ANNÉE 2022

3250.3 Intelligence artificielle I
TP4 – Rapport technique – ISC3il-b

Algorithmes génétique - Labyrinthe

Résoudre un problème de labyrinthe en utilisant Python ainsi que les algorithmes génétiques, avec le framework DEAP.

Étudiants participant à ce travail :

Nicolas Aubert, ISC3il-b

Présenté à :

Fabrizio Albertetti

Restitution du rapport : **27.11.2022**

Période : **2022**

École : **HE-Arc, Neuchâtel**

haute école
neuchâtel berne jura
arc ingénierie
www.he-arc.ch

Table des matières

1 - Introduction

1.1 - Contexte

Ce travail pratique rentre dans le cadre du cours *3250.3 Intelligence artificielle I*, dispensé par M Albertetti. Il nous a été demandé de réaliser un projet permettant de mettre en œuvre nos connaissances concernant les algorithmes génétiques, acquises lors de ce cours.

1.2 - Description du problème

Le but de ce travail consiste à implémenter un algorithme génétique ayant pour objectif de résoudre un labyrinthe. Cette implémentation doit être réalisée avec le langage de programmation *Python*, ainsi que le framework *DEAP*.

Un labyrinthe est représenté par une matrice de dimensions NxN, où chacune des cellules peut prendre pour valeur 0 ou 1.

- 0 représente une cellule vide, dans laquelle l'algorithme peut se déplacer,
- 1 représente une cellule pleine, correspondant à un mur, dans laquelle l'algorithme ne peut pas se trouver.

L'entrée du labyrinthe (cellule initiale) se trouve toujours dans le coin sur supérieur gauche, aux coordonnées (0, 0). L'algorithme peut se déplacer dans quatre directions uniquement : haut, gauche, bas, droite.

2 - Réalisation

2.1 - Définition d'un gène

Un gène peut prendre les 4 valeurs suivantes :

- 0 = Déplacement vers la gauche,
- 1 = Déplacement vers la droite,
- 2 = Déplacement vers le haut,
- 3 = Déplacement vers le bas.

2.2 - Définition d'un chromosome

Un chromosome compose un individu ; une analogie avec l'ADN qui compose l'humain peut être utilisée afin de mieux visualiser cette notion.

Un chromosome est une suite, d'une certaine longueur, de gènes. À partir d'un chromosome et d'une position initiale, un chemin peut être généré, menant à une position finale. Cela est fait dans la fonction de ***`compute_complete_valid_path`*** décrite ci-dessous.

2.2.1 - Taille d'un chromosome

Le labyrinthe menant à un chemin de taille maximale est la forme suivante :

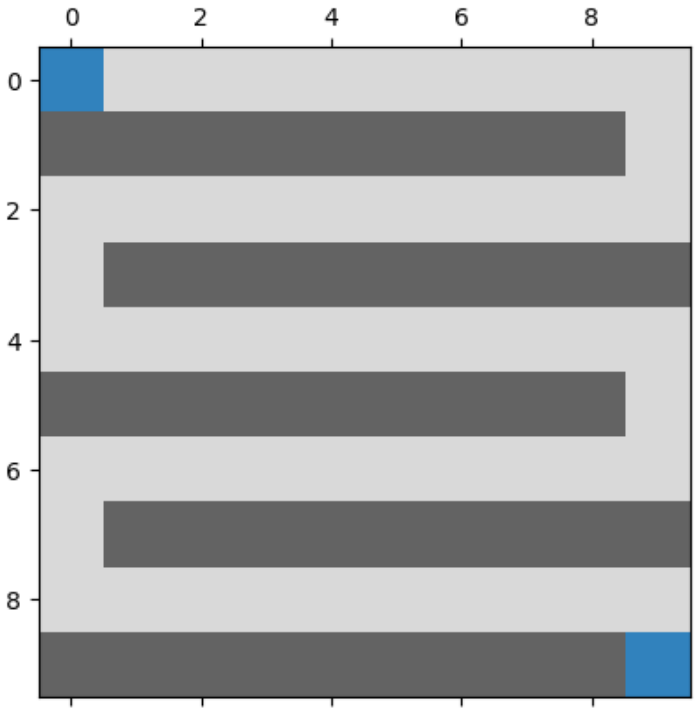

Figure 1 - Labyrinthe menant au chemin le plus long

Le chemin le plus long possède donc une longueur de ***(width * length) / 2***. La taille des chromosomes dans cette implémentation se base sur cette formule.

2.3 - Récupération du chemin à partir d'un chromosome (compute_complete_valid_path)

Afin de générer le chemin lié au chromosome, il suffit, pour chaque gène, d'appliquer le déplacement à la position actuelle, et d'ajouter celle-ci dans une liste pour garder une trace des positions empruntées.

Lors de la première version de mon implémentation, si appliquer le gène actuel résidait en une position illégale (mur, en dehors du tableau), celle-ci n'était juste pas appliquée. Une autre approche a été mise en place par la suite, afin de prévenir les culs-de-sac.

2.3.1 -Prévenir les culs-de-sac

Si l'application d'un gène génère une position illégale, le gène actuel est alors transformé en un autre déplacement, jusqu'à ce que celui-ci donne une position légale. Dans ce contexte, une position légale :

- Reste à l'intérieur du labyrinthe,
- Se trouve dans une cellule vide (valeur à 0),
- Ne revient pas arrière : elle ne se trouve donc pas sur la même position que l'avant-dernière cellule empruntée.

