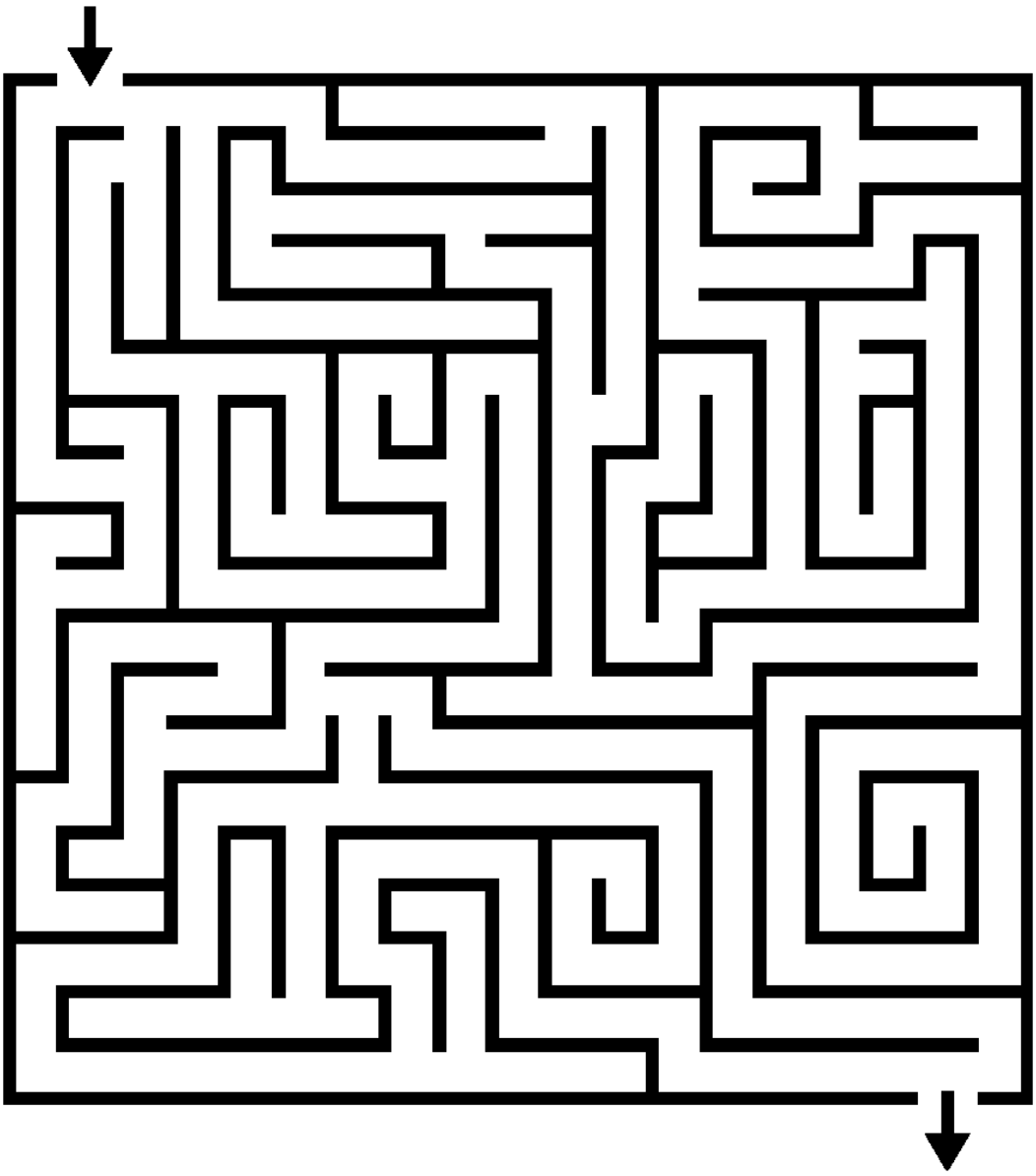
**Projet Terminale ISN – 2015/2016**



# Projet Labyrinthe

# ~

# Partie Génération

# Sommaire

# I – Présentation globale du Projet

## I – 1 : Introduction

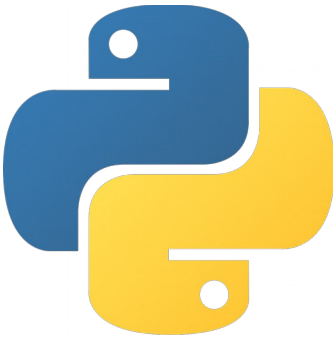
Ce programme réalisé à l'occasion du projet de fin d'année d'ISN permet de générer et résoudre des labyrinthes. Ce projet fut donc pour notre équipe l'objet d'un apprentissage sur d'un nouveau langage et d'algorithmes plus complexes nous poussant à écrire un code d'autant plus propre et plus optimisé pour produire le meilleur programme possible.

