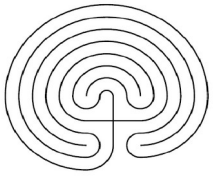
Projet 2025 (Maze Runner)



Résolution d'un labyrinthe en 2D/3D par algorithme génétique

Un labyrinthe est décrit sous la forme d'une matrice **NxN** dans laquelle les murs (obstacles) sont caractérisés par des valeurs **0**.

Sous projet 1 : Construction (aléatoire) de labyrinthe

Construction par parcours en profondeur d'abord (Depth-First Search)
[Wikipedia https://en.wikipedia.org/wiki/Maze_generation_algorithm]

Cet algorithme est une version aléatoire de l'algorithme de recherche en profondeur d'abord. Souvent implémentée avec une **pile**[1], cette approche est l'un des moyens les plus simples de générer un labyrinthe.

Initialisation : On considère l'espace pour un labyrinthe comme une grande grille ou une matrice de cellules, chaque cellule étant initialement initialisée à **0**.

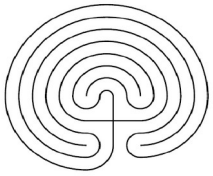
En partant d'une cellule choisie aléatoirement, l'algorithme affecte la valeur **1** à cette cellule, puis sélectionne itérativement et aléatoirement une cellule voisine qui n'a pas encore été visitée (qui contient 0) et qui n'est reliée qu'à un seul voisin visité (contenant la valeur 1), appelé aussi voisin *éligible*.

L' algorithme connecte les deux cellules en marquant la nouvelle cellule comme visitée (affecte **1** dans la cellule choisie), puis l'ajoute à la **pile** pour faciliter le retour en arrière.

L' algorithme poursuit ce processus, une cellule sans voisin *éligible* non visité étant considérée comme une impasse qui induit un *backtracking*, i.e. un dépilement de la pile.

1 Pile : structure de donnée de type premier entré, dernier sorti. On utilisera une liste python, ***append()*** correspondant à un empilement (entrée) et ***pop()*** à un dépilement (sortie)

Projet 2025 (Maze Runner)



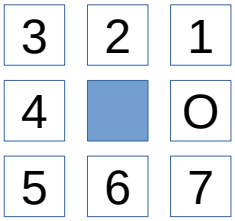
Lorsqu'il se trouve dans une impasse, l'algorithme dépile donc la **pile** (retour arrière ou *backtracking*) jusqu'à atteindre une cellule avec au moins un voisin *éligible* non visité, poursuivant ainsi la génération progressive du labyrinthe en visitant cette nouvelle cellule non visitée .

Ce processus se poursuit tant que la pile n'est pas vide.

Comme indiqué ci-dessus, cet algorithme implique une récursion en profondeur pouvant entraîner des problèmes de débordement de pile sur certaines architectures d'ordinateur. L'algorithme peut être réorganisé en une boucle en stockant des informations de retour en arrière dans le labyrinthe lui-même. Cela fournit également un moyen rapide d'afficher une solution, en commençant à un point donné et en revenant au début.

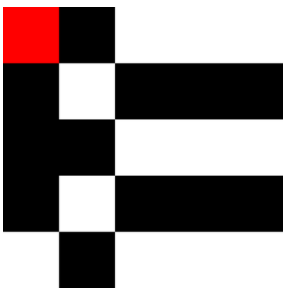
Biais de construction : les labyrinthes générés avec une recherche en profondeur d'abord ont un facteur de ramification faible et contiennent de nombreux longs corridors, car l'algorithme explore le plus possible le long de chaque branche avant de revenir en arrière.

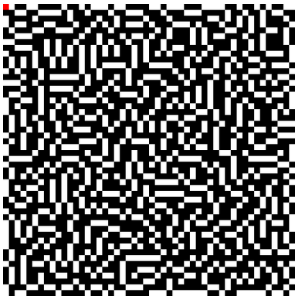
Question 1 : Implémenter l'algorithme précédent (DFS) pour générer aléatoirement des labyrinthes de taille variables (5x5, 50x50, 500x500). On considérera un voisinage à 8 voisins possibles que l'on numérotera de 0 à 7 selon la figure suivante :

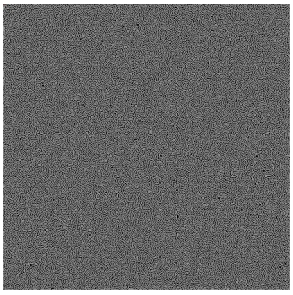


Mesurer les temps de calcul et estimer les complexité de vos algorithmes.