Si aucun des quatre types de gènes ne permet d'obtenir une position légale, alors la position actuelle est un cul-de-sac, et sera considérée comme un mur (fictif) jusqu'à la fin de la génération du chemin, pour ce chromosome uniquement.

2.4 - Récupération du chemin le plus court (compute_subpath)

La première étape consiste à retirer les positions dupliquées consécutives : par exemple, si une partie du chemin est composé de ***[..., (1, 2), (1, 2), (1, 3), ...]***, il deviendra alors ***[..., (1, 2), (1, 3), ...]***.

Par la suite, une fois la redondance éliminée, la fonction regarde si la position souhaitée (***target***) est contenue dans le chemin. Si c'est le cas, elle retourne le chemin le plus court allant de la position initiale ***(0, 0)***, jusqu'à la position souhaitée (***target***)

Si la ***target*** ne se trouve pas dans le chemin, le même raisonnement est appliqué, mais en prenant comme cellule finale la position la plus proche de la ***target*** dans le chemin (distance euclidienne).

2.5 - Fonction de sélection

La fonction de sélection permet d'obtenir les meilleurs individus dans la population. À des fins d'optimisations, la fonction de sélection ***tools.selTournament()***, fourni par *deap* a été utilisée, avec une taille de tournoi (***tournsize***) valant 3.

2.6 - Fonction de crossover

La fonction de crossover permet, à partir de deux parents, d'obtenir deux enfants, dont les chromosomes sont basés sur ceux de leurs géniteurs. La transmission des chromosomes se fait en un seul point. Cela signifie que les chromosomes des parents seront divisés en deux : le chromosome du premier enfant sera composé de la première partie du chromosome du premier parent ainsi que de la deuxième partie du chromosome du deuxième parent. Le chromosome du deuxième enfant sera composé de la deuxième partie du chromosome du premier parent ainsi que de la première partie du chromosome du deuxième parent.

Voici un exemple afin d'illustrer ce paragraphe complexe, dans lequel le chromosome du premier parent est ***[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]***, celui du deuxième est ***[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]***.

Les chromosomes des enfants seront donc : ***[1, 1, 1, 1, 1, 0, 0, 0, 0, 0]*** et ***[0, 0, 0, 0, 0, 1, 1, 1, 1, 1]***.

La probabilité de crossover pour deux individus a été fixée à 0.5 (50%).

2.6.1 -Élitisme

Afin de préserver les meilleurs individus obtenus jusqu'à présent dans l'algorithme, un paramètre d'élitisme a été mis en place, ayant comme valeur 0.01 (1%). Cela signifie qu'à chaque génération, le 1% des meilleurs individus ne subiront pas de crossover ni de mutation.

2.7 - Fonction de mutation

La fonction de mutation permet d'apporter des modifications aléatoires aux gènes d'un chromosome.

Chaque individu possède un certain pourcentage de mutation à chaque génération. Ce taux a pour valeur : 0.8, ce qui correspond à 80%.

Pour muter un individu, chaque gène d'un chromosome est passé en revue, et muté en fonction du taux de mutation individuel d'un gène, valant 0.1 (10%).

Ces mutations modifient, par palier de 1, certains gènes. Par exemple, un gène possédant la valeur 2 pourra devenir 1 ou 3.

Les mutations sont cycliques, cela signifie que si un gène ayant pour valeur 3 subit une mutation de +1, sa valeur deviendra alors 0. Il en va de même pour un gène de valeur 0 subissant une mutation de -1 (deviendra 3). Pour respecter cette règle, il suffit d'appliquer un modulo 4 au gène muté.

2.8 - Fonction de fitness

La fonction de fitness ***compute_fitness()*** permet de calculer le fitness d'un individu, qui peut être comparé à un score, permettant de classer les individus. Dans cette simulation, plus le fitness est faible, meilleur est l'individu.

La première étape consiste à construire le chemin emprunté par l'individu, basé sur son chromosome, à l'aide de la fonction ***compute_subpath()***.

Si l'individu atteint la cible à un moment dans son chemin, son fitness est alors le nombre d'étapes afin de s'y rendre. Comme précisé précédemment, si la cible est atteinte, la fonction ***compute_subpath()*** retourna le chemin le plus court menant à cette cellule. Cela signifie que la dernière cellule du chemin est alors la cellule cible. Le nombre d'étapes est donc la longueur du chemin.

Si l'individu n'atteint pas la cible, la dernière cellule de son chemin sera la cellule la plus proche de la cellule cible (***closest_cell***). Son fitness est alors la longueur du chemin, à laquelle est additionnée la distance euclidienne séparant cette cellule de la cible. Cette distance est multipliée par un certain facteur, afin de ne pas fausser le fitness.

3 - Résultats

Ces tests ont été effectués sur un échantillon de 100 simulations. Les chemins affichés dans ces différents graphiques mènent tous à une solution, se trouvant dans le coin inférieur droit.

Les durées pour chaque test, en fonction des dimensions, ont respecté les consignes demandées, résumées dans le tableau suivant :

Taille	Temps max.
10x10	10s
15x15	15s
20x20	30s
30x30	60s
40x40	90s
30x30*	180s
30x30**	10s

Figure 2 - Table des durées en fonction des dimensions

3.1 - Labyrinthes aléatoires (ouverts)

3.1.1 -Grille 10x10

Voici la grille, de taille 10x10, sur laquelle les tests ont été effectués.