## I – 2 : Cahier des charges

Lors du développement de ce projet, il nous a fallu nous imposer un cahier des charges simples et réalisables par rapport au temps qu'il nous était donné. L'interface se devait d'être agréable à lire et facile à comprendre. La génération de labyrinthe se devait obligatoirement aléatoire. C'est à dire qu'un labyrinthe ne devait être jamais identique au suivant. Et enfin, la résolution de labyrinthe se devait de trouver le chemin le plus court possible.

## I – 3 : Technologies utilisées

Durant notre année d'ISN, nous avons appris quelques langages comme le JAVA, ou la JavaScript, avec chacun ses avantages et ses inconvénients, et c'est pourquoi on s'est tourné vers un langage nouveau pour la plupart des membres de ce projet : Le Python. On souhaitait faire un programme léger et fortement compatible avec tous les systèmes, et le langage Python nous permettait cela. Pour modifier le code source, il suffisait juste de modifier le fichier source en lui-même sans dépendre d'un quelconque IDE.



+

De plus, le Python bénéficie d'une grande variété de bibliothèques. Ainsi nous avons choisi la bibliothèque Tkinter pour l'interface graphique, d'une part par sa simplicité de déploiement et d'autre part pour sa possibilité de créer des « Canvas » c'est à dire des dessins, nécessaire pour l'affichage du labyrinthe.

## I – 4 : Répartition des tâches

Nous avons répartie équitablement le travail pour trois personnes. Pour ma part, je travaillais sur la génération aléatoire de labyrinthe, tandis que d'autre travaillait sur la résolution de labyrinthe ou bien sur l'interface graphique permettant d'afficher le labyrinthe. Avant de parler directement de la génération de labyrinthe, je voudrais d'abord introduire la notion même de labyrinthe, les différents types de labyrinthe et puis les choix que nous avons fait pour trouver le meilleur compromis entre la mise en œuvre et la difficulté. Dans un second temps, j'expliquerai le code de la génération du labyrinthe. Enfin dans un dernier temps, nous allons conclure sur le résultat de ce projet et donc les écarts par rapport à ce qu'on avait prévu ainsi que les idées d'améliorations de ce projet.

## II – Présentation de ma Partie

### II – 1 : Comprendre ce qu'est un labyrinthe

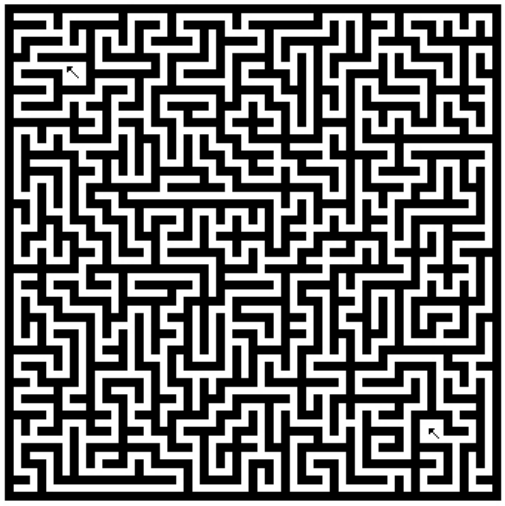
Avant d'entreprendre toute démarche, il est nécessaire de définir ce qu'est un labyrinthe. D'après le dictionnaire Larousse :

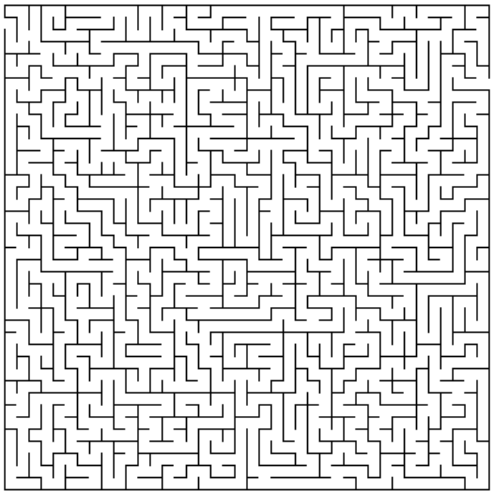
« Réseau compliqué de chemins, de galeries dont on a du mal à trouver l'issue »

Si on se tient à cette définition propre du terme labyrinthe, le but de cette génération est de créer un dédale comportant des embranchements et des impasses. On ajoutera à cette définition la notion de labyrinthe dit « parfait » c'est à dire où un chemin unique passe par toutes les cellules.
Une approche mathématique de cette définition est nécessaire pour envisager une génération procédurale de labyrinthes modernes. De plus, il faut prendre en compte l'aspect aléatoire de cette génération.

