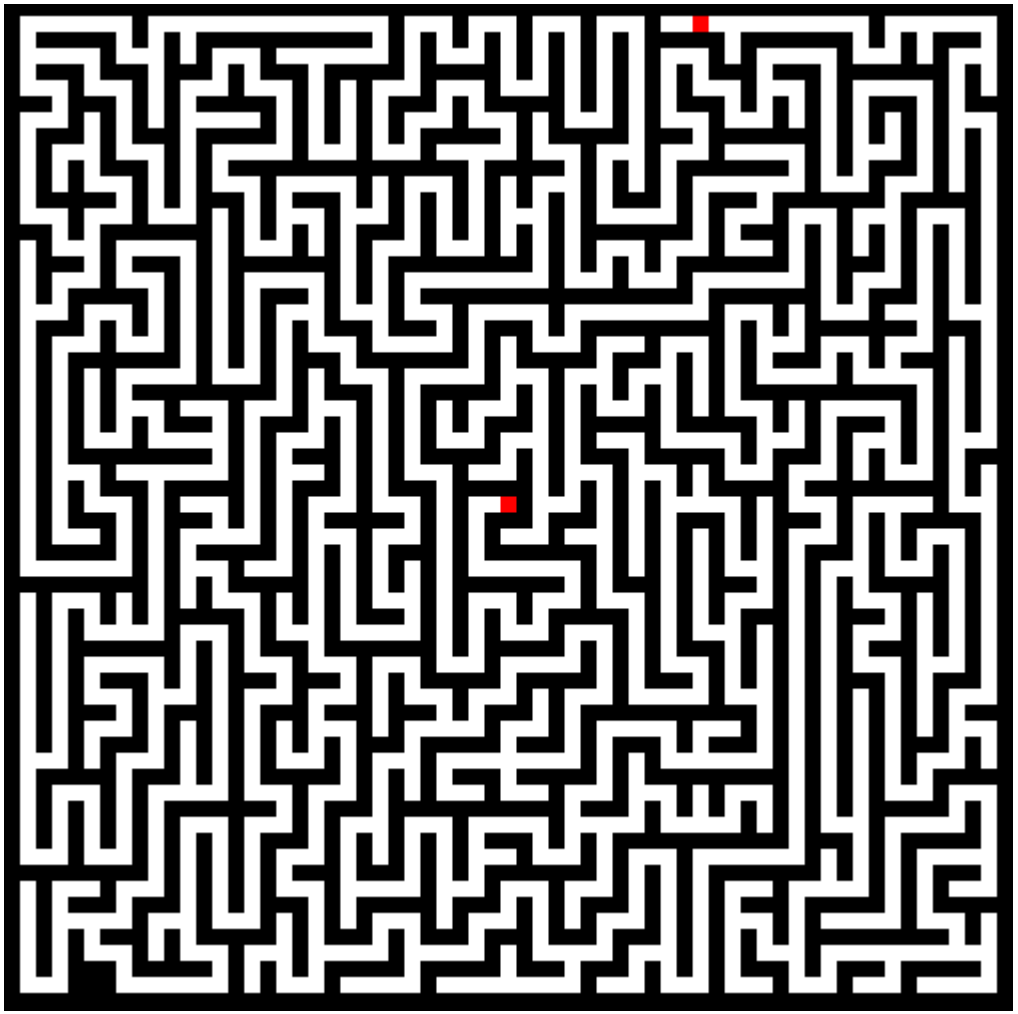
Projet ISN : Labyrinthe



Antoine Gicquel
Yohann Wyssbrod
Gaëtan Madani

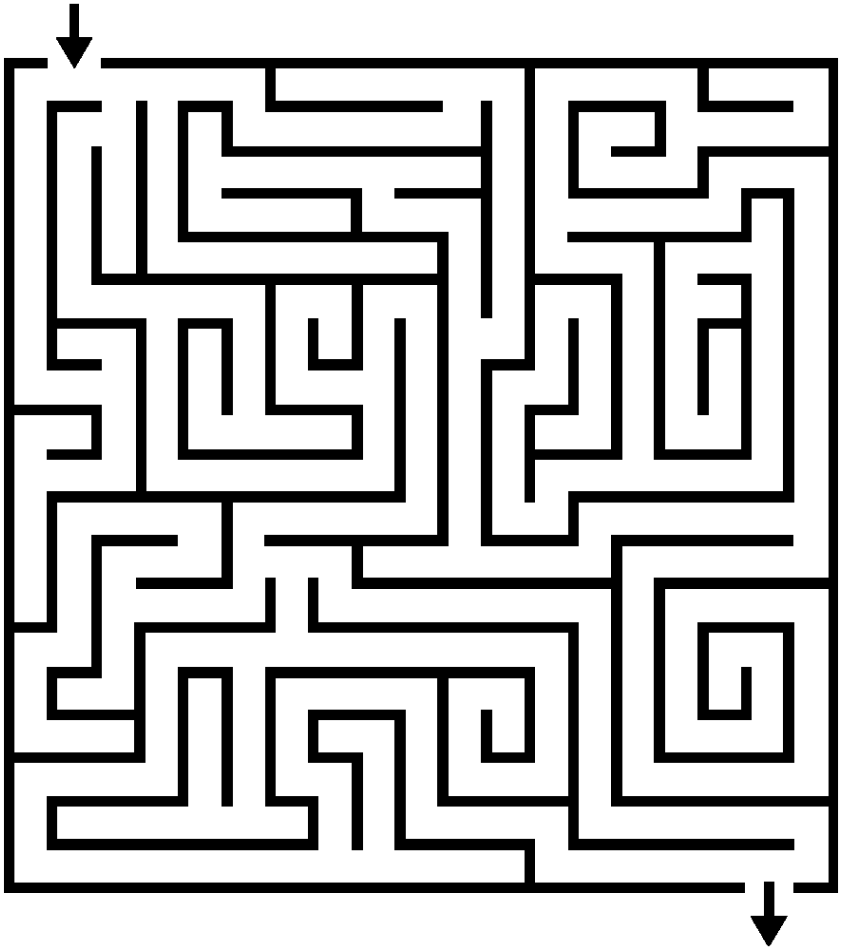
TS5 – Lycée Brequigny

Introduction

Projet ISN

Générer et résoudre des labyrinthes

→ Automatique
→ Aléatoirement