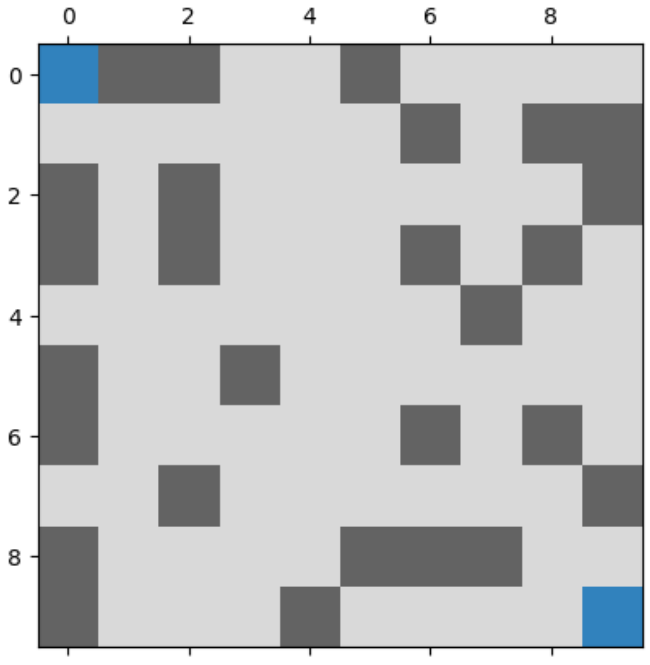

Figure 3 - Grille 10x10 aléatoire utilisée pour les tests

Voici un exemple de solution pour cette grille, qui n'est pas forcément optimale :

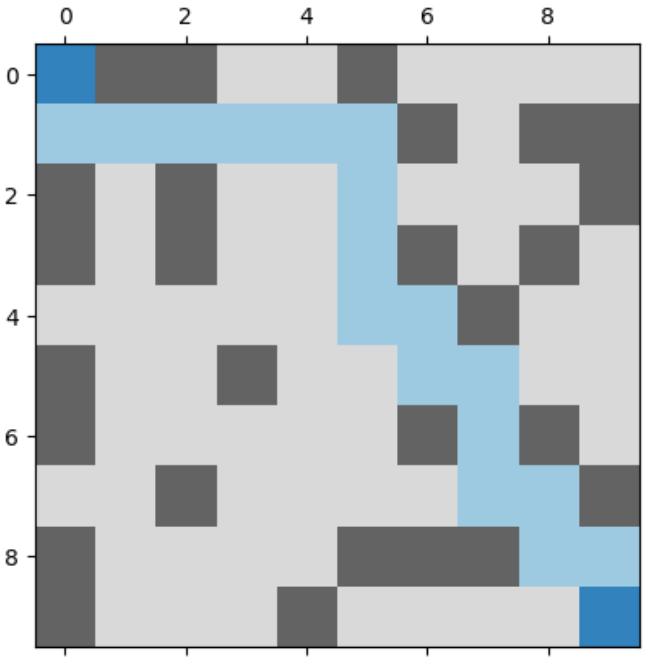

Figure 4 - Exemple de solution pour une grille 10x10 aléatoire

Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

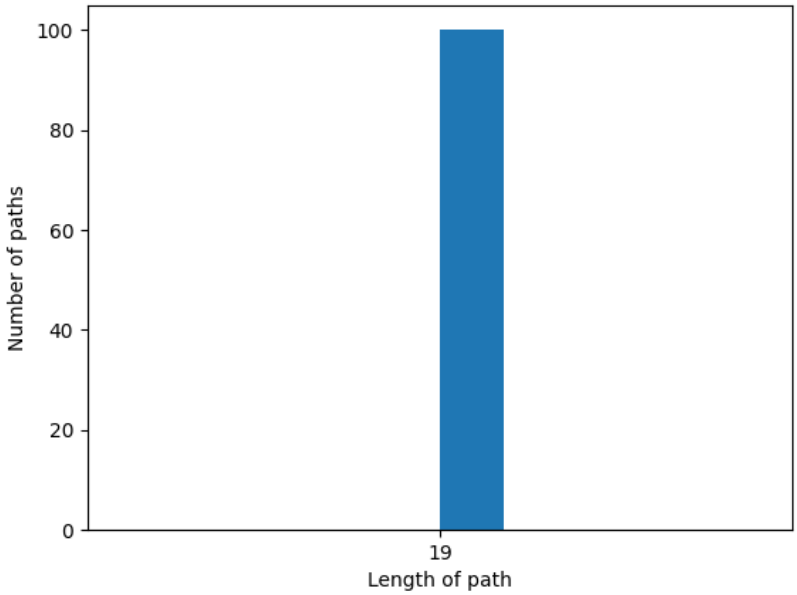

Figure 5 - Nombre de chemins en fonction de leur longueur pour une grille 10x10 aléatoire

Nous pouvons constater que, pour un labyrinthe de faible taille, l'algorithme est consistant, et toutes les simulations parviennent à la même solution, qui est alors probablement la meilleure.

3.1.2 -Grille 20x20

Voici la grille, de taille 20x20, sur laquelle les tests ont été effectués.