### II – 2 : Les différents types de labyrinthe

Lors de notre état de l'art sur la génération de labyrinthe, on a pu distinguer deux types de labyrinthe.





Comme on peut voire sur ces deux images, les labyrinthes peuvent être sous forme de damier, ou alors sous forme d'assemblage de murs.
Sur la figure de gauche, le labyrinthe est construit par des blocs coloriés en noir ou en blanc pour constituer une forme cohérente de labyrinthe tandis que sur la figure de gauche, ce sont des murs qui sont « cassés » pour former un chemin constituant le labyrinthe.
On a choisi la première solution, c'est à dire un labyrinthe de type « damier » car il était techniquement plus simple et visuellement plus clair.

### II – 3 : Les différents algorithmes de génération

La notion de labyrinthe existe depuis les premières civilisations modernes et a longtemps poussé l'homme à la recherche de création et de résolution de labyrinthe ; c'est pourquoi les recherches sur la génération furent très fructueuses.
Il existe de nombreux algorithmes, nous pouvons citer : *Le Growing Tree* ; *Le Hunt & Kill* et *L'algorithme d'Eller*. Pour notre projet, nous avons adopté une variante du *Hunt & Kill* que nous expliquerons par la suite.

## III – La Génération (avec code et explications)

### III – 1 : Algorithme choisi

Comme dit précédemment, nous avons adapté l'algorithme du *Hunt & Kill* car celui-ci à l'avantage d'être vite compréhensible sans prérequis mathématique et les labyrinthes produits restes cohérents et les motifs sont presque toujours originaux même à grande échelle. De plus, il consomme relativement peu de mémoire vive, et peux générer des grands labyrinthes de façon relativement rapide.

Pour la génération de labyrinthe, il faut d'abord choisir une entrée. On a choisi de façon arbitraire le milieu du terrain. C'est aussi de cette position que l'algorithme va commencer. Donc à partir de cette position initiale, il va d'abord regarder si ses cases voisines sont accessibles. Si c'est le cas, il choisit une direction aléatoirement pour s'avancer. En revanche, si il n'y a aucune issue, il va alors reculer à la case précédente. L'algorithme s'arrêtera tout naturellement lorsqu'il se trouvera sur la case initiale et qu'il n'y aura plus d'issue possible.

Cette brève explication peut encore sembler flou à première vu, mais on expliquera davantage à travers le code du programme.

### III – 2 : La création du damier et la classe Cellule

```
class Cellule (object):


    def __init__(self, x, y, size):
        self.x = x
        self.y = y
        self.couleur = "black"
        self.valeur = 0
```

Avant d'expliquer concrètement l'algorithme, il est nécessaire d'introduire la classe Cellule.

Les prochains extraits de code sont des extraits simplifiés du code original du programme. Nous avons choisis de montrer uniquement ce qui est pertinent pour l'explication de l'algorithme.

Nous pouvons voir que la classe Cellule possède quartes attributs. On peut donc retrouver chaque cellule par son abscisse x et son ordonné y car le terrain du labyrinthe est un damier. Chaque cellule possède une couleur, donc dans notre cas soit noir pour du mur, soit blanc pour du chemin ; ainsi qu'une « valeur ».
L'attribut valeur est le plus intéressant car c'est grâce à lui qu'on peut connaître l'identité de chaque cellule. Une cellule peut prendre six valeurs possibles :

- Si **valeur = -1** → la cellule est l'entré du labyrinthe, le point de départ de l'algorithme de génération du labyrinthe
- Si **valeur = -2** → la cellule est la sortie du labyrinthe, le dernier point de passage pour l'algorithme de résolution du labyrinthe
- Si **valeur = 0** → la cellule n'a jamais été visitée
- Si **valeur = 1** → la cellule précédente à celle ci est la cellule de droite
- Si **valeur = 2** → la cellule précédente à celle ci est la cellule de gauche
- Si **valeur = 3** → la cellule précédente à celle ci est la cellule d'en haut
- Si **valeur = 4** → la cellule précédente à celle ci est la cellule d'en bas

Maintenant, il nous est possible de comprendre la création du « damier », le terrain qui va contenir le labyrinthe.

```
def creation(self):
    i = 0
    o = 0

    while i<self.cote:
        while o<self.cote:
            case = Cellule(o, i)
            self.liste.append(case)
            o = o+1
        i = i+1
        o = 0
```

La fonction « creation » va donc préparer ce terrain.