Technologies utilisées

+

Réalisé à l'aide du langage
Python et la librairie Tkinter

Répartition des tâches

Générer

Résoudre

Afficher

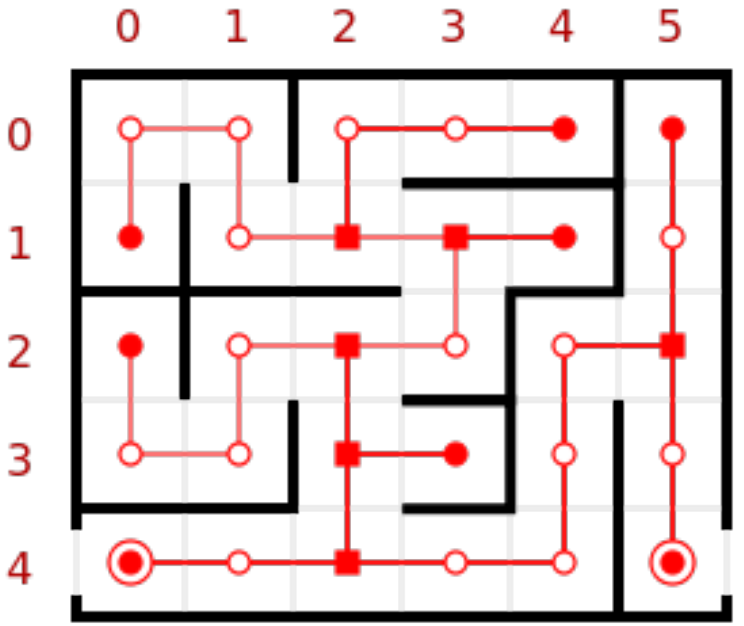
Répartition des tâches

Génération de labyrinthe :

→ Comment modéliser un labyrinthe ?

