_x + DemiTailleHexa / 2, a_y - DemiTailleHexa * RACINE3_2)
    # On trace le sixième segment (dans le sens trigo) : mur 5
    turtle.pensize(EPAISSEUR_LABY[a_type[5]])
    turtle.pencolor(COULEUR_LABY[a_type[5]])
    turtle.goto(a_x + DemiTailleHexa,     a_y)
    return


# Tracé du bas (mur 4) d'un hexagone
```

```
# a_x : abscisse du centre de l'hexagone (en pixels)
# a_y : ordonnée du centre de l'hexagone (en pixels)
# a_type : type de mur (MUR_EXT, MUR_INT, OUVERTURE)
def traceBasHexa(a_x, a_y, a_type) :
    # On lève le stylo
    turtle.up()
    # On va au cinquième segment (dans le sens trigo)
    turtle.goto(a_x - DemiTailleHexa / 2, a_y - DemiTailleHexa * RACINE3_2)
    # On abaisse le stylo
    turtle.down()
    # On trace le cinquième segment (dans le sens trigo) : mur 4
    turtle.pensize(EPAISSEUR_LABY[a_type])
    turtle.pencolor(COULEUR_LABY[a_type])
    turtle.goto(a_x + DemiTailleHexa / 2, a_y - DemiTailleHexa * RACINE3_2)
    return


# Initialisation du graphique et du labyrinthe
# a_largeur : largeur du labyrinthe
# a_hauteur : hauteur du labyrinthe
def init(a_largeur, a_hauteur):
    # Initialisation de turtle
    initTurtle()
    # Initialisation du tableau des données
    initDonnees(a_largeur, a_hauteur)
    return


"""
Nombre de murs 'internes' :
    parois horizontales : (H - 1) * L
    parois inclinées    : (2 * H - 1) * (L - 1)
    Parois totales : (H - 1) * L + (2 * H - 1) * (L - 1) = 3HL - 2(H+L) + 1
"""
# Calcul du nombre de parois internes en fonction de la taille du labyrinthe
# a_largeur : largeur du labyrinthe
# a_hauteur : hauteur du labyrinthe
# Return : nombre de parois internes
def nbParoisInternes(a_largeur, a_hauteur) :
    return (3 * a_largeur * a_hauteur - 2 * (a_largeur + a_hauteur) + 1)


# Tracé d'une grille d'hexagones centrés
def traceGrilleHexa():
    global Orientation
    # Initialisation pour calcul du temps de tracé
    tps = time.time()
    # Désactive l'animation de tracé (pour accelérer le tracé du labyrinthe)
    screen = turtle.getscreen()
    screen.tracer(0)
    # Nettoie l'écran
    turtle.clear()
    # Cache la 'turtle'
    turtle.hideturtle()
    # Mise à zéro du chemin
    Chemin.clear()
    # Boucle de tracé des cellules
    for i in range(Largeur) :
        for j in range(Hauteur) :
            # Calcul de la position du centre de la cellule
            (x, y) = centreHexa(i, j)
            if i % 2 == 0 or i == Largeur - 1 or j == Hauteur - 1 :
                # Tracé complet de la cellule
                traceHexa(x, y, TabLaby[i][j])
            else :
                # Tracé du bas de la cellule
                traceBasHexa(x, y, TabLaby[i][j][4])
```

```
    # Calcul et affichage du temps écoulé
    tps = round(time.time() - tps, 2)
    print("traceGrilleHexa(", Largeur, ", ", Hauteur, ") : tracé taille ", DemiTailleHexa, " en ", tps, " s", sep='')
    # Active l'animation de tracé
    screen.tracer(1)
    # Efface le symbole du chemin
    if Wcanvas != 0 :
        Wcanvas.delete(TAG_CHEMIN)
    # Trace l'entrée
    traceEntree()
    # Trace la sortie
    traceSortie()
    # Positionnement de la turtle en fonction de l'orientation
    if TK :
        Orientation = ORIENTATION.index(VarOrient.get())
    else :
        Orientation = 0
    orientation()
    return


