Etape 1 : Construction aléatoire d'un labyrinthe

Dans cette première partie j'ai implémenté l'algorithme décrit dans la feuille du projet. J'ai décidé de faire un court compte rendu afin d'expliquer rapidement et plus clairement mon code, mais vous pouvez toujours retrouver des commentaires dans le code (fourni avec ce compte rendu) pour plus de détails.

J'ai donc divisé l'algorithme en plusieurs sous fonctions :

- *initialize_maze*(taille) qui prend en entrée une taille et permet d'initialiser et renvoyer un tableau de taille « taille » fois « taille » avec des zéros à l'intérieure.
- *voisins_valides*(maze, i, j) qui prend en entrée un labyrinthe et les coordonnées i et j d'une case du labyrinthe. Cette sous fonction retourne la liste des voisins valides autour de la case de coordonnées (i, j), c'est à dire les case voisines qui sont à 0.
- *voisin_eligible_bool*(maze, i, j) prend en entrée un labyrinthe et les coordonnées i et j d'une case du labyrinthe. Cette sous fonction renvoie une valeur booléenne (True ou False), elle renvoie True si parmi les voisins de la case de coordonnées (i, j) il n'y a que 0 ou 1 voisins qui ont déjà été parcouru, c'est à dire qui sont égales à 1. Sinon elle renvoie False.
- *voisins_eligibles*(maze, liste_voisin_valide) prend en entrée un labyrinthe et une liste de voisins valides. Elle retourne la liste des voisins éligibles parmi les voisins valides. Pour cela cette sous fonction s'aide de la sous fonction nommée *voisin_eligible_bool*.

Enfin on retrouve la fonction principale, *generate_maze*(taille) qui prend entrée la taille voulu d'un labyrinthe. Cette fonction utilise les sous fonctions *initialize_maze*, *voisins_valides*, *voisins_eligibles,* elle permet de générer un labyrinthe comme décrit par l'algorithme fourni, elle utilise entre autres une pile. Cette fonction retourne le labyrinthe généré sous forme d'un tableau.

J'ai également créé d'autre fonctions :

- *display_maze*(maze, title) qui prend en entrée un labyrinthe et un titre et il sert à afficher le labyrinthe sous forme d'une image avec le titre rentré. Le labyrinthe est affiché avec des murs noires, le chemin est blanc et si on en a mis, le point à atteindre (but) sera affiché en rouge et le point de départ sera affiché en bleu. Enfin si on a calculé le chemin minimum, on affichera ce chemin en vert.

- *add_goal_maze*(maze) qui prend en entrée un labyrinthe, il choisit au hasard une case qui sera le but à atteindre parmi les cases du chemin du labyrinthe. Il modifie cette case dans le labyrinthe et renvoie le labyrinthe.

- add_start_point_maze(maze) qui prend en entrée un labyrinthe, il choisit au hasard une case qui sera le point de départ parmi les cases du chemin du labyrinthe. Il modifie cette case dans le labyrinthe et renvoie le labyrinthe. Cette fonction est utilisée seulement à partir de l'étape 2 du projet.

Ainsi pour créer un labyrinthe de taille 50 il suffit de taper :

```
labyrinthe = generate_maze(50)
```

Pour rajouter un but :

```
labyrinthe = add_goal_maze(labyrinthe)
```

Pour l'afficher :

```
display_maze(labyrinthe)
```

Suite à des tests on obtient ce tableau :

Taille (côté du labyrinthe)	5	50	500
Temps (seconde)	0,00429665	0,252506	45,1077

Pour estimer la complexité de notre algorithme, j'ai créé un script qui génère un labyrinthe de taille 10 jusqu'à une taille de 1000 en incrémentant de 10 en 10. A chaque génération de labyrinthe je compte en même temps le temps pris. Ainsi à la fin je peux afficher le temps pris en fonction de la taille. Vous pouvez retrouver ce script à la fin du fichier python en commentaire. Le résultat est le suivant.