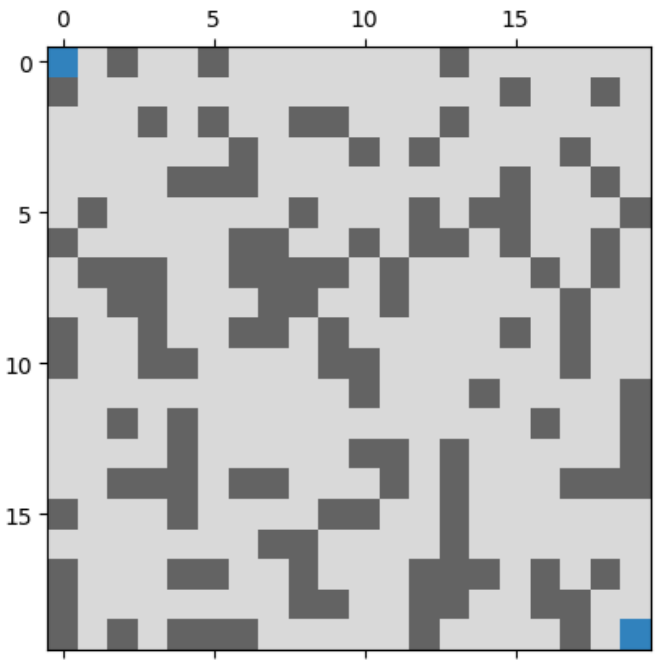

Figure 6 - Grille 20x20 aléatoire utilisée pour les tests

Voici un exemple de solution pour cette grille, qui n'est pas forcément optimale :

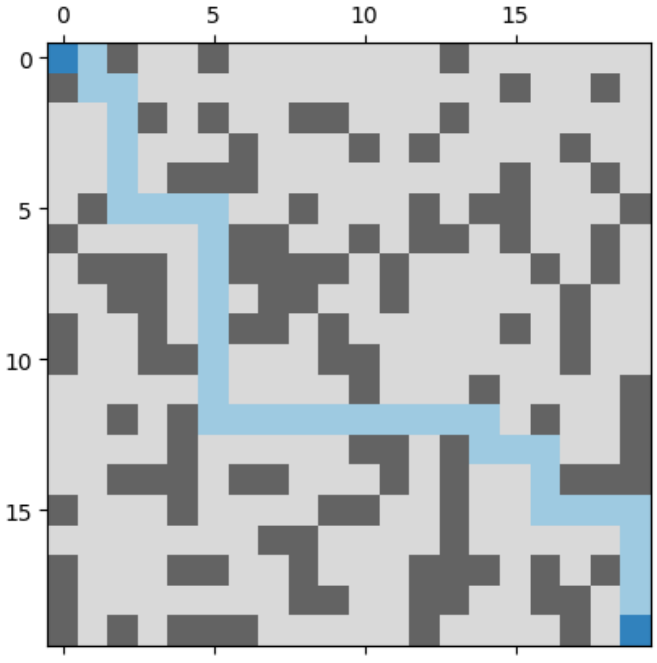

Figure 7 - Exemple de solution pour une grille 20x20 aléatoire

Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

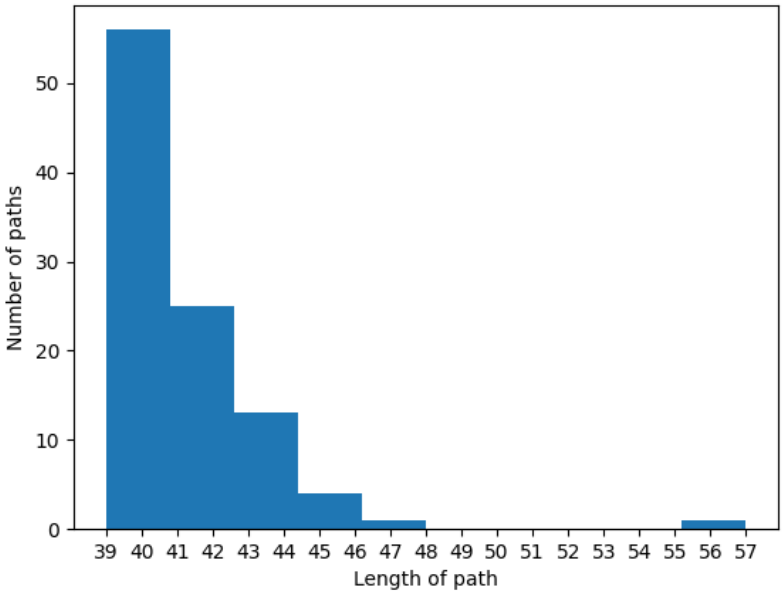

Figure 8 - Nombre de chemins en fonction de leur longueur pour une grille 20x20 aléatoire

Avec un labyrinthe de dimensions 20x20, les simulations ne convergent pas toutes vers la même solution. Cependant, une grande majorité s'entend à dire que la solution optimale est un chemin de taille 39.

3.1.3 -Grille 30x30

Voici la grille, de taille 30x30, sur laquelle les tests ont été effectués.

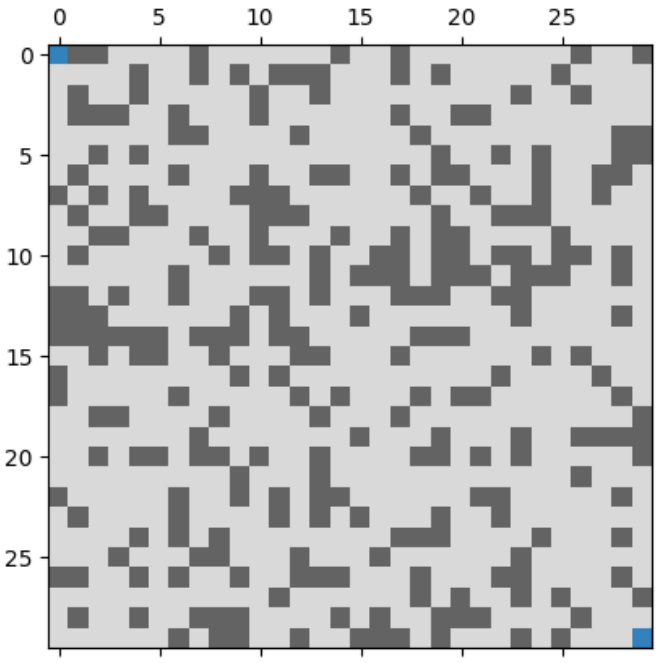

Figure 9 - Grille 30x30 aléatoire utilisée pour les tests

Voici un exemple de solution pour cette grille, qui n'est pas optimale :

