Algorithme Génétique

Les algorithmes génétiques sont des algorithmes de recherche qui utilise des principes de la biologie pour trouver des solutions à des problèmes. Ils sont basés sur la sélection naturelle et la reproduction pour générer des solutions optimales. Les algorithmes génétiques sont souvent utilisés pour la résolution de problèmes difficiles d'optimisation.

Cette archive contient :

- 2 scripts python
- Un dossier Graph_saved
- Un dossier CSV_saved
- Un dossier graphics
- Un dossier game

Objectif :

L'objectif est de concevoir des cartes attrayantes pour les joueurs dans un jeu de type Pokémon, où le but est d'explorer une grotte. La carte doit être suffisamment détaillée et intéressante pour susciter l'exploration, avec des zones facultatives offrant des récompenses pour encourager les joueurs à explorer. Il n'y a pas de risques réels pour le joueur dans ce genre de jeu, mais plutôt une expérience d'exploration à découvrir.

Contenu :

Le script GeneticAlgo.py contient un programme qui applique un algorithme génétique sur une matrice. Cette matrice représente une carte de jeu vidéo avec une entrée, une sortie, des cases ou l'on peut marcher, elles sont dites « walkable » et des cases « wall » qui bloque le passage.

Une fois lancé, le script créé les dossiers CSV_created et Graph_created.
Toutes les 100 générations, des graphiques affichant différents résultats sont sauvegardés dans Graph_created, et 5 individus de la génération sont sauvegardés sous forme de CSV dans CSV_created à titre d'exemple.

Le script show_map.py sert à créer une prévisualisation de la carte générée. Il faut pour cela indiquer le chemin du CSV que l'on souhaite représenter. L'image générée sera sauvegardée dans le dossier graphics.

Le dossier graphics contient les sprites pour la visualisation ainsi qu'un exemple d'image généré avec le script show_map.py, appelé « visualize_map_saved.png ».

Le dossier game contient un jeu développé avec pygame.
Le code source a été trouvé sur internet, je n'y ai pas participé. J'ai juste adapté les sprites et la lecture du CSV afin de créer une carte. Le but est de visualiser la carte créée d'un autre point de vue.

Exécution :

Algorithme génétique:
python3 GeneticAlgo.py

Prévisualisation de carte:
Adapter le chemin situé à la ligne 20 pour utiliser le CSV voulu puis
python3 show_map.py

Lancer le jeu :
Copier/coller son fichier CSV dans le dossier game/source_game à la place de « map.csv ».
Créer le terrain de la carte avec
python3 create_floor.py
Lancer le jeu, dans le dossier game/source_game/ avec
python3 main.py

Code :

Le code est composé de 2 classes, la classe _map et la classe genetic_agent.

- La classe _map contient principalement :
    - La méthode « get_random_env » qui génère une carte aléatoire,
    - La méthode « set_tile_mask » génère une matrice de même taille que le labyrinthe, cette matrice contient des 1 aux positions où le joueur peut accéder et des 0 sinon (les cases « walkable » mais qui sont entouré de murs et donc sont inaccessibles sont à 0).
    - La méthode set_end_reachable(), renvoi vrai si la fin est atteignable depuis le début grâce au masque généré précedemment.
    - La méthode get_shortest_way() est une implémentation d'une variante de l'algorithme A*. La méthode renvoi la distance entre l'entrée et la fin de la carte.
- La classe genetic_agent contient principalement :
    - La méthode « get_random_population() » qui crée une liste de _map
    - La méthode « eval_fitness() qui crée une liste qui répertorie la « fitness » c'est-à-dire la qualité de chaque _map. 4 paramètres ont été pris en considération pour noter les cartes. Tout d'abord la possibilité d'atteindre la fin depuis l'arrivée. Deuxièmement, la distance entre l'entrée et l'arrivée, Troisièmement, la proportion de cases « wall » par rapport aux cases « walkable ». Dernièrement, les cases « walkable » qui ne sont pas accessibles ne sont pas désirable.
    - La méthode « crossover » sélectionne des cartes parentes relativement à leurs qualités. (Les qualités sont divisées par la qualité minimum, ainsi la pire carte aura 1, l'idéal théorique étant 0. On soustrait ce résultat à 1, puis on peut sélectionner les parents selon ce score grâce à la fonction « random.choices » qui prend en considération les poids calculés précédemment. La carte fils est créée à partir de 2 parents. L'algorithme génère un nombre N entre 0 et le nombre de colonne (ou de ligne) de la

carte, il attribue ensuite les colonnes (ou les lignes) 0 à N du parent 1 à l'enfant, puis les colonnes (ou les lignes) N+1 jusqu'à la dernière du parent 2. Lors de ces attributions, on ne prend pas en compte l'entrée et la sortie afin d'éviter d'en transmettre plusieurs à l'enfant. On les choisit a posteriori aléatoirement entre les parents.