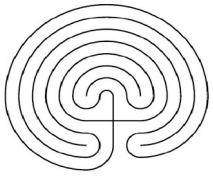
Question 2 : Considérer vos labyrinthe comme des images et visualiser les en utilisant la bibliothèque Pillow (Pil) ou matplotlib. On visualisera en rouge le but (Goal)



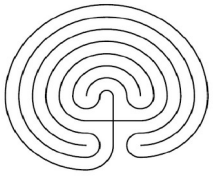




Projet 2025 (Maze Runner)



Projet 2025 (Maze Runner)



Sous projet 2 : Algorithme de Dijkstra

Carte directionnelle (**map**) : une carte directionnelle permet pour chaque cellule de la matrice représentant le labyrinthe de définir la direction à suivre pour atteindre un but localisé dans la matrice : (**i_g,j_g**) sont les indices dans la matrice qui définissent la position du but.

On considère l'algorithme de Dijkstra suivant,

Entrée : M, une matrice NxN représentant un labyrinthe généré lors de l'étape1.

Algorithme 1 (Dijkstra [2])

0 initialize **map** as a NxN Location matrix containing **None** values
1 setup the walls location in M as cells with -1 values in **map**
2 **counter** ← 0
3 goal_Location in **map** ← 0
4 other_Location in **map** ← None
5 For each location with **map** value == counter
6 For each adjacent location **L** with **None** value
7 set value to **counter**+**1** in **L**
8 increment **counter**
9 Repeat until no **None** value in **map**

On considérera que deux cellules **(i,j)** et **(k,l)** sont adjacentes si et seulement si :

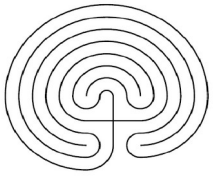
$$|i-k| \leq 1 \; ; \; |j-l| \leq 1 \; ; \; 1 \leq |i-k|+|j-l| <= 2$$

Question 1 : coder l'algorithme de Dijkstra sous forme de fonction python. On réalisera une fonction d'initialisation du labyrinthe, éventuellement en exploitant la méthode de parcours en profondeur utilisée lors de l'étape1.

Question 2 : en déduire l'algorithme permettant de construire une carte directionnelle à partir de la carte map produite par l'algorithme de Dijkstra.

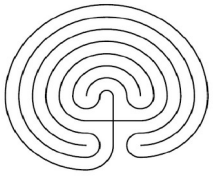
2 Dijkstra, E., "A Note on Two Problems in Connexion with Graphs," Numerische Mathematik 1, pp. 269-271 (1959)

Projet 2025 (Maze Runner)



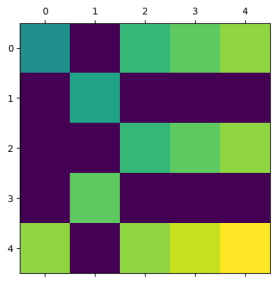
Question 3 : en déduire un algorithme qui résout le problème du labyrinthe en exploitant la carte directionnelle précédente. Vérifier sur quelques cas plausibles, que quelque soit la position du but et du point de départ, le but est atteint en utilisant votre implémentation.

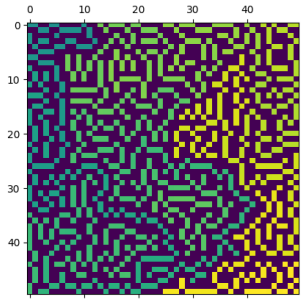
Projet 2025 (Maze Runner)

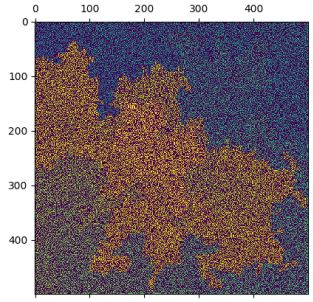


Question 4 : A partir de vos labyrinthes, utilisez l'algorithme de Dijkstra pour « colorer » la distance au but. Avec matplotlib, toute matrice peux être visualisée graphiquement de la manière suivante :

```
import matplotlib.pyplot as plt
MG # your matrix
fig, ax = plt.subplots()
ax.matshow(MG)
plt.savefig("figname.png")
plt.show()
```

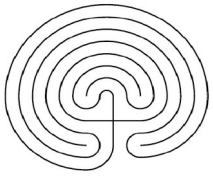






Question 5 : Colorez le chemin solution en couleur rouge dans la matrice labyrinthe et afficher sa longueur. Comment évolue la longueur moyenne de la solution (évaluez la moyenne sur une dizaine de labyrinthes) lorsque N croit (on prendra N dans {8, 16, 32, 64, 128, 256, 512})

Projet 2025 (Maze Runner)



Sous projet 3 : Résolution du problème du labyrinthe par algorithme génétique.

L'algorithme de Dijkstra permet de résoudre de manière optimale le problème du labyrinthe si celui-ci est connu a priori.