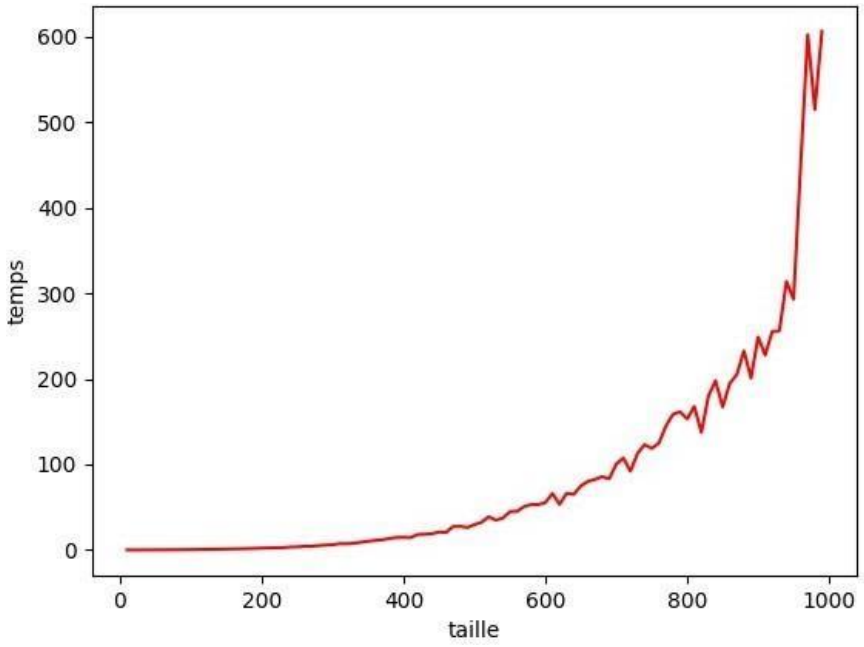


Figure 1 : temps en seconde pris pour générer un labyrinthe en fonction de sa taille

On remarque donc que la complexité semble exponentielle, ce qui est à peu près logique au vu du nombre de boucle for imbriquée qu'on utilise.

Etape 2 : Algorithme de Dijkstra

Dans cette partie j'ai implémenté l'algorithme de dijkstra décrit dans la feuille du projet. J'ai décidé de faire un court compte rendu afin d'expliquer rapidement et plus clairement mon code, mais vous pouvez toujours retrouver des commentaires dans le code (fourni avec ce compte rendu) pour plus de détails. J'ai laissé les fonctions définies lors de l'étape 1 du projet afin de pouvoir effectuer des tests à la fin, ainsi le nouveau code commence seulement à partir de la ligne 184.

J'ai donc en premier lieu implémenté l'algorithme de dijkstra qui prend en entrée un labyrinthe avec un but et une case départ et qui ressort un tableau représentant le labyrinthe affichant dans les cases la distance au but.

Ensuite j'ai créé une fonction qui à partir de la sortie de la fonction djikstra calcul la map directionnelle du labyrinthe, c'est à dire que chaque case pointe sur la prochaine case pour aller dans la direction du but.

A partir de la map directionnelle il est facile de déduire le chemin reliant deux points, c'est ce que j'ai fait avec une nouvelle fonction, elle ressort le labyrinthe modifié avec le chemin.

J'ai également fait une fonction qui trouve le chemin entre deux points sans utiliser de map directionnelle (seulement le tableau fournit, pas dijkstra).

Il était aussi demandé de montrer le labyrinthe coloré en fonction de la distance de la case par rapport au but, c'est ce que j'ai fait avec le code fourni.

Vous pouvez vérifier cela en lançant le fichier python fourni, normalement des tests dans le main sont écrits. La case de départ est affichée en bleu, la case d'arrivée en rouge et le chemin les reliant en vert. Pour rappel, ces deux cases sont tirées aléatoirement.

Enfin il était demandé de calculer la longueur moyenne de la solution en fonction de la taille du labyrinthe. Voici les résultats que j'ai obtenu, vous pouvez retrouver les instructions à la fin du fichier en commentaire.

Taille (côté)	8	16	32	64	128	256	512
Longueur moyenne (N)	13,4	16,4	135,4	290,1	1309,9	3604,6	13093,0

Même s'il aurait fallu faire plus d'itérations pour trouver de meilleures valeurs, on remarque que la longueur du chemin augmente plus que linéairement. Cependant si on rapporte la longueur au nombre total de cases (taille*taille), alors on trouve que le rapport est entre 5 % et 8 % (à partir de taille égale 16).

