Université de Nantes
CLUZEL Ophélie
MAURICE Benjamin
Master MEEF Informatique

Rapport de Projet Python

2020-2021

UE Programmation

Introduction 2
Rappel des attendus 2
Organisation du travail 2
Choix des structures d'implémentation 2
Modélisation des gènes 2
Modélisation des Individus 3
Modélisation d'une Population 3
Les algorithmes génétiques 4
Sélection **Erreur ! Signet non défini.**
Reproduction Croisée 5
Mutation 6
La fonction de coût 7
Réglages possibles 8
Choix du nommage des variables 9
Présentation des résultats 9
Un exemple d'alphabet obtenu après 100 générations 9
Graphique d'évolution du coût par génération 10
Temps d'exécution 10
Exemples de population finale (avec leur coûts et leur temps d'exécution) 11
Conclusion, perspectives d'amélioration 11

I. Introduction

A) Rappel des attendus

Le but de ce projet était d'approcher au mieux un glyphe avec un certain nombre de rectangles. Pour cela nous avons dû suivre les principes des algorithmes génétiques tout en programmant dans le langage Python sur le logiciel Processing .

B) Organisation du travail

Comme nous avons dû travailler à distance, il nous a fallu impérativement utiliser un outil de partage de code. Nous avons donc travaillé avec Git et la plateforme GitHub (nous avions commencé avec celle-ci à laquelle nous étions déjà habitués et la présentation de GitLab est arrivée un peu tard), cependant nous avons importé le projet sur GitLab une fois celui-ci fini.
Lien GitHub: https://github.com/ophlapetite/ProjetPython.git
Lien GitLab: https://gitlab.univ-nantes.fr/E206103A/ProjetPython.git

Bien entendu, décrire les modifications apportées, documenter les fonctions et commenter le code est un travail nécessaire pour faciliter la compréhension du code et la reprise de celui-ci.

Pour le rapport et le diaporama de présentation nous travaillons sur google docs et google slides, pour pouvoir écrire en simultané sur nos documents.

II. Choix des structures d'implémentation

Nous avons choisi d'utiliser une modélisation orientée objet car c'est plus qu'utile dès lors que l'on doit représenter des données un peu plus complexes qu'un simple nombre, ou qu'une chaîne de caractères. D'autre part, la modélisation orientée objet permet d'avoir un code structuré, clair, modulable et facilement réutilisable, c'était le choix idéal pour un travail de groupe comme celui-ci.

A) Modélisation des gènes

Dans ce sujet les gènes sont en fait des rectangles. Ils ont plusieurs attributs qui les caractérisent:

-une orientation (8 orientations possibles multiples de $\frac{\pi}{4}$, valeur aléatoire)
-une largeur fixe (tous la même, valeur définie dans les réglages)
-une longueur variable (valeur aléatoire entre 5 et 20 qui est la moitié de la fenêtre)
-des coordonnées en x et en y (chacune ayant une valeur aléatoire entre 0 et 40, la taille de la fenêtre)

Nous avons donc créé une classe intitulée *Rectangle* avec son constructeur et les accesseurs en lecture et en écriture utiles.

```
class Rectangle:
    def __init__(self,i):
        '''
        Constructeur d'individu

        :param i: entier compris entre 1 et 8
        '''
        self.orientation=i*PI/4    # 8 orientations possibles, valeur i comprise entre 1 et 8
        self.largeur=largeurRect
        self.longueur=0
        self.x=0
        self.y=0
```

B) Modélisation des Individus

Les individus sont une sorte de “collection” de gènes. Dans ce sujet les individus seront donc une collection de rectangles : nous avons choisi de la représenter sous la forme d’une liste. Un des gros avantages des listes en Python est qu’il n’y a pas besoin de déclarer la taille du tableau à l’avance, il est aussi très facile et rapide d’accéder à ses éléments par indice.