- La méthode « mutation » sélectionne un nombre pour la taille et confectionne un « patch ». Un nombre entre 0 et 1 est également généré, ce dernier déterminera si le « patch » changera les cases pour des cases « wall » ou des cases « walkable ». Ce patch est étalé aléatoirement entre la longueur et la largeur sur la distance choisie auparavant. Il parcourt toute la carte et a une certaine probabilité de s'appliquer. Dans ce cas, toutes les cases sous le patch sont modifiées.
  Une mutation spécifique supplémentaire s'opère sur l'entrée et la sortie, afin d'augmenter leurs taux de mutations. Leurs positions sont alors déplacées aléatoirement sur la carte.

Les algorithmes de notation de cartes peuvent se baser sur une multitude de critères. Parmi les plus importants figurent l'accessibilité de la fin, le ratio entre les cases, la distance entre le début et la fin et l'esthétique. Pour éviter une absence de variété, il est crucial d'imposer un ratio entre les cases « walkable » et « wall » et de considérer la distance entre l'entrée et la sortie. De plus, pénaliser les cartes qui comportent des cases "walkable" mais inaccessibles renforce l'aspect esthétique. Les méthodes de mutation et de cross-over ont été améliorées : le cross over ne fonctionne pas case par case, et les mutations utilisent le système de patch, ce qui permet de prendre en compte l'environnement de la case. Cela a plus de sens que de faire des changements locaux car les cases sont fortement dépendantes de leurs influences mutuelles.

Résultats :

Comme précisé plus tôt, nous avons enregistré différents résultats dans Graph_saved pour chaque valeur de mutation spécifiée dans le code en créant un dossier pour chacune. Trois courbes ont été enregistrées à chaque itération (sur la liste de mutation) : la première représente l'évolution de la "fitness" globale, la deuxième l'évolution de la distance entre le début et la fin, et la troisième le pourcentage de cases "wall". Dans la suite de ce rapport, nous nous concentrerons sur la distance entre le début et la fin car c'est le critère le plus difficile à optimiser et le plus intéressant à analyser.

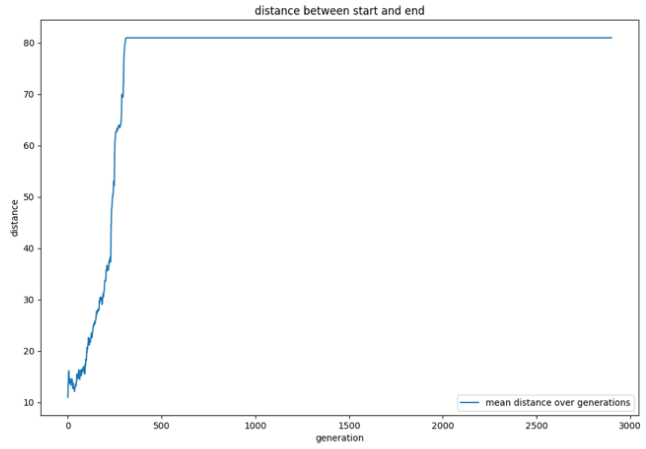
Hyper paramètres :
Les exécutions suivantes se sont déroulées avec une population de 500 individus et sur 4000 générations (3000 si la « fitness » n'évoluait plus). Les cartes générées sont de taille 25*25.

Paramètres à optimiser :
L'objectif de pourcentage à atteindre pour les cases « wall » est de 65%.

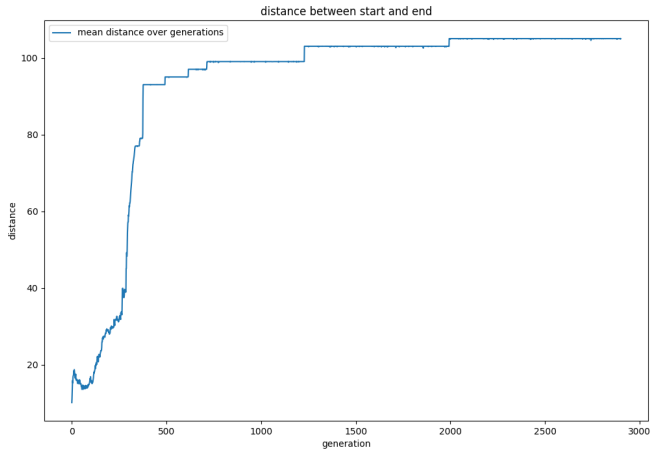
La distance idéale à atteindre est de 8*SIZE_X = 8*25 = 200.

Nous pouvons observer ci-dessous l'impact de la probabilité de mutation sur différents cas. Au-dessus de chaque courbe, on retrouve le taux de mutation.