Etape 3 : Algorithme génétique

Dans cette partie j'ai implémenté un algorithme génétique pour trouver une solution dans le labyrinthe. Premièrement j'ai mis en place différentes fonctions pour mettre en place cette solution. En effet elle est composée de plusieurs phases, la genèse pour générer des chemins et l'évolution pour faire évoluer les chemins. L'évolution est composée de différentes sous phases, l'évaluation, la sélection, la reproduction et la mutation.

Premièrement j'ai créé une fonction permettant de définir un point de départ et d'arrivée assez éloigné afin de ne pas avoir de chemins trop faciles à trouver, pour cela je m'aide de la map générée par *dijkstra* lors du projet précédent.

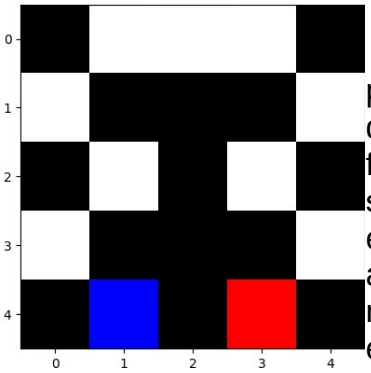
Ensuite la fonction *generation* génère une centaine de chemin, ayant une taille comprise entre la distance minimale pour résoudre le labyrinthe et deux fois la taille du labyrinthe.

La fonction *selection* évalue et sélectionne les chemins. Elle s'appuie sur la fonction de *fitness* et sélectionne un taux ts de chemin.

La fonction *reproduction*, reproduit les chemins sélectionnés (parents) pour créer des chemins enfants qui auront des gènes des 2 parents.

La fonction *mutation* fait apparaître quelques mutations parmi la population de chemins.

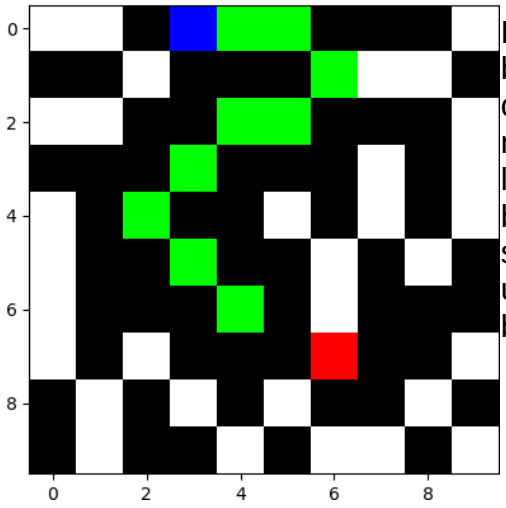
Revenons à la fonction *fitness* qui est la pierre angulaire de l'algorithme génétique, en effet elle permet d'évaluer la performance d'un chemin. Pour calculer un valeur cette fonction prend en compte la distance au but de la dernière case valide du chemin (calculé par la fonction *end_cell*) par rapport à la case d'arrivée, et des pénalités. Pour que les chemins ne soient pas coincés dans des culs-de-sac, j'ai ajouté une fonction qui pose des phéromones, qui s'active à l'intérieur de la fonction *end_cell*. Les pénalités sont définies en fonction du nombre de cases redondantes parcourues, et de la longueur du chemin, en effet dans certains cas le chemin doit s'éloigner de la case d'arrivée pour pouvoir atteindre la case d'arrivée, ainsi il faut compenser le fait que dans fitness on regarde la distance (à vol d'oiseau) entre la case où on se trouve et la case d'arrivée.



En effet dans le cas du labyrinthe ci-contre, si on veut relier le point bleu au point rouge il va y avoir un problème. Car la distance entre les points est faible alors que pour les relier il faut faire tout le tour, ainsi dans fitness la fonction *dist* va nous faire stagner sur cette case, car on si on s'éloigne la distance entre rouge et bleu augmente. On se retrouve donc coincé. Une solution est alors de favoriser dans les pénalités un chemin plus long et lui mettre un poids plus significatif que la distance entre les cases bleu et rouge.