→ Par quoi commencer ?

→ Algorithmes existants

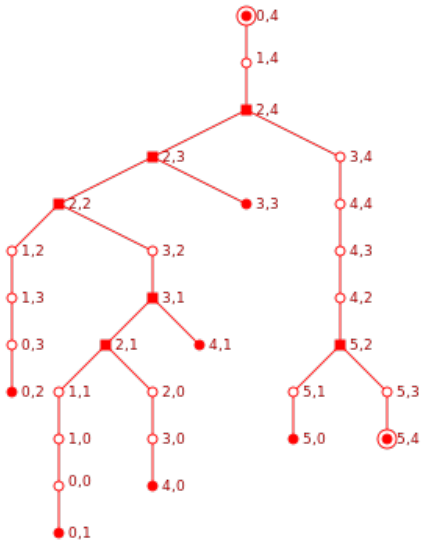


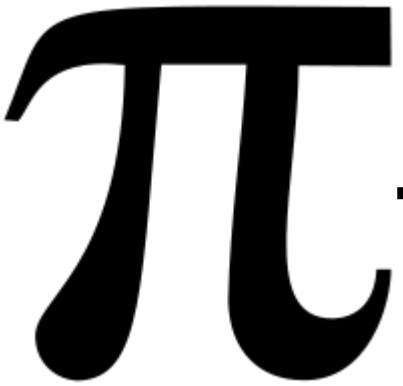
Qu'est ce qu'un « labyrinthe »

D'après Larousse :

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

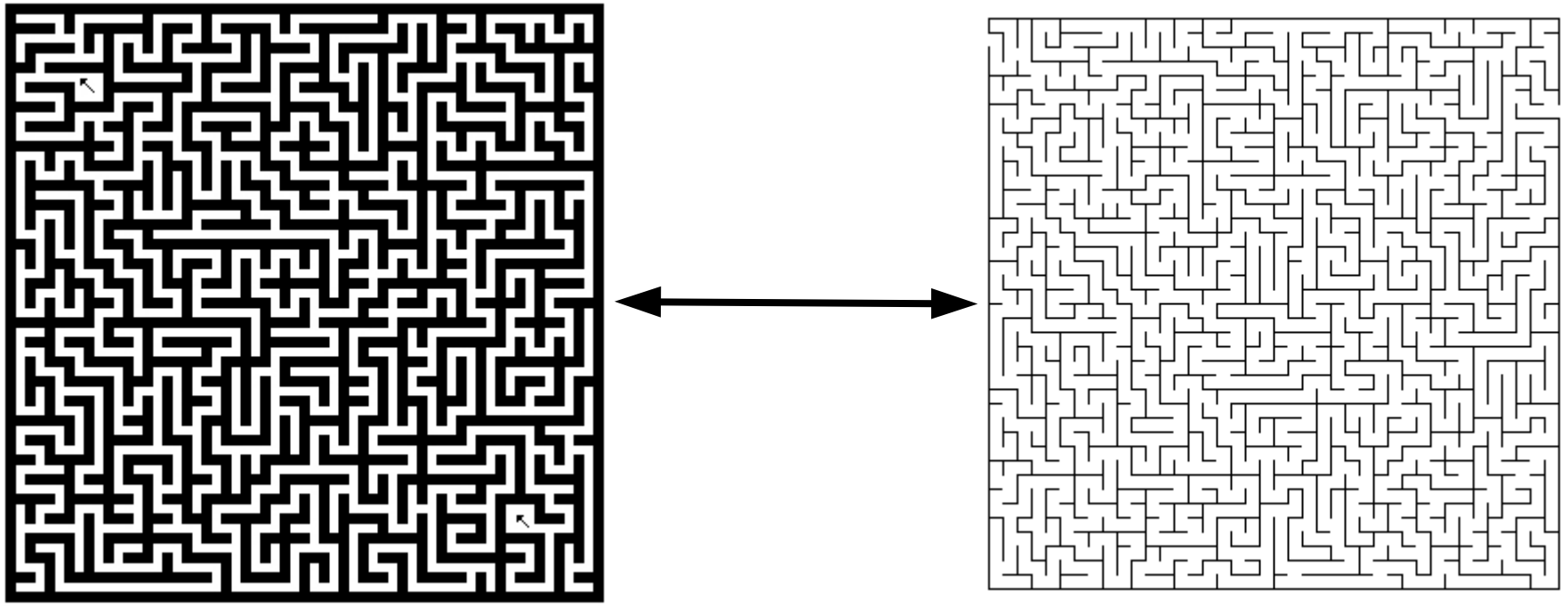
Problèmes fonctionnelles :
→ Générer un point de départ
→ Générer un dédale
→ Générer une sortie