La variable « cote » est la taille en nombre de cellule d'un coté du terrain, le terrain étant carré.

« liste » est le tableau situé en mémoire contenant toutes les cellules du labyrinthe. Ainsi, lors de la génération du labyrinthe, nous travaillerons avec ce tableau nommé « liste ». Chaque cellule aura donc son numéro dans la liste.

La création du terrain se fait donc linéairement avec deux boucles l'une dans l'autre.

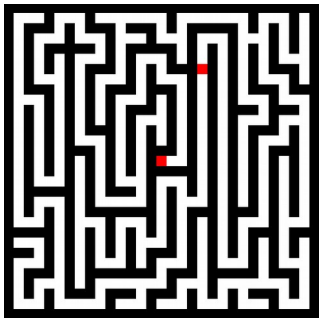
La deuxième boucle enregistre les cellules en avançant sur l'axe des abscisses mais lorsqu'elle arrive au bout de la ligne, la première boucle avance d'une ligne sur l'axe des ordonnées ; ainsi de suite jusqu'à ce que toutes les lignes soient parcourues, c'est à dire toutes les cellules enregistrées dans le tableau « liste ».

### III – 3 : Explication de l'algorithme

On va à présent pouvoir revenir sur l'explication de l'algorithme en détail après ces éclaircissements. La génération se fait donc de façon totalement procédurale et donc on va répéter chacunes des étapes jusqu'à ce que la génération soit terminée. La génération se décompose donc en trois grandes étapes composées elle même de différentes étapes:

- Orientation : On va regarder nos cellules voisines si elles sont accessibles c'est à dire si leur valeur = 0, et que ce n'est pas un mur. On enregistra toutes les directions possibles dans un tableau
- Avancer : Si au moins une direction est possible, on tire au hasard une direction dans le tableau des différentes orientations possibles. Puis on avance, d'une part en coloriant la nouvelle cellule explorée, et d'autre part en modifiant sa valeur suivant d'où on vient (voir définition de la classe Cellule, et son attribut « valeur »).
- Reculer : Uniquement si il n'y a aucune cellule voisine accessible, on lit la valeur de la cellule de notre position actuel pour savoir vers ou on doit reculer.

Ces trois étapes se répéteront tant que nous serons pas retourné sur la case initiale où valeur = -1 ET qu'il n'y ai plus aucune direction possible. Ainsi, le générateur aura forcément exploré toutes les cases disponibles du terrain et de façon totalement aléatoire.



*Exemple de labyrinthe généré aléatoirement en suivant ces procédures*

### III – 4 : Explication du code

Il est à présent temps d'expliquer le code, la traduction informatique des cette algorithme. Je détaillerai le code extrait par extrait pour faciliter la compréhension, et améliorer la lisibilité.

```
# on regarde les chemins possibles
if self.liste[pos + 2].valeur == 0 and self.liste[pos + 2].x != self.cote and self.liste[pos + 2].x != 0:
    orientation.append("droite")
if self.liste[pos - 2].valeur == 0 and self.liste[pos - 2].x != self.cote and self.liste[pos - 2].x != 0:
    orientation.append("gauche")
if self.liste[pos + (2*self.cote)].valeur == 0 and self.liste[pos + (2*self.cote)].y <= (self.cote-2) and self.liste[pos + (2*self.cote)].y >= 0:
    orientation.append("haut")
if self.liste[pos - (2*self.cote)].valeur == 0 and self.liste[pos - (2*self.cote)].y <= (self.cote-2) and self.liste[pos - (2*self.cote)].y >= 0:
    orientation.append("bas")
```

Cette extrait de code permet de voir si nos cellules voisines sont accessibles et si c'est le cas, enregistrer la direction possible dans un tableau nommé « orientation ».
On constate que peu importe la direction que l'on prends, on fait toujours des pas de deux cellules, car comme défini au préalable, notre terrain est un damier, donc il est nécessaire de faire des déplacements de deux cellules pour laisser place aux murs.