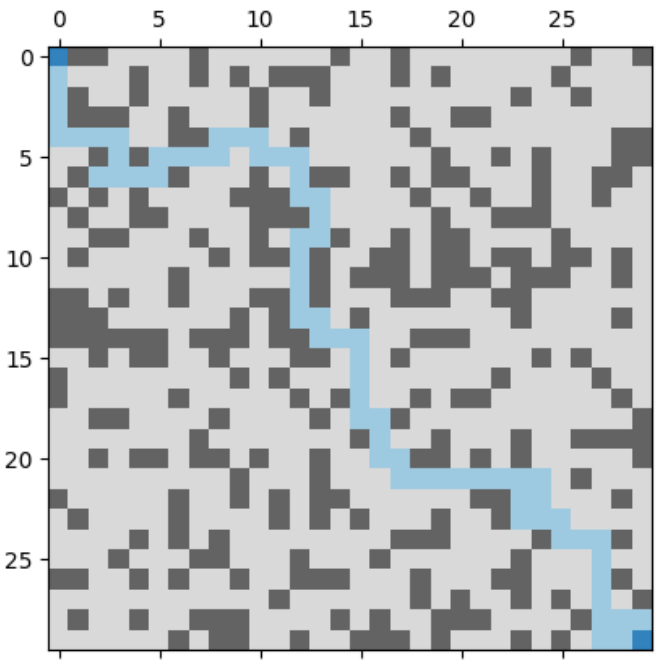

Figure 10 - Exemple de solution pour une grille 30x30 aléatoire

Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

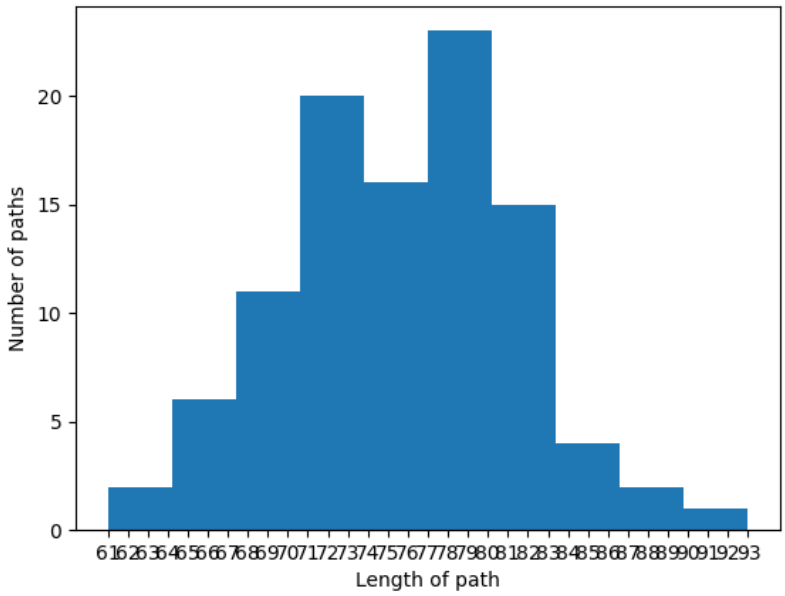

Figure 11 - Nombre de chemins en fonction de leur longueur pour une grille 30x30 aléatoire

Contrairement aux deux premiers graphiques, avec un labyrinthe de dimensions 30x30, l'algorithme ne permet pas d'approcher une solution optimale à tous les coups. Avec plus d'individus (de simulations), ce graphique ressemblerait certainement à une gaussienne.

3.2 - Labyrinthes aléatoires réalistes (fermés)

3.2.1 -Grille 10x10

Voici le labyrinthe réaliste, de dimensions 10x10, qui a été utilisé pour réaliser ces tests.

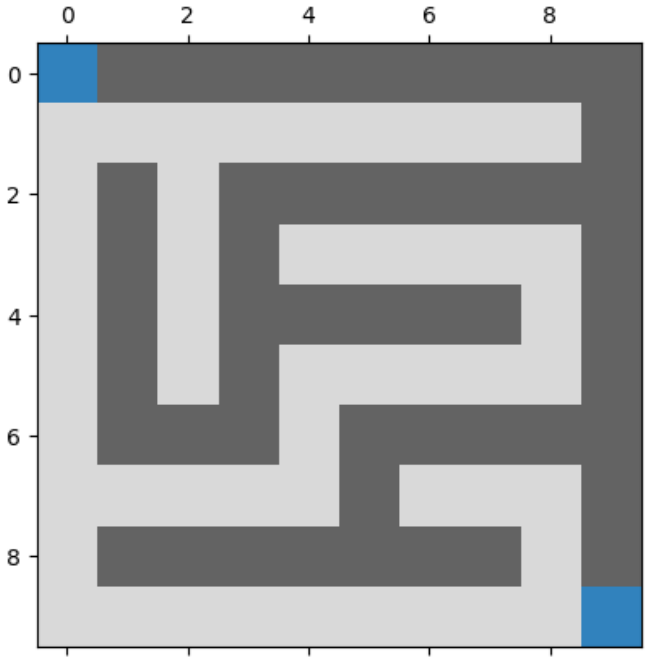

Figure 12 - Grille 10x10 réaliste utilisée pour les tests

Voici la solution optimale pour ce labyrinthe :