Lorsque le labyrinthe n'est pas connu du solveur, i.e. lorsque seuls le point de départ et le but sont spécifiés, et qu'un oracle permet de dire si un programme/chemin est légitime où pas (le chemin intersecte ou non un obstacle), le problème est beaucoup plus difficile.

On se propose d'utiliser le paradigme des algorithmes génétiques pour résoudre (de manière sous-optimale) ce problème.

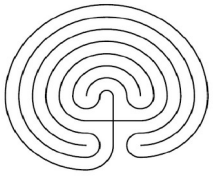
Algorithmes génétiques

Les algorithmes génétiques s'inspirent de la théorie de l'évolution, initiée par Charles Darwin au XIXeme siècle. Dans cette théorie, une population d'individus évolue grâce au mécanisme de la reproduction sexuée. Les individus les plus adaptés à leur milieu se reproduisent plus que les autres, favorisant les caractères des plus adaptés. Parfois, des mutations peuvent survenir, qui introduisent de la variété dans la population. On retrouve donc les notions suivantes :

Théorie de l'évolution	Algorithmes génétiques	Problème du labyrinthe
Individu	Une solution potentielle au problème	Un programme dans le labyrinthe
Population	L'ensemble des solutions étudiées	Les programmes
Reproduction	Croisement de deux solutions pour en produire une nouvelle	Nouveau programme obtenu par combinaison de deux autres
Mutation	Modification aléatoire d'une solution	Changement aléatoires de direction(s) dans un programme
Sélection	Élimination des solutions les moins adaptées	Élimination des programmes les plus éloignés de la solution

Le principe de fonctionnement d'un algorithme génétique est le suivant :

- Genèse : création d'une population initiale
- Évolution au cours de plusieurs générations :
    - Évaluation des individus de la population
    - Sélection d'une partie de la population
    - Reproduction par croisement de certains individus
    - Mutation de certains individus
- Sélection du meilleur individu (programme) de la dernière génération



L'avantage de ce type d'algorithme est qu'il permet de trouver une (très) bonne solution en un temps raisonnable.

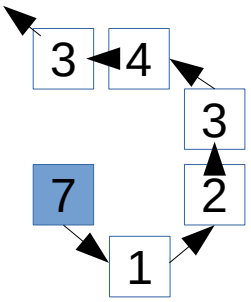
Application au problème du labyrinthes

1) Genèse

Dans le cas du labyrinthe, on considérera que les individus sont des programmes/chemins issus du point de départ (donné en entrée). On pourra utiliser l'algorithme de Dijkstra pour sélectionner un point de départ suffisamment éloigné (en terme de distance à parcourir) du but.

Un programme/chemin sera représenté sous la forme d'une séquence de taille fixe. La taille des programmes/chemins est une donnée d'entrée de l'algorithme : il conviendra de la choisir suffisamment longue relativement à la taille du labyrinthe. Chaque élément de la séquence associée à un programme/chemin contiendra une direction (entier de **0** à **7**) à suivre. Un programme/chemin spécifie donc une liste de déplacements.

Ainsi le programme/chemin **[7, 1,2, 3, 4, 3]** spécifiera un parcours :



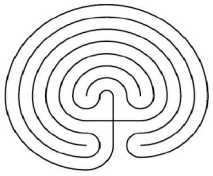
On générera de manière aléatoire un nombre **#P** (une centaine) de programmes/chemins qui constitueront notre population initiale.

2) Evolution

Durant la phase d'évolution, nous devons en premier lieu évaluer tous les individus de la population afin de les ordonner en fonction de leur « proximité » vis-à-vis d'une (bonne) solution au problème posé. Cette évaluation des programmes/chemins repose sur la définition d'une fonction dite de « fitness » (fonction que l'on peut assimiler en optimisation à une fonction dite « objectif »).

Fonction de fitness : on considérera la fonction de fitness simple suivante :

fitness(C, G, M) = **dist(endCell(C), G)** + **penalties(C,M)**,



où **C** est un programme/chemin, **G** est la position du but, **endCell(C)** retourne la position de la cellule atteinte à l'extrémité du plus long sous-chemin qui relie de manière valide cette extrémité au point de départ de **C**. **penalties(C,M)** détermine la pénalité globale associée au programme/chemin (à vous de déterminer les pénalités que vous souhaitez prendre en compte).

Sélection : la fonction de fitness permet d'ordonner les programmes/chemins de la population (ici, un fitness faible signifie un bon programme/chemin, un fitness élevé caractérisera un mauvais programme/chemin) et de sélectionner les meilleurs programmes/chemins. On utilisera un paramètre dans **[0;1]** qui spécifiera le taux de sélection **ts** (% des individus sélectionnés dans la population). Les individus non sélectionnés sont éliminés (**1** – **ts** est assimilé à un taux de mortalité).

Les individus sélectionnés pourront être amenés à se reproduire lors de la phase de reproduction.