# Calcule la position du centre d'une cellule (coordonnées turtle, axe des ordonnées orienté vers le haut)
# a_i : index en abscisse de la cellule
# a_j : index en ordonnée de la cellule
# Return : position du centre de le cellule (x, y) en pixels
def centreHexa(a_i, a_j) :
    # Décalage en abscisse : (DemiTailleHexa + DemiTailleHexa * cos(60)) * i
    x = 1.5 * DemiTailleHexa * a_i - DemiLargeurTotale
    # Décalage en ordonnée : 2 * DemiTailleHexa * sin(60) * j pour les abscisses paires, + DemiTailleHexa * sin(60) pour
    les impaires
    y = DemiTailleHexa * RACINE3_2 * (2 * a_j + (a_i % 2)) - DemiHauteurTotale
    return (x, y)


# Initialise le tableau de données du labyrinthe
# a_largeur : largeur du labyrinthe (nombre d'hexagones en abscisse)
# a_hauteur : hauteur du labyrinthe (nombre d'hexagones en ordonnée)
def initDonnees(a_largeur, a_hauteur):
    global Largeur, Hauteur, TabLaby, Sortie
    # Initialisation de la largeur
    Largeur = a_largeur
    # Initialisation de la hauteur
    Hauteur = a_hauteur
    # Position de la sortie au coin opposé à l'entrée
    Sortie = [Largeur - 1, Hauteur - 1]
    # Initialisation du tableau des données (NBR_HEXA valeurs par hexagone)
    TabLaby = [[[0 for _ in range(NBR_HEXA)] for _ in range(a_hauteur)] for _ in range(a_largeur)]
    # Affectation du type de paroi (MUR_EXT, MUR_INT ou OUVERTURE) pour les côtés
    for i in range(a_largeur) :
        for j in range(a_hauteur) :
            for k in range(NBR_HEXA) :
                # Intérieur (on positionne des murs internes partout)
                for k in range(6) :
                    TabLaby[i][j][k] = MUR_INT
                # Coin bas gauche
                if i == 0 and j == 0 :
                    TabLaby[i][j][2] = TabLaby[i][j][3] = TabLaby[i][j][4] = TabLaby[i][j][5] = MUR_EXT
                # Coin haut gauche
                elif i == 0 and j == a_hauteur - 1 :
                    TabLaby[i][j][1] = TabLaby[i][j][2] = TabLaby[i][j][3] = MUR_EXT
                # Coin bas droit
                elif i == a_largeur - 1 and j == 0 :
                    TabLaby[i][j][0] = TabLaby[i][j][4] = TabLaby[i][j][5] = MUR_EXT
                    if i % 2 == 0 :
                        TabLaby[i][j][3] = MUR_EXT
                # Coin haut droit
                elif i == a_largeur - 1 and j == a_hauteur - 1 :
```

```
                TabLaby[i][j][0] = TabLaby[i][j][1] = TabLaby[i][j][5] = MUR_EXT
                if i % 2 == 1 :
                    TabLaby[i][j][2] = MUR_EXT
            # Bord gauche
            elif i == 0 :
                TabLaby[i][j][2] = TabLaby[i][j][3] = MUR_EXT
            # Bord droit :
            elif i == a_largeur - 1:
                TabLaby[i][j][0] = TabLaby[i][j][5] = MUR_EXT
            # Bord bas
            elif j == 0 :
                TabLaby[i][j][4] = MUR_EXT
                if i % 2 == 0 :
                    TabLaby[i][j][3] = TabLaby[i][j][5] = MUR_EXT
            # Bord haut
            elif j == a_hauteur - 1:
                TabLaby[i][j][1] = MUR_EXT
                if i % 2 == 1 :
                    TabLaby[i][j][0] = TabLaby[i][j][2] = MUR_EXT
    # Initialisation des valeurs de centrage du labyrinthe
    initCentrage()
    return