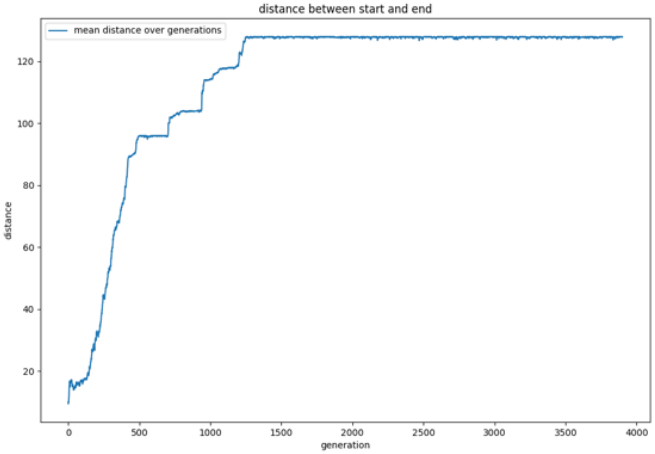
Sans mutation



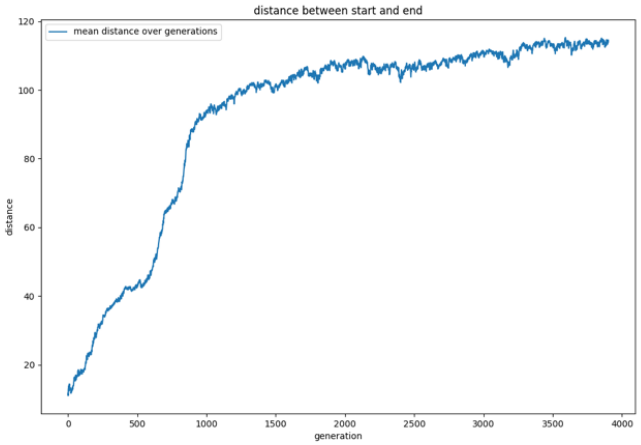
1e-5



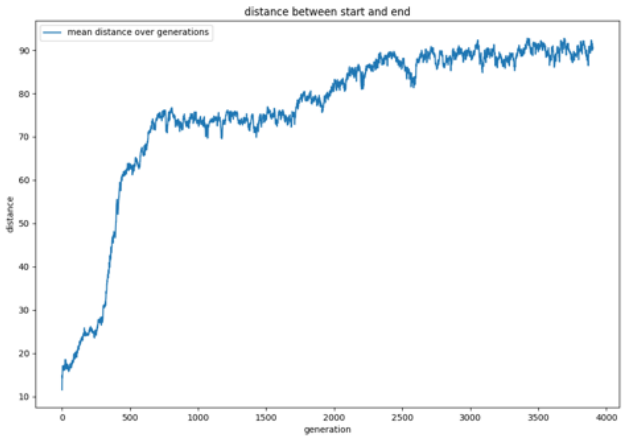
1e-4



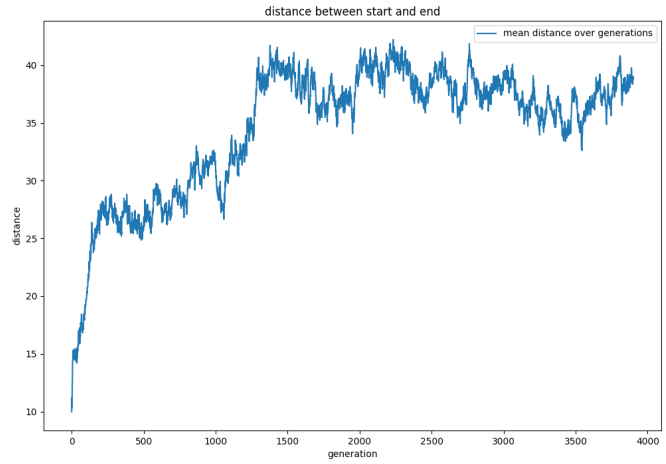
5e-4



1e-3

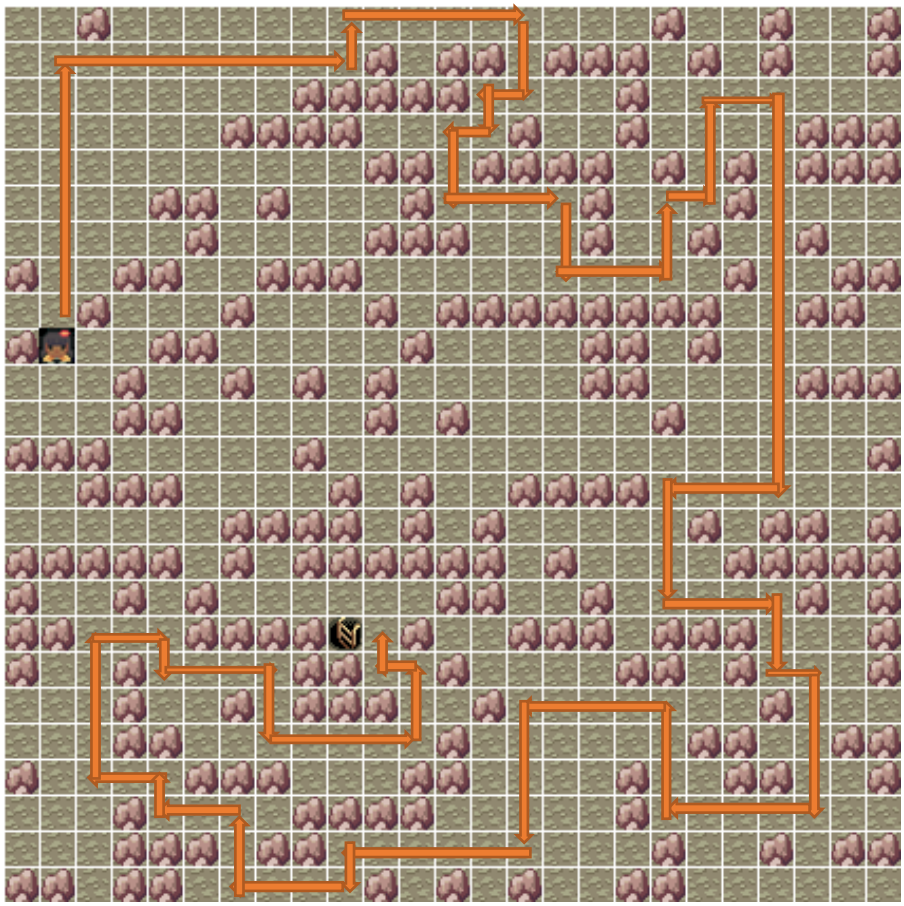


5e-3



Nous pouvons constater que sans mutation, l'algorithme tombe dans un minimum local au bout d'environ 400 générations.
Parmi les tests que j'ai fait la distance a été maximale pour un taux de mutation de 1e-4 avec une distance de presque 130.
On peut ensuite voir que si le taux de mutation est trop important, la courbe fluctue et a plus de difficulté à augmenter.

Voici un exemple de carte généré avec un taux de 1e-4 (Le chemin a été rajouté à la main) :



