Rapport Python Genetique

Leo Perrat

1/Présentation du programme

Dans le programme processing nous avons 3 classes:
_**Rectangle** : représente un rectangle qui sera utilisé comme gène

_**Individu** : représente un individu qui possède un nombre de gènes rectangles

_**Nurserie** : représente une population d'individus sur laquelle on va effectuer les différentes générations successives

Fonctionnement :
Quand on crée une nurserie elle est vide, on doit ensuite l'initialiser avec un nombre d'individus qui va rester le même pour chaque génération, les individus sont générés procéduralement en fonction des paramètres globaux au début du programme

C'est aussi le cas pour les individus, on doit les initialiser après leur création ils vont se voir attribuer un nombre de gènes générés procéduralement, on peut contrôler le nombre de gènes donnés à la création.

Une fois qu'on a initialisé la nurserie, on peut effectuer une nouvelle génération, elle va être créée de la sorte :
On calcule le coût, ou score de chaque individu, parmi ces individus on garde les nbCandidates meilleurs (nbCandidates passé en paramètre) et on crée la nouvelle génération en reproduisant les meilleurs candidats entre eux et à chaque reproduction il y a une chance que l'une des 3 mutations se produise

Chaque individu possède une fonction de coût ou de score, qui est similaire à celle qui nous a été fournie, il y a 3 mutations possibles

différente : mutationAdd, mutationDelete et mutationAddDelete qui combine les deux

2/Choix

Structures de données

La représentation sous forme de classe de Rectangle Individu et Nurserie m'a paru évidente, car ces concepts sont à la fois liés à des données (une nurserie contient des individus) et à des opérations (on effectue une génération à partir de la génération actuelle de la nurserie)

La seule exception aurait été pour Rectangle car à part les getters et les setters il n'a aucune méthodes, mais pour garder une uniformité j'ai considéré que c'était mieux

Aléatoire

Il y a beaucoup d'utilisation de l'aléatoire dans ce projet, mais la fonction random() de processing est assez limité, l'utilisation de bibliothèques externes aurait été utile, j'ai été confronté à deux problèmes :

_Dans la listes des meilleurs candidats à chaque génération, je veux sélectionner en priorité les meilleurs (par exemple que le le premier meilleur ai plus de chance que le second meilleur)

_Le pourcentage de chances de mutation défini juste si une mutation aura lieu ou non, et pas plusieurs

J'ai donc créé la fonction randomDistribution(distribution,maximum) qui pourrait se traduire par "combien de fois puis-je tomber sur face d'affilé en tirant à pile ou face" si le paramètre distribution est égal à 0.5, de manière générale, les probabilités sont les suivantes avec distribution = n :

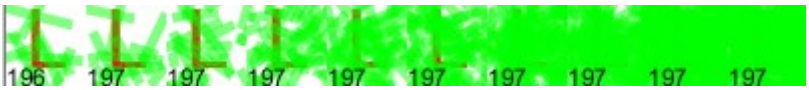
probabilité d'avoir 1 : n
probabilité d'avoir 2 : (1-n)*n
probabilité d'avoir 3 : (1-(1-n)*n)*n
etc...

0,5 et le meilleur paramètre pour distribution afin d'avoir une répartition plus équilibrée des probabilités de tomber sur chaque indice.
Cette fonction m'a permis de séléctionner un indice d'une liste de manière aléatoire avec plus de chance de tomber sur les premiers indices, ou que le pourcentage de chances de mutation ne représente pas juste la probabilité qu'une mutation se produise mais plusieurs (jusqu'à un maximum)

Fonction de coût/score

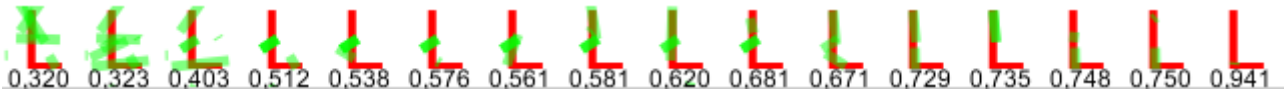
La fonction de coût utilisée ressemble beaucoup à celle fournie, à la différence que le calcul du coût était donné à faire.

La premiere stratégie a été de simplement compter le nombre de pixels composés d'un mélange de vert et de rouge : ce sont les pixels des rectangles qui ont été dessinés sur le glyphe, malheureusement si le score correspond uniquement à cette valeur, la solution la plus simple est de recouvrir toute la zone de dessin, quand un individu a un énorme rectangle qui recouvre tout le glyphe, les générations suivantes vont garder ce rectangle seulement.



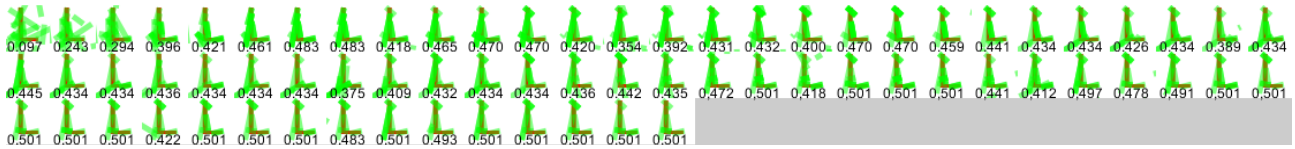
générations successives avec fonction de coût naïve

La deuxieme stratégie a été de compter négativement les pixels de l'individu en dehors du glyphe, donc les pixels verts uniquement, mais cette approche pose aussi probleme : les individus vont essayer de prendre le moins de risque et on se retrouve avec des individus dépourvus de rectangles



générations successives avec la deuxième fonction de coût

La troisième approche a été de compter négativement en plus du reste, les pixels du glyphe non recouverts, cette approche est la meilleure que j'ai trouvé jusqu'à présent.



générations avec la dernière fonction de coût

Autres problèmes rencontrés

Ce qui m'a le plus posé problème était la disparition des gènes, les individus avaient tendance et ce peut importe la fonction de coût, à perdre des gènes, j'ai finalement trouvé la solution : dans ma fonction de fusion de deux individu, on prend 50% des gènes des deux côtés, or la division étant entiere, on se retrouve de temps en temps à perdre un gène, et de manière statistique le nombre de gènes par individus diminue au fur et à mesure.

J'ai donc rajouté un gène en plus d'un des deux parents quand ils avaient un nombre de gènes impair.

Conclusion

Je ne suis pas allé très loin sur l'aspect génétique du projet car ce n'était pas demandé, mais j'ai pu remarquer que presque tout repose sur la fonction de coût, il faut tout faire pour éviter que les individus essaient de "tricher"