# Initialisation des valeurs globales pour le centrage du labyrinthe
def initCentrage() :
    global DemiLargeurTotale, DemiHauteurTotale
    # Moitié de la largeur totale (décalage pour centrage de la grille en largeur)
    DemiLargeurTotale = (Largeur * DemiTailleHexa * 1.5 + DemiTailleHexa / 2) / 2
    # Moitié de la hauteur totale (décalage pour centrage de la grille en hauteur)
    DemiHauteurTotale = DemiTailleHexa * RACINE3_2 * (2 * Hauteur + 1) / 2
    # Décalage en abscisse
    DemiLargeurTotale -= DemiTailleHexa
    # Décalage en ordonnée
    DemiHauteurTotale -= DemiTailleHexa * RACINE3_2
    return


# Calcule si les valeurs des cellules sont nulles
# Return : True si les valeurs ne sont pas toutes nulles
#          False si elles sont toutes nulles
def cellulesNonNulles() :
    valNonNulle = False
    for i in range(Largeur) :
        if valNonNulle : break
        for j in range(Hauteur) :
            if TabLaby[i][j][6] > 0 :
                valNonNulle = True
                break
    return valNonNulle


# Algorithme de création d'un labyrinthe :
# 1 : on affecte des numéros croissants à chaque cellule (en commençant à 0)
# 2 : tant que les valeurs des cellules ne sont pas toutes nulles
#   - on supprime un mur interne (existant) aléatoirement qui sépare 2 cellules de valeurs différentes
#   - on affecte la valeur minimale des 2 cellules concernées aux cellules de valeur maximale
def creeLaby() :
    # Mise à jour des compteurs de murs
    global MursInit, MursMax, MursPlus, MursMoins
    # Initialisation pour calcul du temps de création
    tps = time.time()
    # Valeur à mettre dans la cellule
    val = 0
    # Affectation d'une valeur croissante pour chaque cellule
    for i in range(Largeur) :
        for j in range(Hauteur) :
```

```
            TabLaby[i][j][6] = val
            # On incrémente la valeur
            val += 1
    # Nombre maximum de murs (tous les murs internes possibles)
    MursInit = MursMax = nbParoisInternes(Largeur, Hauteur)
    # Pas de murs ajoutés ou supprimés
    MursPlus = MursMoins = 0
    # On a nbMurCibles murs internes destructibles (* 2 pour les 2 côtés du mur possibles)
    nbMursCibles = 2 * MursMax
    # Les valeurs des cellule devront être toutes nulles pour un labyrinthe parfait
    while cellulesNonNulles() :
        # Choix d'un mur cible à détruire
        murCible = random.randrange(nbMursCibles)
        if Debug : print('creeLaby : nbMursCibles =', nbMursCibles, ' cible =', murCible, '\n', TabLaby)
        # nbMur : nombre de murs internes
        nbMurs = 0
        # On recherche le mur cible
        for i in range(Largeur) :
            # Sortie optimisée
            if murCible < 0 : break
            for j in range(Hauteur) :
                # Sortie optimisée
                if murCible < 0 : break
                for k in range(6) :
                    # Si on trouve un mur interne
                    if TabLaby[i][j][k] == MUR_INT :
                        # Cellule voisine
                        v_i, v_j, v_k = voisin(i, j, k)
                        # Si les 2 cellules ne sont pas déjà reliées
                        if TabLaby[i][j][6] != TabLaby[v_i][v_j][6] :
                            # Si c'est le mur cible
                            if nbMurs == murCible :
                                # On le détruit
                                TabLaby[i][j][k] = OUVERTURE
                                # On détruit aussi le mur voisin (l'autre côté de la paroi)
                                TabLaby[v_i][v_j][v_k] = OUVERTURE
                                # On décrémente le nombre de murs initial
                                MursInit -= 1
                                # Valeurs minimales et maximales des 2 cellules
                                if TabLaby[i][j][6] < TabLaby[v_i][v_j][6] :
                                    valMin = TabLaby[i][j][6]
                                    valMax = TabLaby[v_i][v_j][6]
                                else :
                                    valMin = TabLaby[v_i][v_j][6]
                                    valMax = TabLaby[i][j][6]
                                # Recalcul des valeurs de cellules
                                nbMursCibles = 0
                                for m in range(Largeur) :
                                    for n in range(Hauteur) :
                                        # On affecte la valeur min aux cellules de valeur max
                                        if TabLaby[m][n][6] == valMax :
                                            TabLaby[m][n][6] = valMin
                                # Recalcul du nombre de murs cibles
                                nbMursCibles = 0
                                for m in range(Largeur) :
                                    for n in range(Hauteur) :
                                        # Calcul du nombre de murs
                                        for o in range(6) :
                                            if TabLaby[m][n][o] == MUR_INT :
                                                v_m, v_n, v_o = voisin(m, n, o)
                                                if TabLaby[m][n][6] != TabLaby[v_m][v_n][6] :
                                                    nbMursCibles += 1
                                if Debug : print('creeLaby : cible =', i, j, k, '/', v_i, v_j, v_k, '; nbMursCibles =', nbMursCibles)
                                # Optimisation de la boucle (on arrête la boucle quand on a trouvé la cible)
                                murCible = -1
                                break
                            # On incrémente pour savoir sur quel mur cible potentiel on est
```