Les individus ont aussi d’autres attributs:
- un coût qui lui sera attribué sur la base de ses gênes
-une image qui lui est associée (qui comporte le glyphe en rouge et ses rectangles en vert transparent)
-un numéro (cet attribut n’est pas indispensable mais nous l’avons ajouté pour avoir un moyen d’identifier l’individu lors de la sauvegarde de son image)

Nous avons donc créé une classe intitulée *Individu* avec son constructeur et les accesseurs en lecture et en écriture utiles.

```
class Individu:
    def __init__(self,n):
        '''
        Constructeur d'individu

        :param n: Le numéro de l'individu créé
        '''
        self.rectangles=[]
        self.cout=0
        self.numero=n
        self.img=createGraphics(imgWidth,imgHeight)
```

Cette classe contient également des fonctions qui vont permettre de générer un individu et des fonctions d’affichage(en vert avec le gyphe rouge ou seul en noir pour l’affichage final) et de sauvegarde de l’image.

C) Modélisation d’une Population

Une population est un ensemble d’individus. Nous avons choisi de le représenter sous la forme d’une liste d’Individus pour les mêmes raisons que précédemment mais également car il est très facile de récupérer des “tranches” d’une liste ce qui nous sera utile. Une population a aussi un attribut *num* qui représente le numéro de génération auquel elle correspond (combien de populations ont été générées avant elle).

Nous avons donc créé une classe intitulée *Population* avec son constructeur et les accesseurs en lecture et en écriture utiles.

```
class Population:


    def __init__(self,n):
        '''
        Constructeur d'une population

        :param n: Le numéro de la population créée
        '''
        self.individus=[]      #une liste d'individus initialisée à vide
        self.num=n             #numéro de génération correspondant
```

Cette classe contient également des fonctions qui vont permettre de générer une population, les différentes fonctions correspondant aux algorithmes génétiques (sélection, reproduction croisée et mutation) et une fonction d'affichage.

D) Les algorithmes génétiques

Maintenant que les différents "objets" du projet ont été modélisés, nous pouvons rentrer dans le vif du sujet.
Nous avons donc suivi le principe des algorithmes génétiques expliqué dans le sujet.
Nous commençons par générer une première population au hasard. Puis chaque prochaine population sera construite à partir de la précédente.

```
for i in range(nbGeneration):
    N += 1
    nouvellePop=Population(N)
    Pop.engendrePopulationSuivante()
    nouvellePop.individus = nurserie
    nouvellePop.stats()
    Pop.individus = nouvellePop.individus
```

Nous allons donc engendrer un certain nombre de générations de populations en appelant notre fonction *engendrePopulationSuivante()* . Cette fonction va appeler successivement nos fonctions de sélection, reproduction croisée et mutation.

```
def engendrePopulationSuivante(self):
    '''
    Engendre la population suivante à partir d'une population

    :return: la nouvelle population créée
    '''
    global nurserie

    nurserie = self.selection()
    nurserie += self.reproductionCroisee()
    nurserie += self.mutation()
```

➔ <u>Sélection</u>

Pour notre fonction sélection, nous faisons simplement un tri par ordre croissant de coût des individus de la population courante et nous récupérons les n premiers individus

qui seront placés dans la nurserie (n étant défini dans les réglages sous le nom de *nbSelection*). Nous avons choisi de représenter la nurserie également sous forme d'une liste.

```
def selection(self):
    '''
    Fonction qui va selectionner les n meilleurs individus d'une population
    '''
    popTriee = triFusion(self.individus)
    return popTriee[:nbSelection+1]
```