→ Approche mathématique

Les différents types de labyrinthes



Type « damier »

Type « mur »

Les différents algorithmes

Recherches fructueuses :

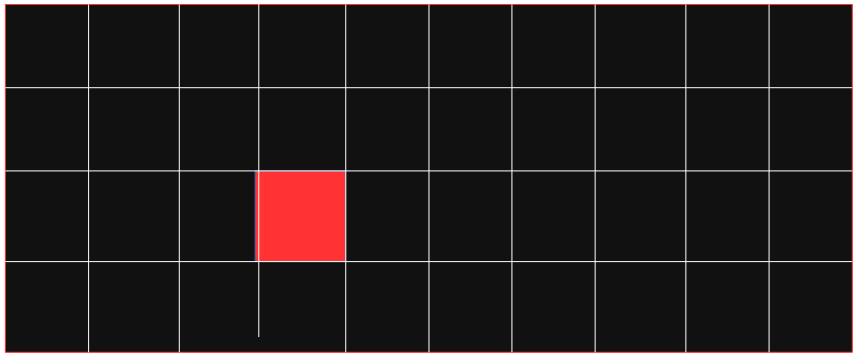
- Le Growing Tree
- Le Hunt & Kill
- L'algorithme d'Eller

→ Variante du Hunt & Kill

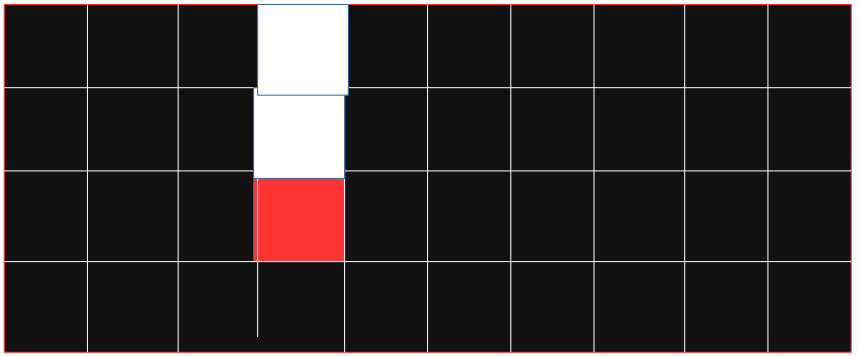
Labyrinthe très ancien !

