TP4 – Rapport technique – ISC3il-b

|  |
| --- |
| **Algorithmes génétique - Labyrinthe** |

|  |
| --- |
| 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** |

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

Table des matières

[1 - Glossaire 2](#_Toc120277165)

[2 - Introduction 3](#_Toc120277166)

[2.1 - Contexte 3](#_Toc120277167)

[2.2 - Description du problème 3](#_Toc120277168)

[2.3 - Choix des technologies 3](#_Toc120277169)

[2.3.1 - Python 3](#_Toc120277170)

[2.3.2 - Framework DEAP 3](#_Toc120277171)

[2.3.3 - Algorithmes génétiques 3](#_Toc120277172)

[3 - Réalisation 4](#_Toc120277173)

[3.1 - Définition d’un gène 4](#_Toc120277174)

[3.2 - Définition d’un chromosome 4](#_Toc120277175)

[3.2.1 - Taille d’un chromosome 4](#_Toc120277176)

[3.3 - Récupération du chemin à partir d’un chromosome (compute\_complete\_valid\_path) 5](#_Toc120277177)

[3.3.1 - Prévenir les culs-de-sac 5](#_Toc120277178)

[3.4 - Récupération du chemin le plus court 5](#_Toc120277179)

[3.5 - Fonction de sélection 5](#_Toc120277180)

[3.6 - Fonction de crossover 5](#_Toc120277181)

[3.7 - Fonction de mutation 6](#_Toc120277182)

[3.8 - Fonction de fitness 6](#_Toc120277183)

[4 - Résultats 7](#_Toc120277184)

[4.1 - Labyrinthes aléatoires 7](#_Toc120277185)

[4.1.1 - Grille 10x10 7](#_Toc120277186)

[4.1.2 - Grille 20x20 7](#_Toc120277187)

[4.1.3 - Grille 30x30 7](#_Toc120277188)

[4.2 - Labyrinthes réalistes 7](#_Toc120277189)

[4.2.1 - Grille 10x10 7](#_Toc120277190)

[4.2.2 - Grille 20x20 7](#_Toc120277191)

[4.2.3 - Grille 30x30 7](#_Toc120277192)

[5 - Améliorations / optimisations potentielles 8](#_Toc120277193)

[5.1 - Critère(s) d’arrêt 8](#_Toc120277194)

[6 - Annexes I](#_Toc120277195)

[6.1 - Table des illustrations I](#_Toc120277196)

[6.2 - Bibliographies et références II](#_Toc120277197)

[6.2.1 - Sites Web II](#_Toc120277198)

[6.2.2 - Livres II](#_Toc120277199)

[6.2.3 - Autres II](#_Toc120277200)

1. Introduction
   1. Contexte
   2. Description du problème
   3. Choix des technologies
      1. Python
      2. Framework DEAP
      3. Algorithmes génétiques
2. Réalisation
   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.
  1. 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. Un chromosome, à partir d'une position initiale, peut être appliqué afin d'obtenir une position finale. Cela est fait dans la fonction de compute\_complete\_valid\_path décrite ci-dessous.

* + 1. Taille d’un chromosome

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

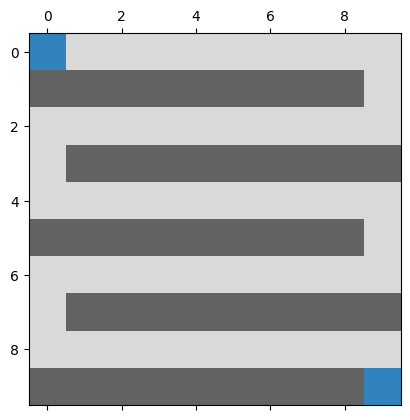


Figure 1 - Labyrinthe menant au chemin le plus long

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.

* 1. 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.

* + 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.

Si aucun des quatre types de gène 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.

* 1. 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 à partir de la dernière position (0, 0) se trouvant avant la target, jusqu'à la première position correspondant à la target.

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

* 1. 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 un taille de tournoi (tournsize) valant 3.

* 1. 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 : Le chromosome du premier parent est [1, 1, 1, 1, 1, 1, 1, 1, 1, 1], celui du deuxième vaut [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].

* 1. 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 de 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.

* 1. Fonction de fitness

La fonction de fitness compute\_fitness() permet de calculer la fitness d'un individu, qui peut être comparée à un score, permettant de classer les individus. Dans cette simulation, plus la 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, sa 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). Sa fitness est alors la longueur du chemin, à laquelle est additionnée la distance euclidienne séparant cette cellule de la cible.