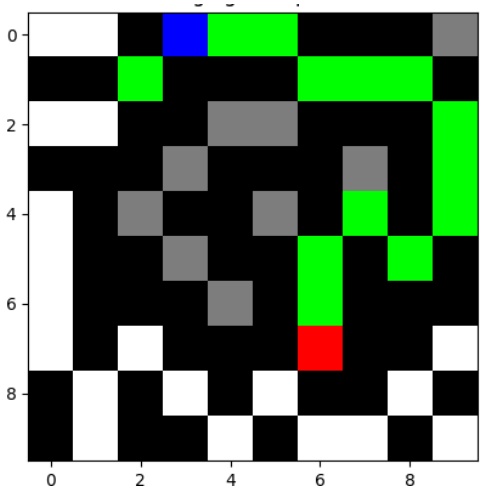
Cependant cela peut favoriser des chemins plus longs par rapport au critère de la distance dans d'autres labyrinthes.

Intéressons-nous maintenant à la question 1 (sans phéromone) et ses limites. Dans cette partie nous cherchons un chemin à partir de la distance à la case d'arrivée et aux pénalités. Dans un premier temps sur certains labyrinthes cela fonctionne correctement. Mais on remarque que dès que des chemins arrivent dans des impasses près de la case d'arrivée, ils restent coincés car ils ne veulent pas faire demi-tour. Pour qu'ils puissent se décoincer il faut alors attendre une série de mutations heureuse permettant de trouver un meilleur chemin.



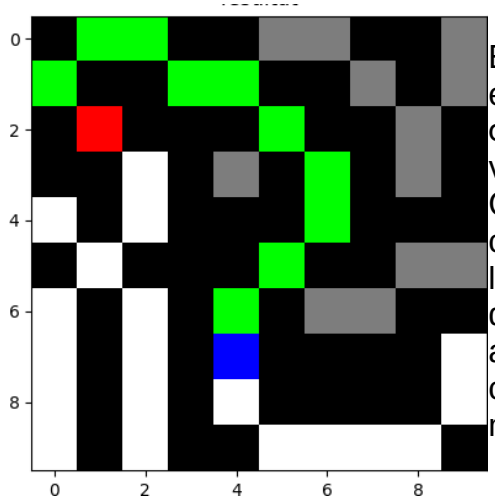
Dans ce cas concret, le chemin (en vert) va rester bloqué dans cette impasse car il se trouve près de la case d'arrivée (rouge), il ne va pas pouvoir partir. A moins d'une série de bonnes mutations. C'est une des limites de cette première partie, il faudrait attendre beaucoup de générations pour pouvoir trouver une solution. C'est pourquoi dans la deuxième partie on utilise un mécanisme de phéromones pour éviter d'être bloqué à un minimum local.

Analyse de la deuxième partie. Dans la deuxième partie une pose de phéromone a été ajoutée. C'est à dire que dès qu'on est dans une impasse, on remonte l'impasse jusqu'à un croisement et on condamne l'impasse avec des phéromones afin de signifier aux autres chemins de ne plus passer par là. Cela nous sert à éviter d'être bloqué avec des minimums locaux. Ainsi on peut résoudre le problème précédent.



Ici après être allé dans l'impasse le chemin est remonté et a condamné l'impasse en posant des phéromones (cases grises), pour permettre aux autres chemins d'emprunter un nouvel itinéraire.

Cependant même si cela fonctionne, à cause du fait de favoriser une longueur de chemin assez grande pour éviter d'être bloqué au début, j'obtiens des incohérences parfois.