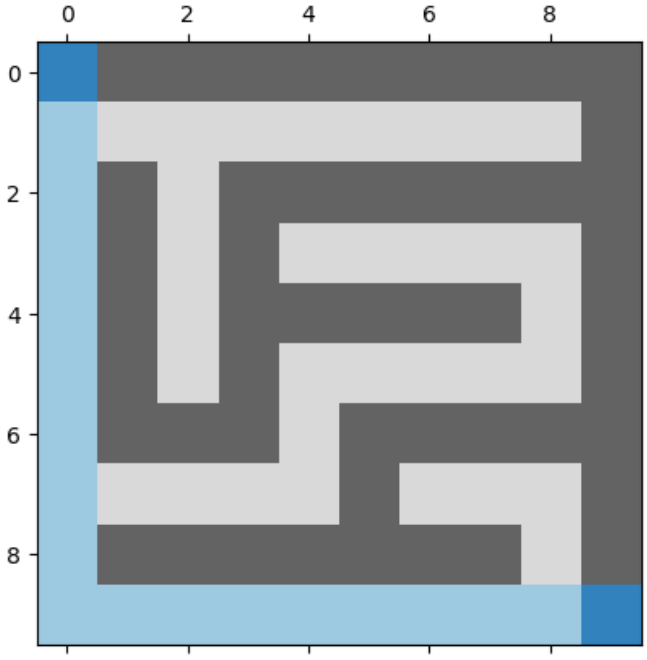

Figure 13 - Exemple de solution pour une grille 10x10 réaliste

Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

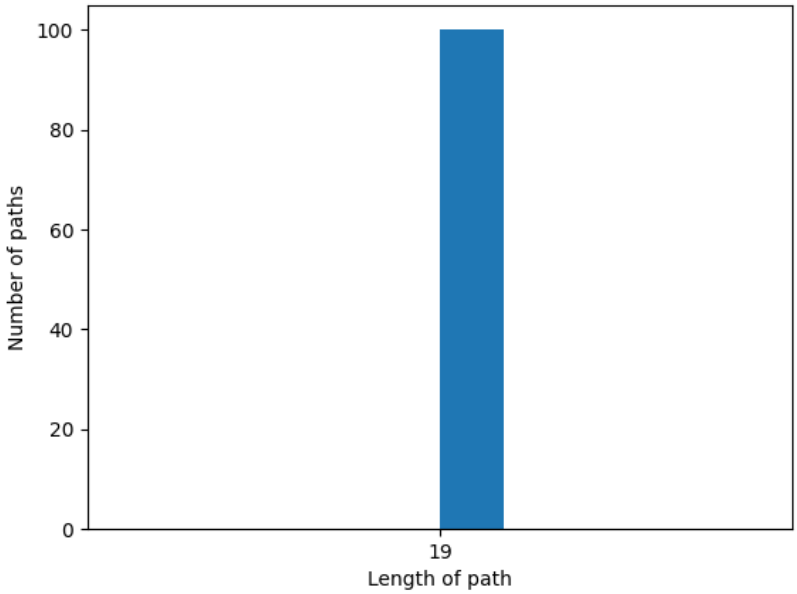

Figure 14 - Nombre de chemins en fonction de leur longueur pour une grille 10x10 réaliste

Grâce à la prévention des culs-de-sac, l'algorithme donne des résultats constants, même avec des durées bien plus faibles que celles conseillées. Avec de telles dimensions, l'algorithme permet de trouver la solution optimale en moins d'une demi-seconde.

3.2.2 -Grille 20x20

Voici le labyrinthe réaliste, de dimensions 20x20, qui a été utilisé pour réaliser ces tests.

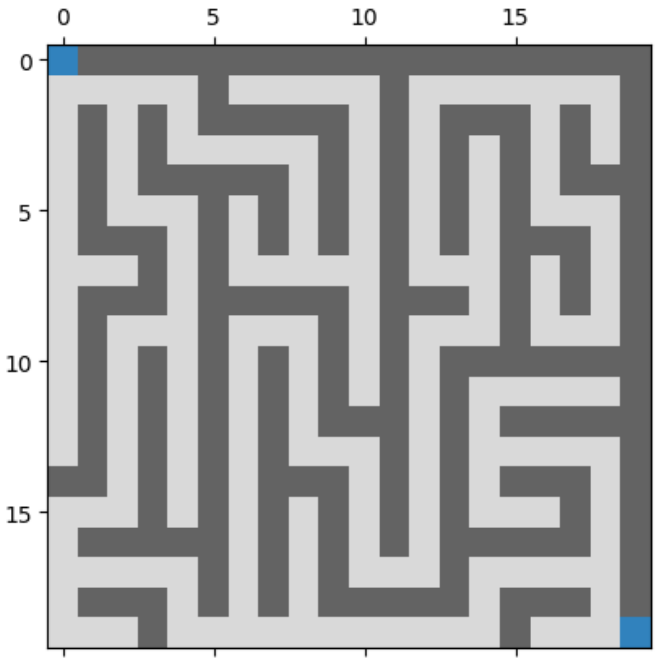

Figure 15 - Grille 20x20 réaliste utilisée pour les tests

Voici la solution optimale pour ce labyrinthe :