1. 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 :

Une image contenant table

Description générée automatiquement

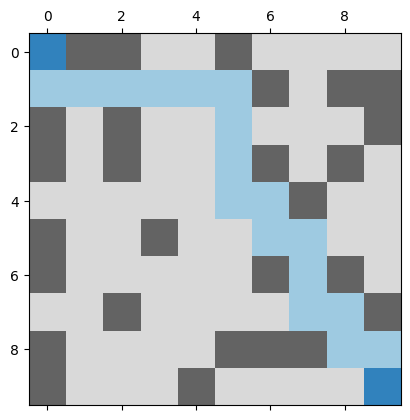
* 1. Labyrinthes aléatoires (ouverts)
     1. Grille 10x10

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

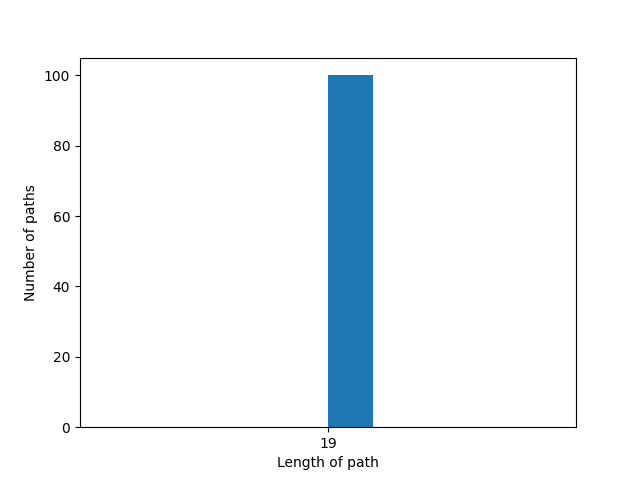
Une image contenant texte, mots croisés

Description générée automatiquement

Voici un exemple de solution, qui n’est pas forcément optimale :

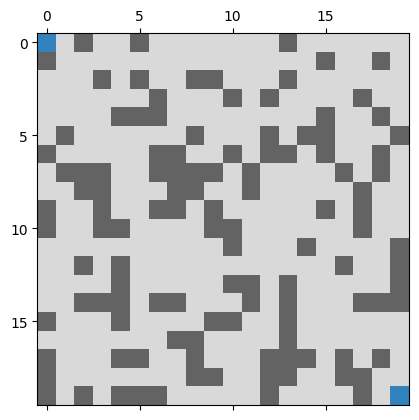


Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

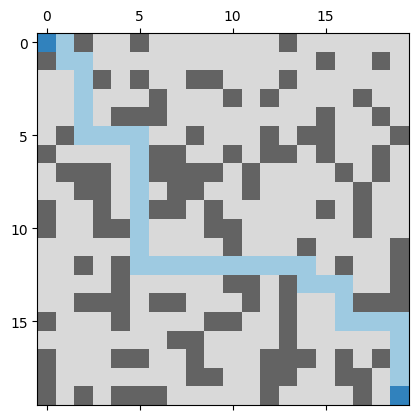


Nous pouvons constater que, pour un labyrinthe de faible taille, l’algorithme est consistant, et toutes les simulations parviennent à la même solution.

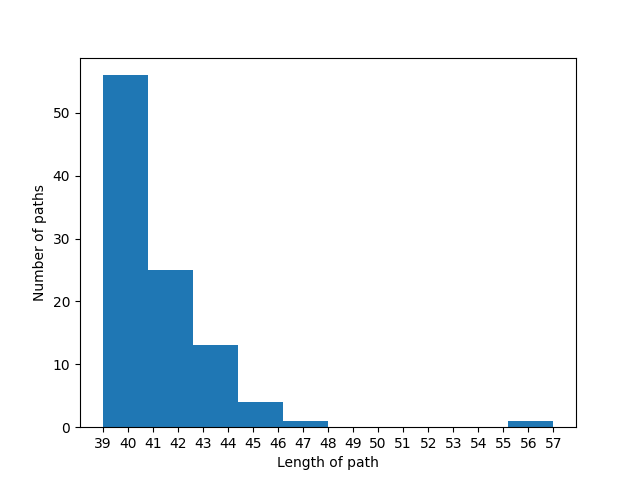
* + 1. Grille 20x20



Voici un exemple de solution, qui n’est pas forcément optimale :