Pour le tri nous avons utilisé un tri fusion de part sa complexité moyenne intéressante en $\Theta(n \log n)$. Comme le tri va être appelé pour chaque population, si l'on envisage d'engendrer un grand nombre de génération il est préférable de choisir un tri qui a une bonne complexité.

```
def fusion(gauche,droite):
    resultat = []
    indexGauche, indexDroite = 0,0
    while indexGauche < len(gauche) and indexDroite < len(droite):
        if gauche[indexGauche].cout >= droite[indexDroite].cout:
            resultat.append(gauche[indexGauche])
            indexGauche += 1
        else:
            resultat.append(droite[indexDroite])
            indexDroite += 1

    if gauche:
        resultat.extend(gauche[indexGauche:])
    if droite:
        resultat.extend(droite[indexDroite:])

    return resultat

def triFusion(m):
    if len(m) <= 1:
        return m
    milieu = len(m)//2
    gauche = m[:milieu]
    droite = m[milieu:]
    gauche = triFusion(gauche)
    droite = triFusion(droite)
    return list(fusion(gauche, droite))
```

Dans un premier temps, la fonction *triFusion()* va découper récursivement la liste en deux sous-listes puis va les fusionner avec la fonction *fusion()* tout en les triant. On répète cela jusqu'à avoir une liste d'une seule case (donc triée par définition).

Voici un exemple de population après une première sélection:

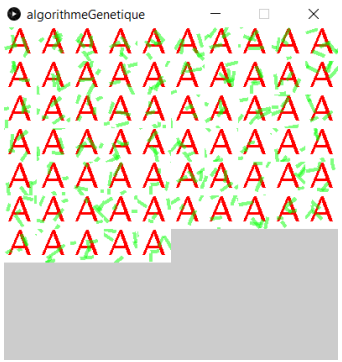
➔ <u>Reproduction Croisée</u>

Ensuite la fonction reproduction croisée telle que décrite dans le sujet doit tirer 2 individus au hasard dans la nurserie. Nous avons fait le choix de les tirer dans toute la population de départ et pas seulement dans la sélection des meilleurs individus stockée dans

la nurserie car les résultats s'avéraient plus satisfaisants, sans doute car la diversité était plus grande.
En partant de ces individus, on va créer un nouvel individu qui aura des gènes de ses deux parents. Pour la coupure nous avons fait le choix de la placer à la moitié des gènes de chaque parent (*nbRect*// 2, *nbRect* défini dans les réglages) il est important d'effectuer une division entière puisqu'on ne veut pas de "morceaux" de gènes. A la fin on choisit de remettre les parents tirés au sort pour qu'ils puissent être tirés au sort à nouveau.

```
def reproductionCroisee(self):
    '''
    Fonction qui va engendrer la génération suivante en faisant des croisements de 2 parents au hasard
    '''
    groupeParent = self.individus
    laSelection = []

    nb= nbReproCroisee
    for repro in range(nb):
        rectanglesEnfant = []

        indice1 = randint(0,len(groupeParent)-1)
        parent1 = groupeParent[indice1]
        groupeParent.pop(indice1)

        indice2 = randint(0,len(groupeParent)-1)
        parent2 = groupeParent[indice2]
        groupeParent.pop(indice2)

        rectParent1 = parent1.rectangles
        rectParent2 = parent2.rectangles

        coupure=nbRect//2

        rectanglesEnfant=parent1.rectangles[:coupure]+parent2.rectangles[coupure:]

        #on créé le fils résultant
        I=Individu(len(nurserie)+len(laSelection)+1)
        I.rectangles=rectanglesEnfant
        I.genereImg()
        I.calculCout()
      laSelection.append(I)

      #on remet les parents dans la liste à leur place initiale pour pouvoir les tirer au sort au prochain tour
      groupeParent.insert(indice1,parent1)
      groupeParent.insert(indice2,parent2)
```

Voici un exemple de population après une première sélection et reproduction croisée:



➔ <u>Mutation</u>

Pour finir nous avons fait une fonction de mutation.
Cette fonction prends n individus (n défini dans les réglages par *nbMutations*) au hasard dans la nurserie. Ensuite on va parcourir chaque rectangle de cet individu et on va au

hasard soit modifier sa longueur, son orientation, ses coordonnées ou ne rien faire. A la fin on calcule le nouveau coût de l'individu, si le coût est moins bon par rapport à un certain coefficient du coût initial , on recommence. Plus on augmente ce coefficient, plus le résultat est satisfaisant mais plus le temps d'exécution augmente.

```
def mutation(self):
    '''
    Un individu subissant une mutation génétique voit un ou plusieurs de ses gènes
    '''
    groupeMutation = nurserie[:] #copie sans pointeur
    enfants = []

    for i in range(nbMutation):
        indice = randint(0,len(groupeMutation)-1)
        indMutation = groupeMutation[indice]
        groupeMutation.remove(indMutation)
        coutInitial = indMutation.cout
        coutApres = -1
        I = Individu(len(nurserie)+len(enfants)+1)
        #coef modifiable
        while (coutApres < coutInitial*coefMutation):
            I.rectangles = indMutation.rectangles
            lesRects = []
            for rect in I.rectangles:
                mutation=randint(1,4)
                newRec = Rectangle(1)

                if(mutation==1):
                    #on change l'orientation
                    i = random(1,8)
                    newRec.orientation=i*PI/4
                    newRec.longueur = rect.longueur
                    newRec.x = rect.x
                    newRec.y = rect.y
                elif(mutation==2):
                    #on change la taille
                    newRec.setLongueur(random(5,20))
                    newRec.x = rect.x
                    newRec.y = rect.y
                    newRec.orientation = rect.orientation
                elif(mutation==3):
                    #on change les coordonnées
                    newRec.setX(random(0,40)) ; newRec.setY(random(0,40))
                    newRec.orientation = rect.orientation
                    newRec.longueur = rect.longueur
                elif(mutation==4):
                    i = random(1,8)
                    newRec.orientation=i*PI/4
                    newRec.setLongueur(random(5,20))
                    newRec.setX(random(0,40)) ; newRec.setY(random(0,40))
                lesRects.append(newRec)

            I.rectangles=lesRects

            I.genereImg()
            I.calculCout()
            coutApres = I.cout

        enfants.append(I)

    return enfants
```

Voici un exemple de population après une première sélection puis reproduction croisée puis mutation:

E) La fonction de coût

La fonction de coût est très importante dans un algorithme génétique puisque c'est elle qui va permettre de faire converger les résultats vers

le résultat escompté. Ici le but était d'avoir un coût élevé si l'individu (donc ses rectangles) recouvrait au mieux le glyphe.

Pour notre fonction de coût nous avons décidé de la calculer à partir de la proportion des couleurs des pixels. Nous commençons par compter les pixels blancs, rouges et verts transparents(couleur des rectangles) et nous comptons aussi tous les pixels restants (ils correspondent à la couleur obtenue avec la superposition des rectangles entre eux ou mélangés avec la couleur du glyphe). Ensuite nous calculons le coût à partir de ces couleurs récupérées .

Notre fonction de coût calcule le pourcentage des pixels de couleur autre (potentiellement ceux qui sont bien placés) par rapport à la somme des autres pixels de couleur (hors blanc) : les pixels rouges (partie du glyphe non recouverte), les pixels verts (partie des rectangles en dehors du glyphe) et les autres.

Notre fonction ci-dessous:

```
def calculCout(self):
    '''
    Fonction qui calcule le cout de l'individu en fonction de la répartition des couleurs des pixels de son image
    '''
    pB=0.0 #nb pixels blancs
    pR=0.0 # rouges
    pV=0.0 #verts
    pA=0.0 # autre couleur/mélange
    for i in range(0,len(self.img.pixels)):
        col=(red(self.img.pixels[i]), green (self.img.pixels[i]),blue(self.img.pixels[i]))
        if col==(255,255,255):
            pB=pB+1
        else:
            if col==(255,0,0):
                pR=pR+1
            else:
                if col==(128,255,128):
                    pV=pV+1
                else:
                    pA=pA+1

    self.cout=int((pA/(pR+pV+pA))*100)
```

Nous avons essayé plusieurs autres fonctions de coût qui n'étaient pas vraiment satisfaisantes.

F) Réglages possibles

Nous avons déclaré des variables globales dont il est possible de régler la valeur.