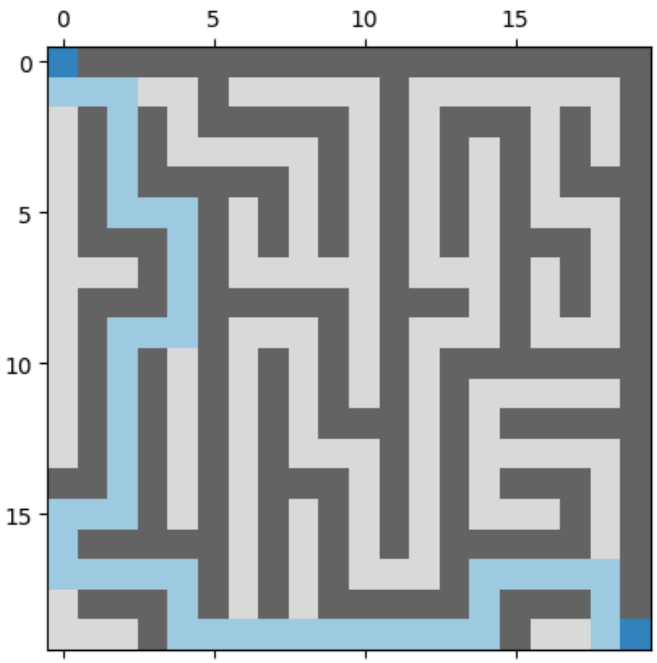

Figure 16 - Exemple de solution pour une grille 20x20 réaliste

Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

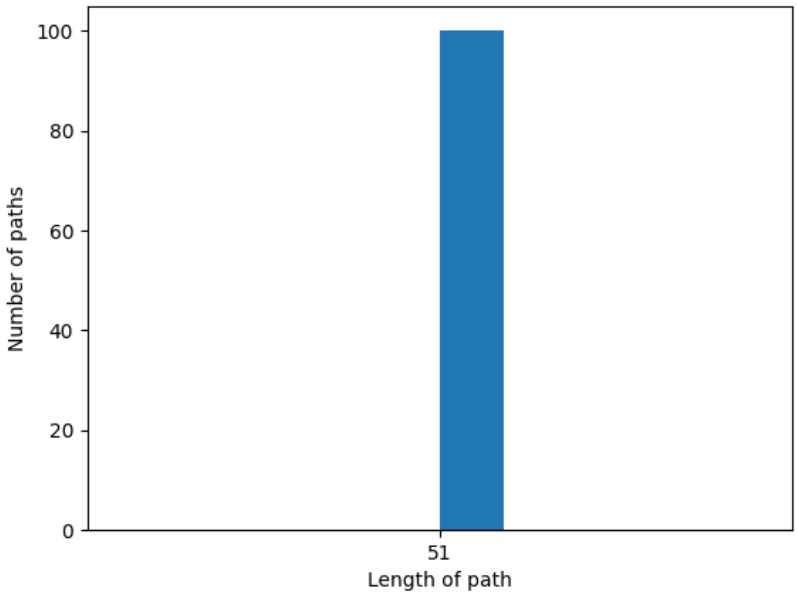

Figure 17 - Nombre de chemins en fonction de leur longueur pour une grille 20x20 réaliste

La même constatation qu'avec un labyrinthe de dimensions 10x10 peut être faite.

3.2.3 -Grille 30x30

Voici le labyrinthe réaliste, de dimensions 30x30, qui a été utilisé pour réaliser ces tests.

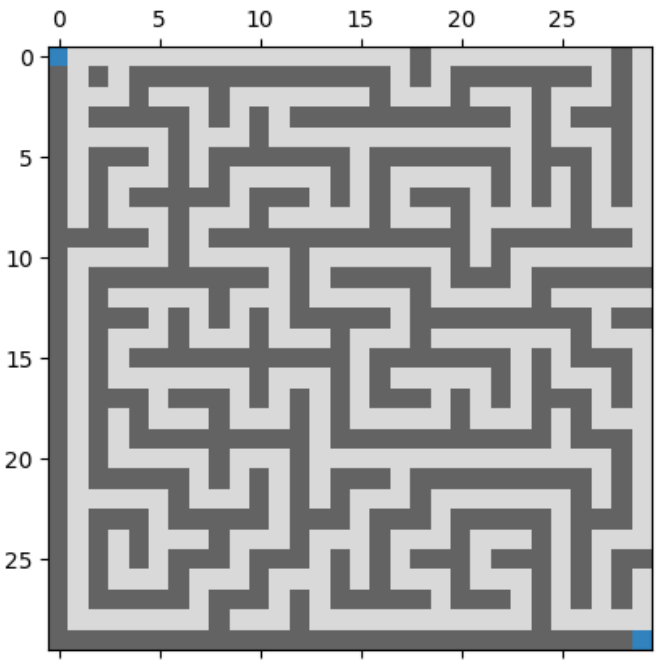

Figure 18 - Grille 30x30 réaliste utilisée pour les tests

Voici la solution optimale pour ce labyrinthe :