Notre Algorithme

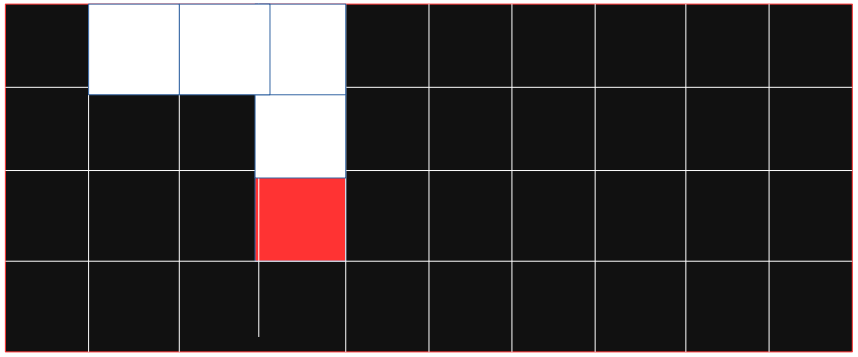
1



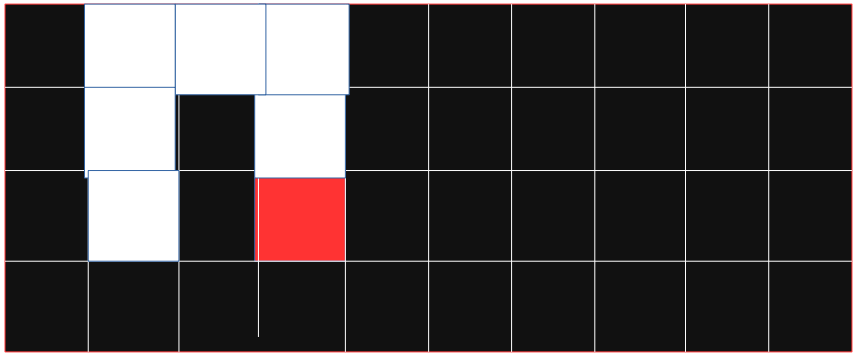
2



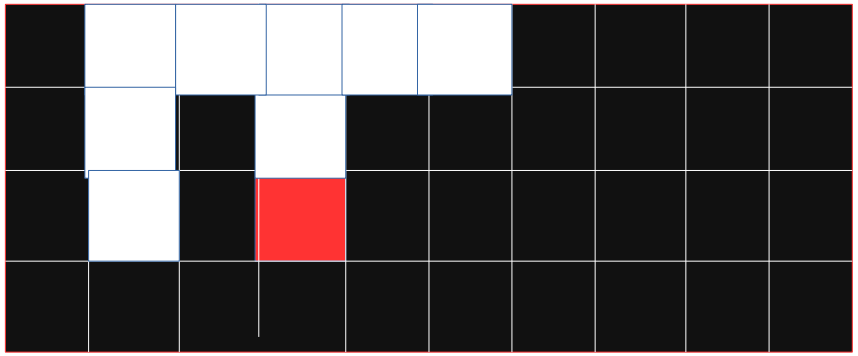
3



4



5



. . .

Explication du code : Création du « damier »