En effet, en fonction des réglages les résultats et le temps d'exécution sont plus ou moins satisfaisants. Les voici:

```
nbGeneration=100          → Le nombre de générations engendrées par le programme
nbRect=5                  → Le nombre de rectangles ou "gènes" par individu
largeurRect=4             → La largeur fixe des rectangles des individus
lettre='A'                → Le glyphe à approximer
imgWidth=40               → La largeur de l'image d'un individu
imgHeight=40              → La hauteur de l'image d'un individu
indParPopulation=100      → Le nombre d'individus par population
nbSelection=50            → Le nombre d'individus sélectionnés
nbReproCroisee=25         → Le nombre d'individus générés par reproduction croisée
nbMutation=25             → Le nombre d'individus générés par mutation
coefMutation=0.5          → Le coefficient de mutation
```

```
saveStats= False     - Sauvegarder ou non les statistiques liées au coût dans un
saveTmpExec= False   fichier csv
                     - Sauvegarder ou non les statistiques liées au temps
    d'exécution dans un fichier csv
```

Dans la pratique il vaut mieux éviter d'augmenter le réglage *indParPopulation* car notre affichage avec la fonction *drawPop()* est limité à 100 individus.Par contre cela ne poserait pas de problème pour l'affichage du meilleur individu. Cependant on peut tout à fait diminuer sa valeur, mais ce ne serait pas très intéressant. Il faut aussi que la somme de *nbSelection*, *nbReproCroisee* et *nbMutation* soit toujours égale à la valeur de *indParPopulation* pour éviter toute erreur.

G) Choix du nommage des variables

Nous avons essayé de rendre le nommage des variables, fonctions et classes le plus clair et explicite possible.
Nous avons aussi instauré certaines règles respectant les conventions de nommage :

-le nom des classes commence par une majuscule

-le nom des variables et fonctions commence par une minuscule mais s'il y a plusieurs mots les prochains commencent par une majuscule

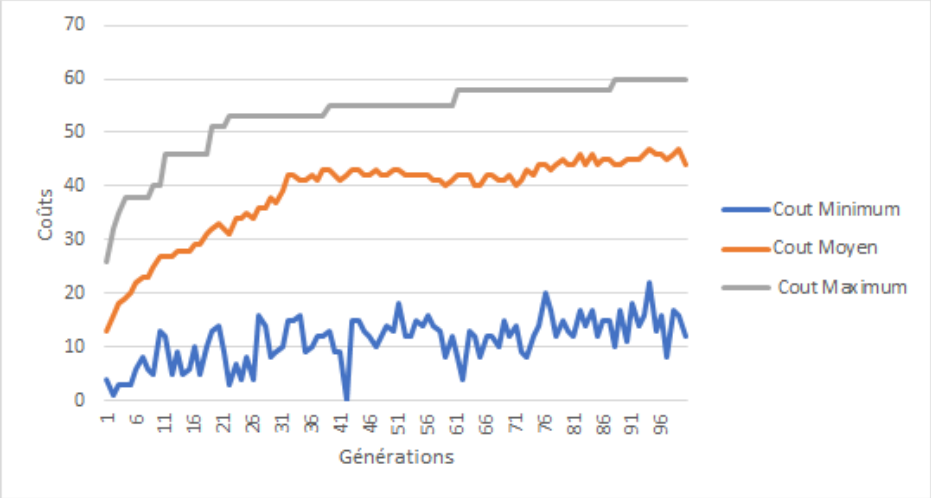
III. Présentation des résultats

- **Un exemple d'alphabet obtenu après 100 générations**

Pour réaliser cet alphabet nous avons affiché le meilleur individu obtenu après 100 générations de population pour tous les glyphes de A à Z.

- **Graphique d'évolution du coût au cours des générations**

Nous l'avons réalisé sous excel à partir d'un fichier csv rempli lors de l'exécution du programme (si la variable saveStats est à true).