```
                        nbMurs += 1
    # Calcul et affichage du temps écoulé
    tps = round(time.time() - tps, 2)
    print('creeLaby(', Largeur, 'x', Hauteur, '=', Largeur * Hauteur, ') : en ', tps, ' s', sep='')
    # Mise à jour du titre de la fenêtre
    majTitre()
    return


# Calcul le côté voisin d'un côté donné (l'autre face)
# a_i : abscisse de l'hexagone (indice)
# a_j : ordonnée de l'hexagone (indice)
# a_k : rang du côté (0 = NE, 1 = N, 2 = NO, 3 = SO, 4 = S, 5 = SE)
# Return : triplet de définition (i, j, k) du côté 'voisin' (k = -1 si pas de voisin)
def voisin(a_i, a_j, a_k) :
    # Sur les bords il n'y a pas de voisin
    if a_i == 0 and (a_k == 2 or a_k == 3) or a_i == Largeur - 1 and (a_k == 0 or a_k == 5) :
        # Indicateur voisin inexistant
        i = j = k = -1
    # Indices des abscisses et ordonnées de l'hexagone voisin
    else :
        # Voisin direction Nord-Est
        if a_k == 0 :
            i = a_i + 1
            j = a_j
            if i % 2 == 0 : j += 1
        # Voisin direction Nord
        elif a_k == 1 :
            i = a_i
            j = a_j + 1
        # Voisin direction Nord-Ouest
        elif a_k == 2 :
            i = a_i - 1
            j = a_j
            if i % 2 == 0 : j += 1
        # Voisin direction Sud-Ouest
        elif a_k == 3 :
            i = a_i - 1
            j = a_j
            if i % 2 == 1 : j -= 1
        # Voisin direction Sud
        elif a_k == 4 :
            i = a_i
            j = a_j - 1
        # Voisin direction Sud-Est
        else :
            i = a_i + 1
            j = a_j
            if i % 2 == 1 : j -= 1
        # Indice du mur de l'hexagone voisin
        k = (a_k + 3) % 6
    return (i, j, k)