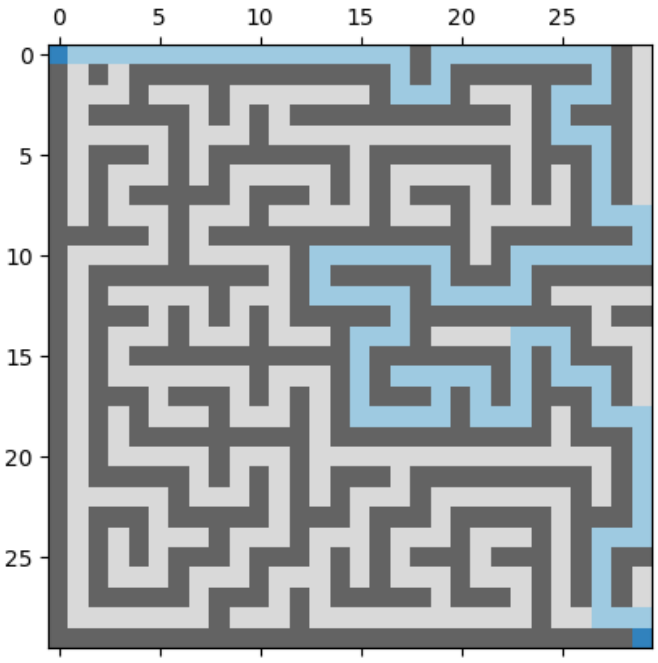

Figure 19 - Exemple de solution pour une grille 30x30 réaliste

Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

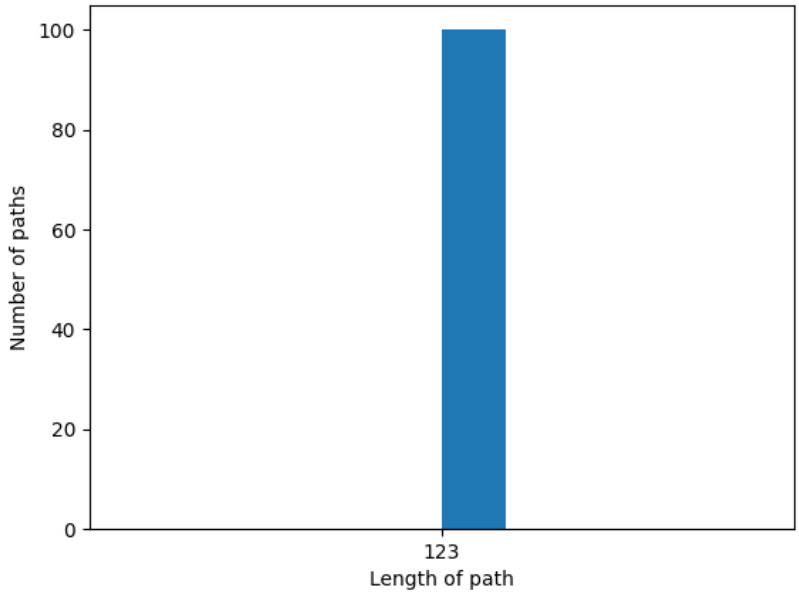

Figure 20 - Nombre de chemins en fonction de leur longueur pour une grille 30x30 réaliste

Même avec des dimensions plus élevées (30x30), nous pouvons constater que toutes simulations convergent vers la même solution, qui est probablement la meilleure.

Nous pouvons donc en conclure que l'algorithme fonctionne bien mieux avec des labyrinthes aléatoires réalistes (fermés). En effet, même avec des durées bien plus faibles, les mêmes résultats sont atteints.

4 - Améliorations / optimisations potentielles

4.1 - Critère(s) d'arrêt

Pour l'instant, le seul critère d'arrêt de cet algorithme est le temps écoulé. Si celui-ci dépasse le temps maximal précisé dans les paramètres de la fonction ***`solve_labyrinthe()`***, le chemin du meilleur individu trouvé jusqu'à présent est retourné.

Un second critère pourrait être mis en place : **la convergence**. Si, à partir d'un certain nombre de générations, les individus n'évoluent plus, cela signifie que la diversité dans la population s'est beaucoup affaiblie. Le seul moyen d'obtenir de nouveaux résultats dépend uniquement de la mutation des individus.

4.2 - Labyrinthe impossible

Cette implémentation ne permet pas de détecter si le labyrinthe est solvable ; si au moins un chemin permet d'atteindre la cellule de destination. Comme aucun critère de convergence pour l'arrêt n'a été mis en place, l'algorithme ne s'arrêtera pas avant que le temps écoulé ne dépasse la durée maximale fournie. La solution retournée existera, mais ne permettra pas d'atteindre la cellule finale, et aucune indication sur la faisabilité du labyrinthe ne sera fournie à l'utilisateur.

Avant le commencement des générations, un algorithme, notamment *Astar*, pourrait contrôler si au moins une solution existe. Si ce n'est pas le cas, *None* pourrait être retournée à l'utilisateur, ou alors une exception pourrait être levée.

5 - Annexes

5.1 - Table des illustrations

5.2 - Bibliographies et références

Pasquier, T. & Erdogan, J. (2011). *Genetic Algorithm Optimization in Maze Solving Problem*. science.donntu.edu.ua. https://science.donntu.edu.ua/ipz/sobol/links/maze_solving.pdf

Jonasson, A., Westerlind, S. & Herman. (2016). *Genetic algorithms in mazes : A comparative study of the performance for solving mazes between genetic algorithms, BFS and DFS.* Digitala Vetenskapliga Arkivet. https://www.diva-portal.org/smash/get/diva2:927325/FULLTEXT01.pdf