- **Temps d'exécution**

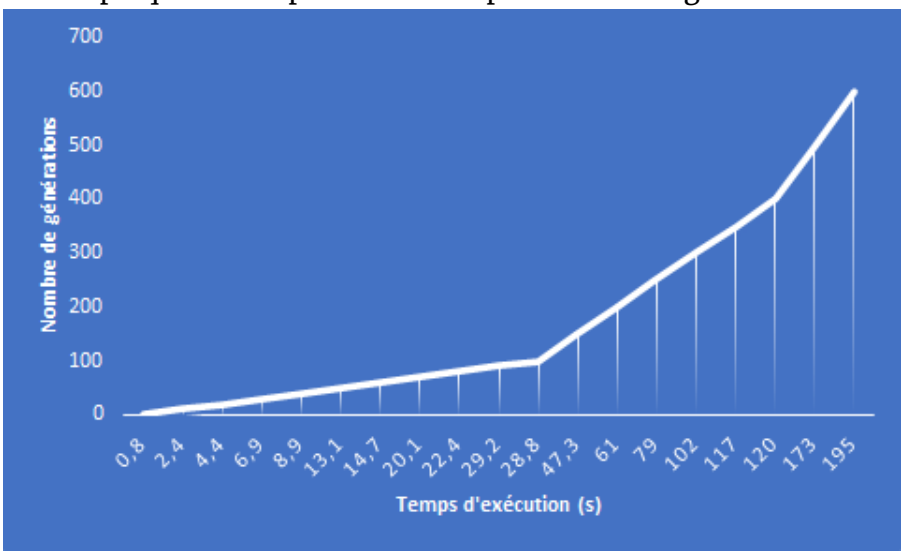
Nous calculons le temps d'exécution grâce à la méthode time.clock de processing. On récupère le temps avant de générer les populations et on récupère une nouvelle fois le temps après. Il suffit à la fin de calculer la différence entre les deux.

```
tempsExec = range(100000)
tps1 = time.clock()
tempsExec.sort()
```

…. code qui génère les populations…

```
tps2 = time.clock()
print("temps d'execution : ",tps2-tps1, " secondes.")
```

Graphique du temps d'exécution par nombre de générations:



Pour réaliser ce graphique nous avons stocké des valeurs de temps d'exécution et leur nombre de générations associées dans un fichier csv. Nous avons fait varier le nombre de générations nous même.

```python
if(saveTmpExec == True):
#sauvegarde temps d'execution dans fichier csv
    file=open("statsExec.csv","a")
    line=str(tmptot)+";"+str(nbGeneration)
    file.write(line)
    file.write("\n")
    file.close()
```

- **Exemples de population finale (avec leur coûts et leur temps d'exécution)**
  - Après 10 générations :


```
('Min : ', 17, ', Max : ', 63, ', coutMoyen : ', 42)
-----------
("temps d'execution : ", 9.518047000000024, ' secondes.')
```

  - Après 700 générations :


```
('Min : ', 13, ', Max : ', 87, ', coutMoyen : ', 57)
-----------
("temps d'execution : ", 322.48318040000004, ' secondes.')
```

V. Conclusion, perspectives d'amélioration

Pour finir nous pensons avoir respecté les attendus indiqués dans le sujet mais certaines choses comme le temps d'exécution pourraient être améliorées. Il reste tout de même raisonnable mais tout dépend de l'utilisation qu'on veut faire du programme.

Nous pourrions aussi "forcer" les mutations à être plus efficaces en faisant en sorte que les modifications sur les rectangles correspondent plus aux résultats attendus. Cela ferait converger plus vite les résultats vers une solution mais cela correspondrait beaucoup moins à une mutation "naturelle".

Pour conclure, nous avons découvert à travers ce projet la programmation génétique et nous avons trouvé ce sujet très intéressant. Quant au choix du logiciel imposé, Processing nous a permis, grâce à sa fenêtre graphique, d'avoir un très bon visuel sur nos résultats et ainsi de rapidement comprendre nos erreurs de programmation.