La classe Cellule

```
class Cellule (object):

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

→ Instancier avec ses coordonnées

→ Noir par défaut

→ Une valeur spécial :

- 0 = jamais visité
- -1 = départ
- -2 = sortie
- 1 = droite
- 2 = gauche
- 3 = haut
- 4 = bas

Explication du code : Création du « damier »

Création du damier

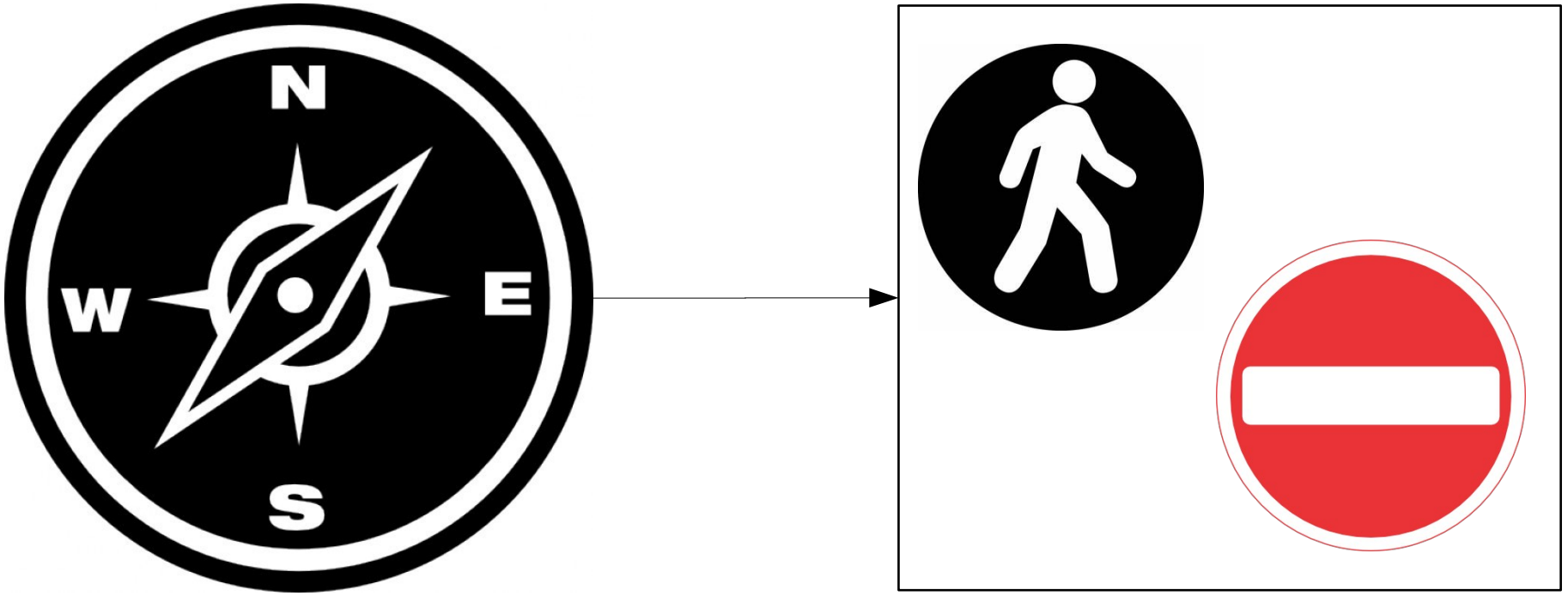
```
def creation(self):

    i = 0
    o = 0

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

→ Une boucle pour les ordonnées

→ Une boucle pour les abscisses

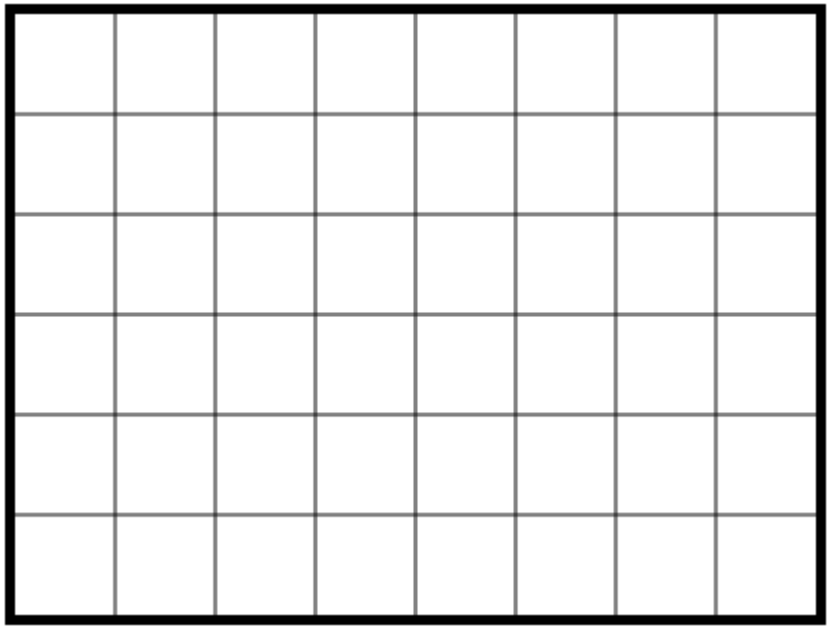
→ Cellules stockées dans tableau

Explication du code : Génération en 3 étapes



Explication du code : S'orienter

```
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")
```



On vérifie si :

- Cellule jamais visitée ?
- Cellule à l'extrémité le damier ?