Reproduction (Cross-over) : la phase de reproduction est effectuée conformément à l'algorithme suivant :

1) sélectionner aléatoirement deux individus *parents* dans l'ensemble des individus sélectionnés, C1 et C2.

2) sélectionner aléatoirement la position, le **cut**, qui va servir à découper les programme/chemins (génomes) des parents. On tirera aléatoirement le **cut** autour du milieu des programmes/chemins pour ne pas trop déséquilibrer les apports des deux *parents*.

3) on construit alors un nouvel individu *enfant* en assemblant les 2 sous-programmes/chemins issus des individus *parents* : **C** = **C1[:cut]**+**C2[cut:]** que l'on ajoute à la nouvelle population (génération).

4) on répète l'opération précédente (étapes 1 à 3) tant que l'on a pas atteint le nombre de reproduction souhaité.

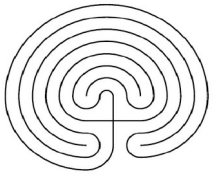
Mutation : la mutation a pour objectif de maintenir un certain niveau de diversité dans la population pour éviter que celle-ci s'atrophie du point de vue génotype.
On suit les étapes suivantes pour simuler ce mécanisme :

1) on sélectionne au hasard un individu de la population

2) on sélectionne au hasard une position sur le programme/chemin (*génome*) correspondant à l'individu sélectionne. On modifie ensuite le *gène* (la direction) en le remplaçant par un gène aléatoire (un entier de 0 à 7)

3) on répète les deux étapes précédentes tant que le nombre de mutations spécifié n'a pas été atteint. En pratique, on considérera un taux de mutation **tm** dans **[0;1]**.

Projet 2025 (Maze Runner)



Algorithme final

1) genèse : paramètres : le nombre d'individus **N**, la longueur des programmes/chemins **L**

2) évolution : paramètre : nG, nombre maximal de générations

1. i) Calcul de la fonction de fitness et tri des individus
2. ii) Sélection des individus les plus adaptés : paramètre : **ts**
3. iii) Reproduction : construction des nouveaux individus : paramètre Ne, nombre d'enfants produits. Pour que la population reste stable, on considérera **Ne=N*(1-ts)**
4. iv) Mutation : paramètre **tm**
5. on itère de 1 à 4 tant que le nombre de génération **nG** n'est pas atteint ou qu'une solution acceptable n'a pas été trouvée.

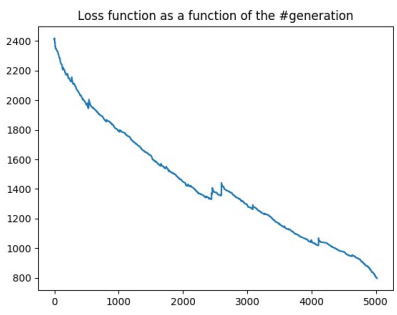
Cet algorithme n'est pas forcément très efficace pour résoudre le problème du labyrinthe, mais il illustre bien qu'une théorie « darwinienne » de l'évolution est malgré tout assez efficace sur le temps long pour résoudre des problèmes complexes en se basant sur des règles simples et une fonction de « fitness » judicieuse !

Question 1 : programmer et tester ce type d'algorithmes sur vos labyrinthes. Observer la décroissance (ou non décroissance) de la fonction de « loss ». Visualisez vos résultats

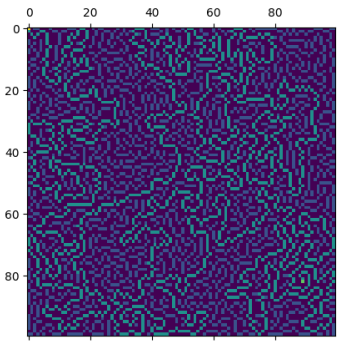
Question 2 : pour améliorer la convergence et pour éviter que la population reste « agglutinée » autour d'un minimum local bloquant (ne conduisant pas à la solution), on utilisera un mécanisme de dépôt de « phéromone » qui permettra d'indiquer qu'une branche explorée du labyrinthe conduit à une impasse. Cela revient à virtuellement couper la branche du labyrinthe comme si on réintroduisait un mur pour barricader l'accès à la branche. Ainsi, la génération suivante ne sera plus « autorisée » à explorer cette branche « morte ».

NB : faites preuves d'astuces pour améliorer (assurer) la convergence de votre algorithme.

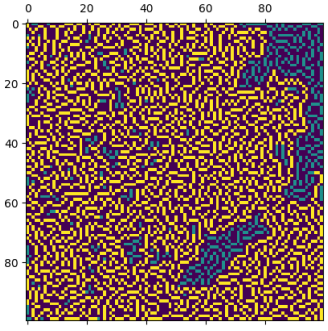
Résultats obtenus pour un labyrinthe 100x100



Loss Function



Solution



Exploration