Pour les directions droite et gauche, on travaille sur l'axe des abscisses donc avec l'attribut x. Lorsque x = 0, et x = cote, correspondent les limites du labyrinthe respectivement de gauche et de droite ; tandis que pour les directions haut et bas, on travaille sur l'axe des ordonnées donc avec l'attribut y. Lorsque y ≤ cote-2 et y ≥ 0, correspondent les limites du labyrinthe respectivement du bas et du haut.

```
if len(orientation) != 0:
    alea = randint(0,len(orientation))
    alea -= 1
    # puis on deplace le curseur
    if orientation[alea] == "droite":
        pos += 2
        self.liste[pos].valeur = 1
        self.liste[pos-1].change_couleur("white")
    if orientation[alea] == "gauche":
        pos -= 2
        self.liste[pos].valeur = 2
        self.liste[pos+1].change_couleur("white")
    if orientation[alea] == "haut":
        pos += (2*self.cote)
        self.liste[pos].valeur = 3
        self.liste[pos-self.cote].change_couleur("white")
    if orientation[alea] == "bas":
        pos -= (2*self.cote)
        self.liste[pos].valeur = 4
        self.liste[pos+self.cote].change_couleur("white")
    # on colorie en blanc la case
    self.liste[pos].change_couleur("white")
```

Pour ce qui est d'avancer, on commence par vérifier la taille du tableau « orientation » contenant les directions possibles. Si il n'est pas égale à zéro, c'est qu'il y a au moins une direction possible.

En suite, on choisis un nombre aléatoirement entre zéro et le nombres de direction possible contenu dans le tableau « orientation ».

Enfin, on regarde à quelle direction correspond le nombre aléatoire dans le tableau « orientation » possible pour en suite se déplacer dans la direction voulue. On colorie les cellules que l'on parcours et leur attribut la valeur correspondante au déplacement effectué.

Sinon, si le tableau « orientation » est vide c'est à dire qu'aucune direction n'est possible, on va reculer suivant la valeur de la cellule sur la quelle on se trouve.

```
else:
    if self.liste[pos].valeur == 1:
        pos -= 2
    if self.liste[pos].valeur == 2:
        pos += 2
    if self.liste[pos].valeur == 3:
        pos -= (2*self.cote)
    if self.liste[pos].valeur == 4:
        pos += (2*self.cote)
```

Il est nécessaire de savoir quand il faut arrêter ce processus de génération :

```
if self.liste[pos].valeur == -1 and len(orientation) == 0:
    fin  = False
```

C'est en effet assez simple, on lit la valeur de notre cellule actuelle, donc valeur = -1 si on est bien sur la cellule de départ, et si taille du tableau « orientation » est vide donc si il n'y a plus aucune cellule accessible, alors fin = False c'est à dire c'est la fin de la génération, on peut quitter la boucle.

Après avoir généré un labyrinthe complet, il faut déterminer une sortie. Celle ci doit se trouver sur une cellule blanche, donc pas sur un mur, et se trouvera sur une cellule choisie aléatoirement sur une des dix premières lignes du labyrinthe.

On commence par déclarer la variable sortie à zéro. En suite, on va générer une sortie aléatoirement de la première à la dixième ligne jusqu'à ce que celle ci soit sur une cellule blanche.