Explication du code : Avancer

```
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")
```

Uniquement si une direction est envisageable

Orientation choisie aléatoirement

Explication du code : Reculer

```
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)
```

Uniquement si aucune direction est envisageable

Lecture de la valeur de la Cellule actuelle

Explication du code : Arrêter la génération

Si la cellule actuelle est celle de départ et plus aucune direction n'est envisageable

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

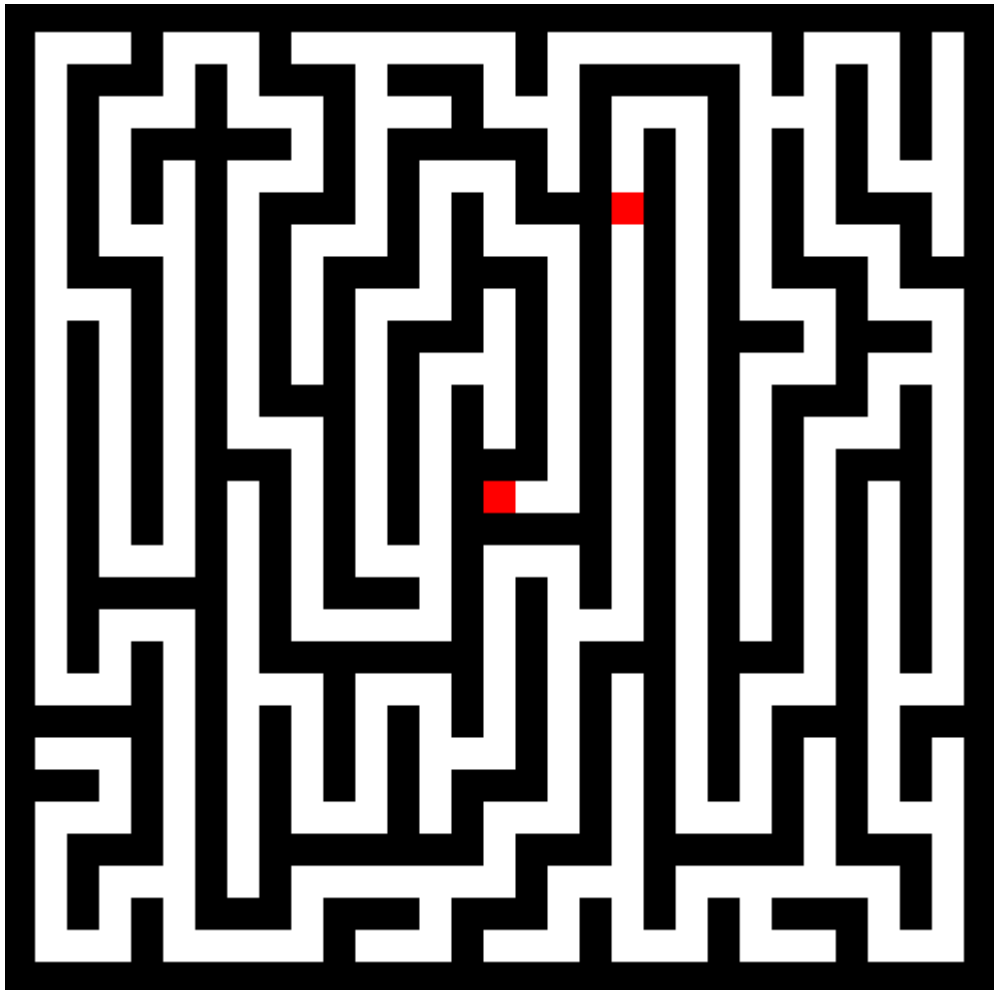
Booléen Faux → On quitte la boucle

Explication du code : Créer une sortie

Cherche une sortie sur une cellule blanche positionnée sur les dix premières lignes

```
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")
```

Conclusion



Exemple de labyrinthe généré

La génération fonctionne !

Ce que le projet m'a apporté :

→ Travail de groupe
→ Algorithmie complexe
→ Approfondissement sur le langage Python