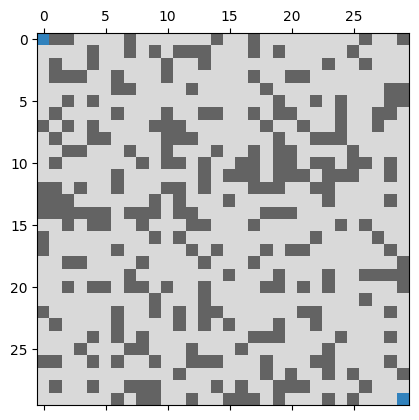


Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

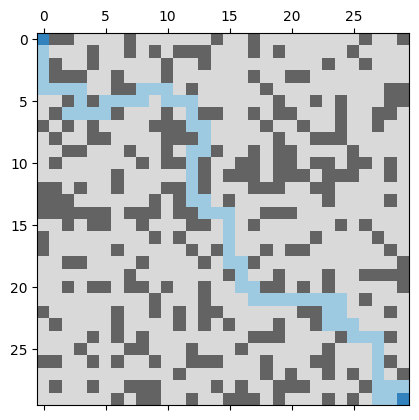


Avec un labyrinthe de dimensions 20x20, les simulations ne convergent pas toutes vers la même solution. Cependant, une grande majorité s’entend à dire que la solution optimale est un chemin de taille 39.

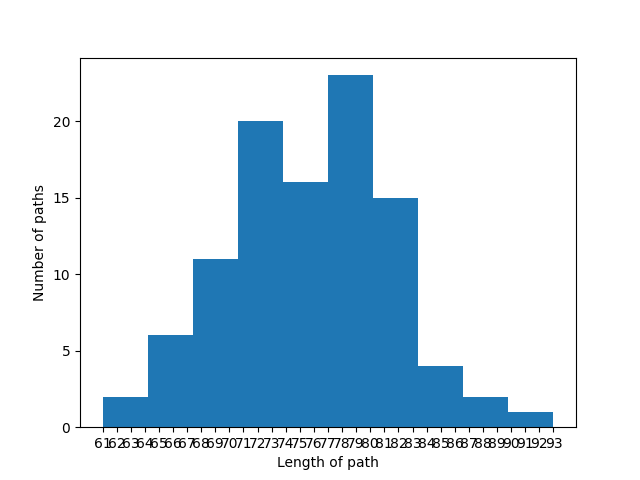
* + 1. Grille 30x30



Voici un exemple de solution, qui n’est pas forcément optimale :

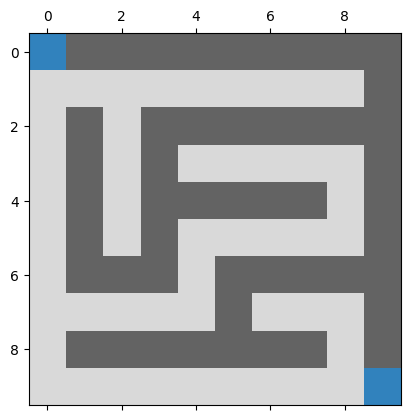


Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

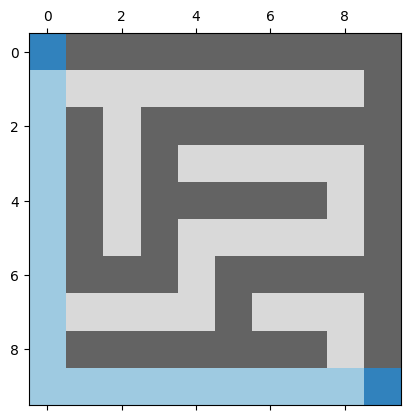


Contrairement aux deux premiers graphiques, un labyrinthe de dimensions 30x30 ne permet d’approcher une solution optimale à tous les coups. Avec plus d’individus (de simulations), ce graphique ressemblerait certainement à une gaussienne.

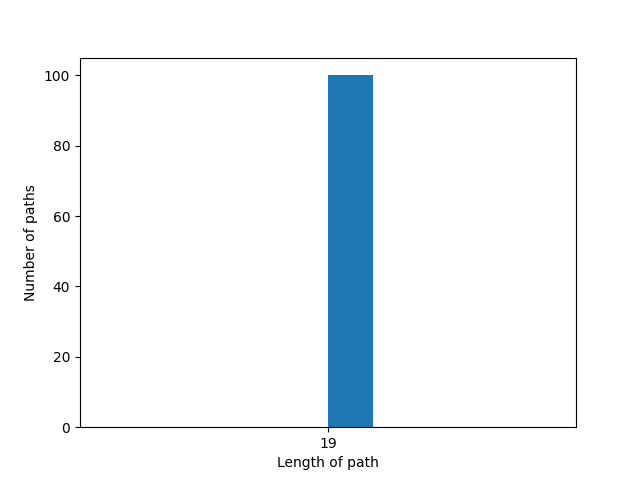
* 1. Labyrinthes aléatoires réalistes (fermés)
     1. Grille 10x10



Voici un exemple de solution, qui n’est pas forcément optimale :



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



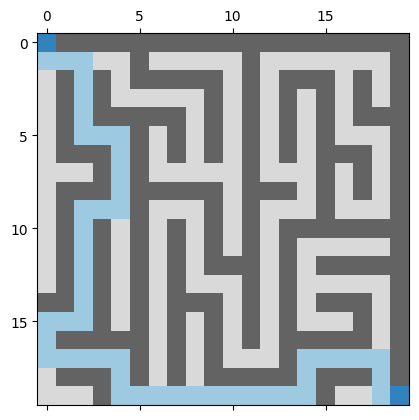
Grâce à la prévention des culs-de-sac, l’algorithme donne des résultats constants.