# On avance d'une cellule à partir de la position a_pos dans la direction Orientation
# a_pos : position initiale
# Return : nombre de cellules visitées
def avance(a_pos) :
    global Chemin
    # On cherche le voisin
    (i, j, k) = voisin(a_pos[0], a_pos[1], Orientation)
    # On va jusqu'au centre suivant
    turtle.goto(centreHexa(i, j))
    # On se positionne à la nouvelle cellule
    a_pos[0] = i
    a_pos[1] = j
    # Sauvegarde de la position
```

```
    Chemin.append((a_pos[0], a_pos[1]))
    return 1


# Trace l'entrée du labyrinthe
def traceEntree() :
    # On enleve le dessin d'entrée s'il existe
    traceSymbole(Entree[0], Entree[1], TAG_ENTREE, CoulEntree)
    # Au cas où il y a superposition on retrace la sortie qui est au-dessus
    if Sortie == Entree :
        traceSortie()
    return


# Trace la sortie du labyrinthe
def traceSortie() :
    # On enleve le dessin d'entrée s'il existe
    traceSymbole(Sortie[0], Sortie[1], TAG_SORTIE, CoulSortie)
    return


# Tracé d'un symbole dans l'hexagone
# a_largeur : rang en abscisse de l'hexagone
# a_hauteur : rang en ordonnée de l'hexagone
# a_symbole : symbole pour identifier et pouvoir effacer
# a_color   : couleur du tracé
# a_text    : text du symbole (pour le type TAG_CHEMIN)
def traceSymbole(a_largeur, a_hauteur, a_symbole, a_color, a_text='') :
    if InitTurtle :
        # Position du centre
        (x, y) = centreHexa(a_largeur, a_hauteur)
        # Rayon de la sortie plus petit (il est tracé après l'entrée, cela permet de voir les 2 si superposés)
        if a_symbole == TAG_SORTIE :
            rayon = DemiTailleHexa / 3
        # Rayon
        else :
            rayon = DemiTailleHexa / 2
        # Tracés avec tag sur le canvas de Tk
        if Wcanvas != 0 :
            # On enleve le symbole s'il existe
            Wcanvas.delete(a_symbole)
            # Tracé du symbole (texte pour le chemin)
            if a_symbole == TAG_CHEMIN :
                Wcanvas.create_text(x, -y, text=a_text, fill=a_color, tags=a_symbole)
            # Carré pour entrée et sortie
            else :
                Wcanvas.create_rectangle(x - rayon, -y - rayon, x + rayon, -y + rayon, fill=a_color, tags=a_symbole)
        # Tracé sur turtle sans Tk
        else :
            # Pas de tracé
            turtle.penup()
            # On va au centre de la cellule
            turtle.goto(x, y - rayon)
            # Couleur du symbole
            turtle.pencolor(a_color)
            # Tracé
            turtle.pendown()
            # Tracé du symbole (un carré, coin vers le bas)
            turtle.setheading(0)
            turtle.circle(rayon, steps=4)
            # Pour l'entrée : on visualise la 'turtle' après la tracé de la sortie
            if a_symbole == TAG_SORTIE :
                # Position du centre
                (x, y) = centreHexa(Entree[0], Entree[1])
                # Orientation de la 'turtle'
                orientation()
                # On lève le stylo
```

```
            turtle.penup()
            # On va au centre de la cellule
            turtle.goto(x, y)
            # Affichage de la 'turtle'
            turtle.showturtle()
    return


# Test du labyrinthe : création de la fenêtre et du labyrinthe
# a_largeur : largeur du labyrinthe
# a_hauteur : hauteur du labyrinthe
def testLaby(a_largeur = LARGEUR_DEF, a_hauteur = HAUTEUR_DEF) :
    # Tracé avec interface Tk
    if TK :
        creeInterface()
    # Tracé uniquement avec turtle
    else :
        init(a_largeur, a_hauteur)
        creeLaby()
        traceGrilleHexa()
    return


#---------------------------------------------------------------------------
# Appel lors de l'exécution du programme
if __name__ == '__main__':
    # Affichage des informations
    info()
    # Démarrage du projet
    testLaby()

```