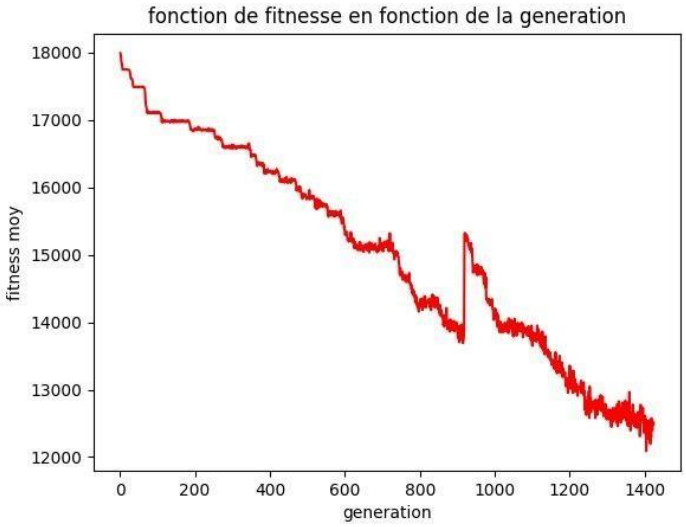
En effet dans ce cas, on remarque que à l'intersection, en haut au milieu (4,1), le chemin a essayé d'aller à droite (cases grises), alors que cela n'a aucun sens. S'il va à droite il s'éloigne de la case d'arrivée (case rouge). C'est à cause du fait que je favorise la longueur du chemin dans mes pénalités. Cela peut donc augmenter les temps de calcule. Mon problème principal est de devoir à la fois donner un poids important à la distance au but mais aussi pas trop important, pour que le chemin ne reste pas bloqué à un endroit proche du but mais inatteignable.

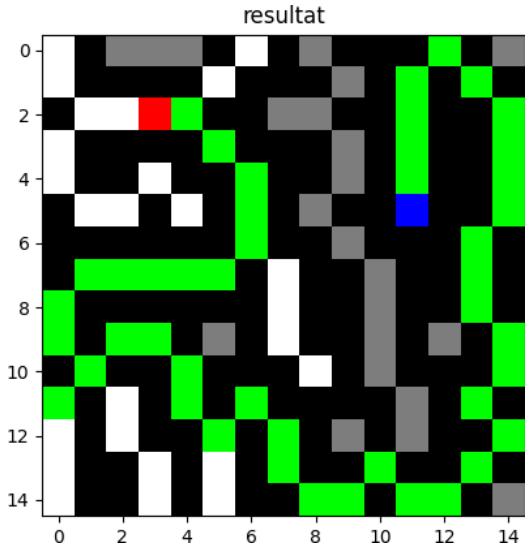
Axes d'améliorations. Pour éviter le problème mentionné on pourrait regarder à chaque intersection que l'on croise quelle direction est la plus proche de l'arrivée et choisir en conséquence le chemin à prendre. En effet si on arrive dans une impasse suit au chemin pris, la pose de phéromone nous fera revenir sur nos pas, et on prendra la direction restante.
Une autre chose pour accélérer la recherche du chemin serait de pouvoir opérer des mutations à des endroits ciblés des chemins, plutôt qu'au hasard dans tout le chemin. En effet faire des mutations juste après tous les gênes communs du début des chemins (à cause de la reproduction) permettrait de trouver de nouveaux chemins plus rapidement. Enfin au lieu de choisir des positions au hasard, choisir seulement des positions valides peut permettre une meilleure rapidité.

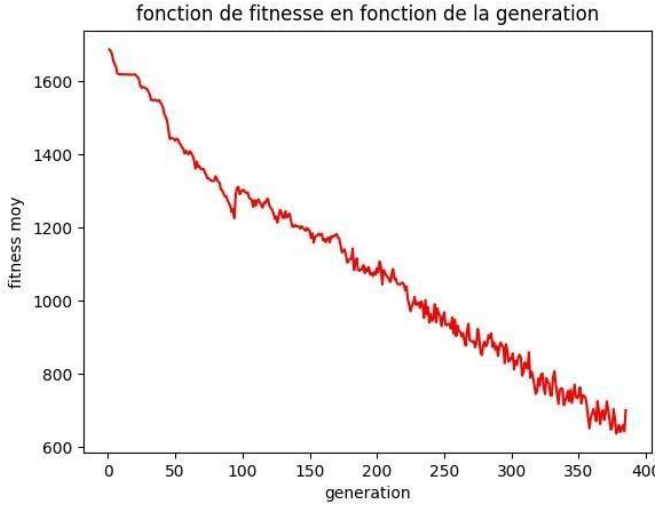
Vous pouvez retrouver toutes ces fonctions, d'autres sous-fonctions associées et toutes les fonctions des projets précédents dans le fichier python, avec des commentaires. Des tests sont fournis, deux d'entre eux utilisent des fichiers textes (cas particuliers), si vous ne les voulez pas, mettez en commentaire la première partie du main. Ensuite 10 labyrinthes avec leurs fonctions de fitness sont générés aléatoirement et affichés un par un (nb : il faut fermer le graphique à chaque fois pour que le programme continue).