* + 1. Grille 20x20

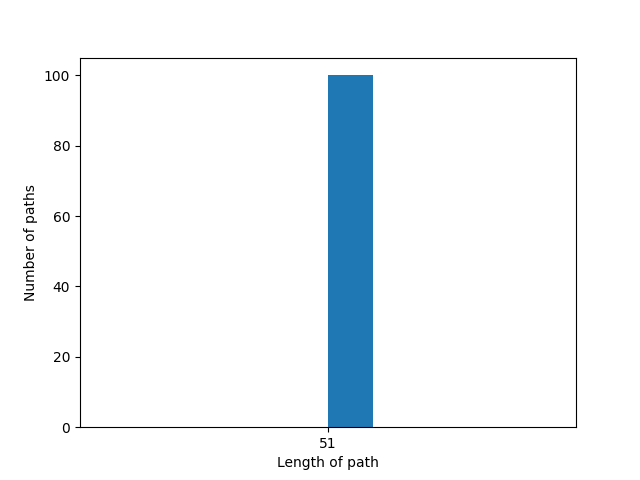
Une image contenant texte, blanc

Description générée automatiquement

Voici un exemple de solution, qui n’est pas forcément optimale :

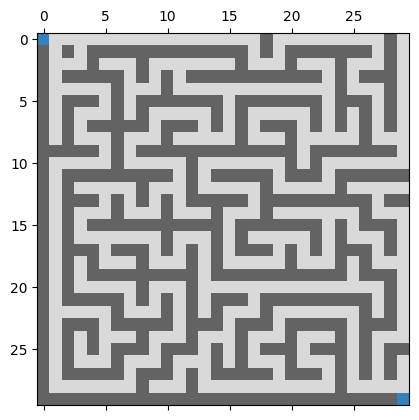


Le graphique ci-dessous montre le nombre de chemins en fonction de leur longueur.

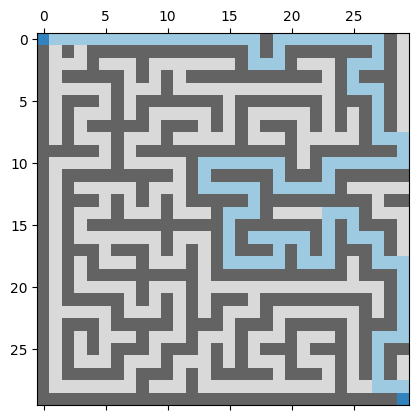


La même constatation que une labyrinthe de dimensions 10x10 peut être faire.

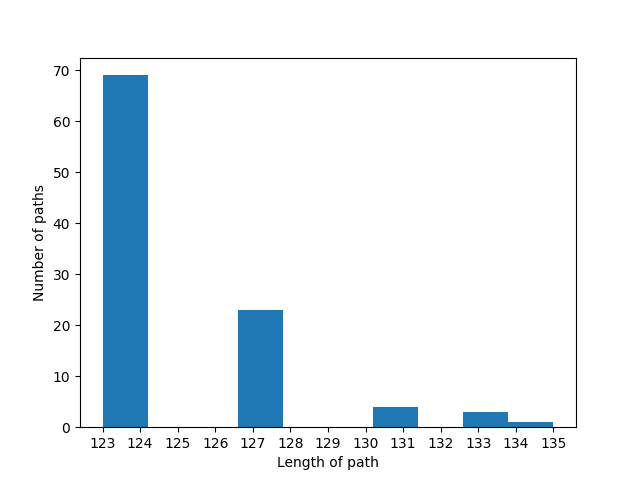
* + 1. Grille 30x30



Voici un exemple de solution, qui n’est pas forcément optimale :



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



Avec une dimensions plus élevés

1. Améliorations / optimisations potentielles
   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 maximale 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 très affaiblie. Le seul moyen d'obtenir de nouveaux résultats dépend uniquement de la mutation des individus.

1. Annexes
   1. Table des illustrations

[Figure 1 - Labyrinthe menant au chemin le plus long 5](#_Toc120277016)

* 1. Bibliographies et références
     1. Sites Web
     2. Livres
     3. Autres