On constate que le chemin n'est pas linéaire, l'algorithme a réussi à utiliser les obstacles afin de rallonger la distance de parcours.

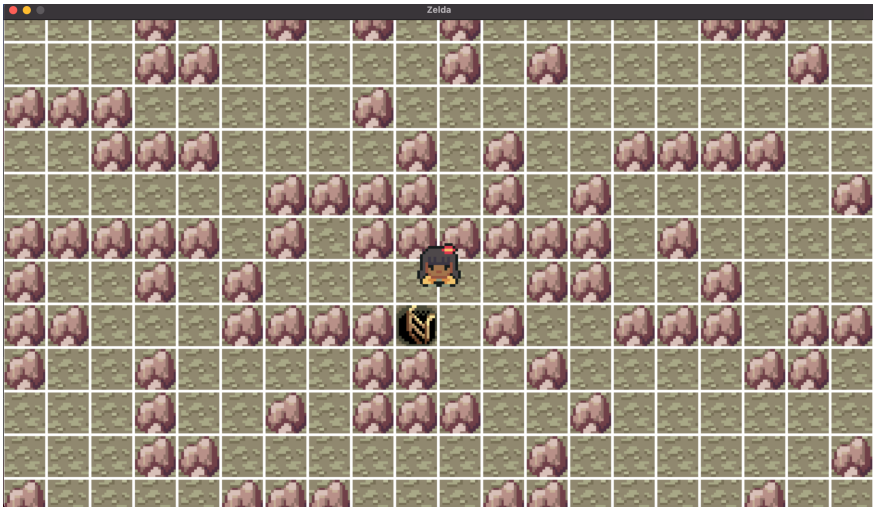


Figure 1:Image du jeu

On constate que la fin est bien atteignable =).

Conclusion :

Pour continuer ce TP, j'aurai aimé ajouter d'autres critères d'évaluation ainsi que d'autres éléments dans la carte comme des sprites servant à l'esthétisme de la grotte (on peut en trouver dans game/graphics/cave_images). On aurait également pu ajouter un critère servant à créer des zones facultatives à explorer (des chemins menant à un cul-de-sac pour y mettre des bonus optionnels pour le joueur).

J'ai apprécié travailler sur ce TP et je compte le continuer dans les semaines à venir.