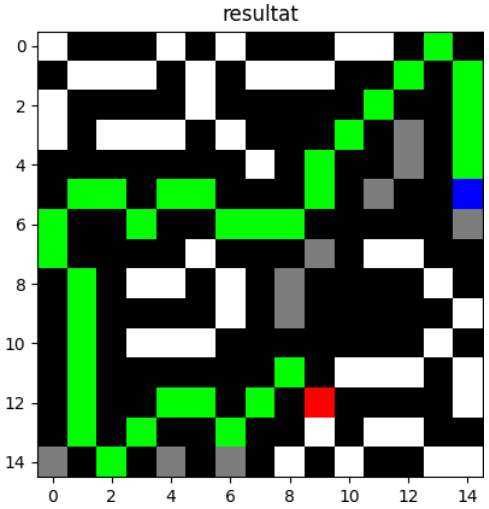
Peu content des performances de mon algorithme pour de grands labyrinthes j'ai fait un autre algorithme (*resol_other_way*) n'utilisant pas la résolution génétique mais utilisant les mêmes hypothèses de départ.

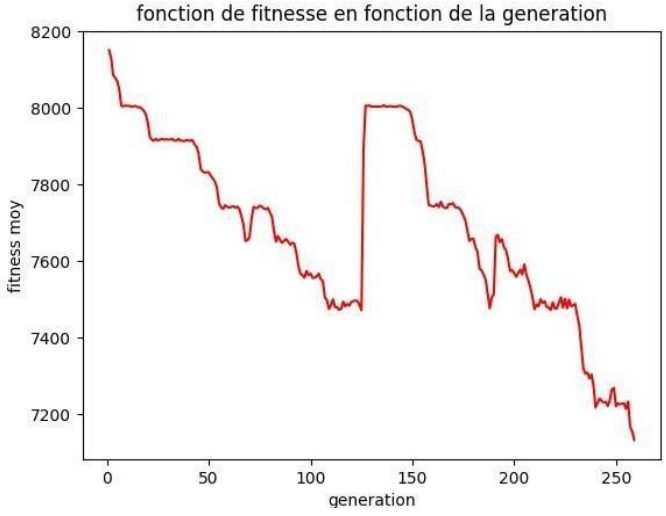
Voici des exemples avec la résolution génétique, on remarque des piques lorsqu'on bouche une impasse et qu'on repart par un autre chemin :

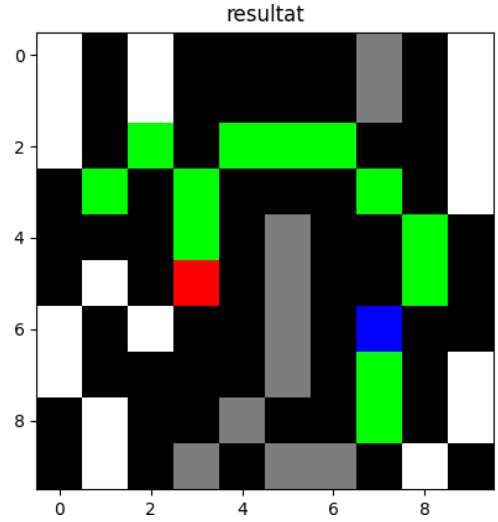

fonction de fitnesse en fonction de la generation
fitness moy
generation
18000
17000
16000
15000
14000
13000
12000
0
200
400
600
800
1000
1200
1400


resultat
0
2
4
6
8
10
12
14


fonction de fitnesse en fonction de la generation
fitness moy
generation
1600
1400
1200
1000
800
600
0
50
100
150
200
250
300
350


resultat
0
2
4
6
8
10
12
14


fonction de fitnesse en fonction de la generation
fitness moy
generation
8200
8000
7800
7600
7400
7200
0
50
100
150
200
250


resultat
0
2
4
6
8