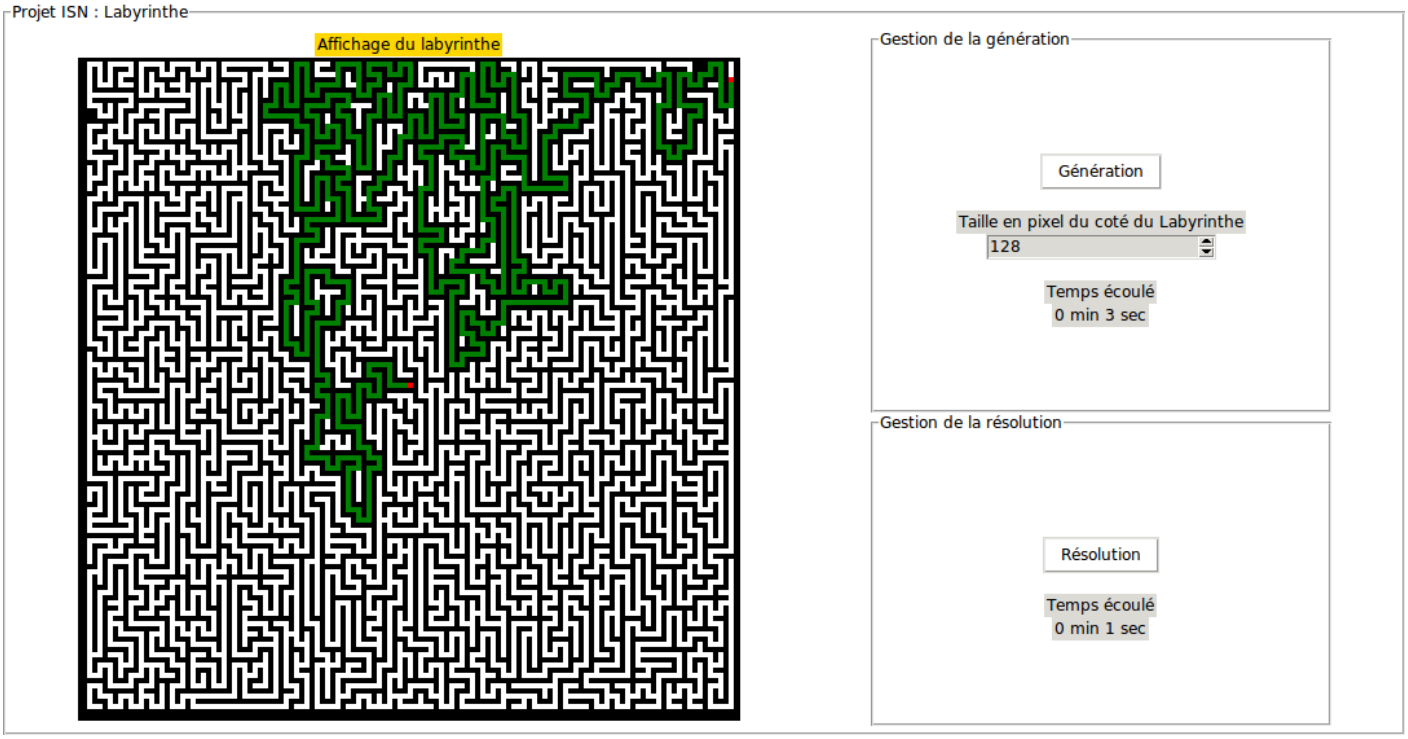
```
sortie = 0
while self.liste[sortie].couleur in "black":
    sortie = randint(self.cote,self.cote*10)
self.liste[sortie].valeur = -2
self.liste[sortie].change_couleur("red")
```

Une fois que l'on a notre sortie, on lui donne la valeur -2 et on la colorie en rouge pour pouvoir la distinguer des autres cellules.

# IV – Conclusion

## IV – 1 : Production finale

Voici la production finale de notre projet :




Ceci est le résultat des trois parties du projet. La génération et la résolution fonctionne. On peut même choisir le nombre de cellules par coté du labyrinthe. La résolution trouve bien le chemin le plus court jusqu'à la sortie et le colorie en vert. L'interface graphique affiche correctement le labyrinthe et les boutons de façon ordonnée, et permet même de voir les temps que prends la génération du labyrinthe avec sa taille souhaitée et du temps qu'il prends pour le résoudre.

Personnellement, j'ai pris beaucoup de plaisir à réaliser ce projet. J'ai appris beaucoup sur l'algorithmie en général et ainsi que les enjeux du développement de projet en groupe.

## IV – 2 : Les écarts par rapport au projet initial

Lors de la création de notre projet, nous voulions ajouter des contraintes supplémentaire pour la génération et la résolution du labyrinthe comme par exemple des points de passage obligatoire ou alors plusieurs entrées ou plusieurs sorties, mais cela n'était pas réaliste par rapport à la durée imposée du projet et de notre niveau technique.

On a préféré se cantonner à une génération et une résolution simple, mais qui fonctionne correctement de façon optimale.

## IV – 3 : Idées d'améliorations

L'ensemble de l'équipe est plutôt satisfaite du résultat mais on peut noter quelques points ou une amélioration pourrait être appréciable. On peut notamment évoquer la taille maximal du labyrinthe limité à 512 cellules de coté ; ou bien avoir la possibilité de choisir différents algorithmes de génération de labyrinthe pour pouvoir les comparer entre eux.