```
import random
import pickle
import numpy as np
import copy
import imageio
import matplotlib.pyplot as plt
#from multiprocessing import Process #multiprocessing permet de paralléliser les tâches , deja teste avec threading bien moins efficace
#import Array
import time
fig1, ax1 = plt.subplots()
fig2, ax2= plt.subplots()

def generer_circuit(image):
    img=imageio.imread(image)
    print(len(img),len(img[0]))
    matrice=np.zeros((len(img),len(img[0])))
    for i in range(len(img)):
        for j in range(len(img[0])):
            r,g,b=tuple(img[i][j][:3])
            if g>100 and b<10 and r<10: #prendre en compte des lignes vertes sur le circuit pour récompenser les bons individus
                matrice[i,j]=2
            elif r+g+b<10:
                matrice[i,j]=1
    print(np.shape(matrice))
    ax1.imshow(np.transpose(matrice))
    return matrice

global no_generation
no_generation =0
normalisation_poids=10
angles_vision=3
distance_vision=20

circuit=generer_circuit('circuit8.png')
#imageio.imwrite('test_circ.png',np.array(circuit))
#print(np.array(circuit[13:85][:100]))

position_initiale=(80,500)
dt=1/2
xmax,ymax=np.shape(circuit)
#circuit=np.zeros((xmax,ymax))
#circuit[55][50]=1
#for k in range(1000):
#   xx,yy=random.randrange(0,1000),random.randrange(0,1000)
#   circuit[xx][yy]=1
#   plt.plot(xx,yy,marker='o',markersize=4)

class Neurones:

    def __init__( self , tailles, fct_activation):

        ainit=[[random.random() for k in range(tailles[i])]for i in range(len(tailles))]
        self.a=np.array(ainit)                                          #a[i][j] valeur du neurone j de la couche i ,
                                                                        #ce n'est pas une matrice , on initialise aléatoirement
        self.tailles=tailles
        winit=np.empty(len(tailles)-1,dtype=object) #On initialise le tableau des poids , pour L couches , elle contient L matrices de poids entre la couche L+1 et L

        for l in range(len(tailles)-1):


            winit[l]=((np.random.rand(tailles[l],tailles[l+1]))-0.5)*normalisation_poids # np.random.rand(i,j) renvoie une matrice de nombres aléatoires
```

```
            entre 0 et 1 de taille i*j

        self.w=np.array(winit) #w[L][i,j] est le poids de la connexion entre le neurone i de la couche L et le neurone j de la couche L+1

        self.sig=fct_activation     #on appelle sig comme sigma la fonction d'activation du réseau
        self.memo={}    #memoisation pour ameliorer les performances

    def propagation(self,entree):
        if tuple(entree) not in self.memo: #si la valeur du calcul n'est pas connue on la calcule
            """Attention: l'entree sera une liste/un tableau ne pouvant pas etre utilise comme cle de dictionnaire
            c'est pourquoi il faut le transformer d'abord en tuple"""
            self.a[0]=copy.deepcopy(entree)
            for l in range(1,len(self.tailles)):
                #print(self.a[l-1],self.w[l])
                self.a[l]=self.sig(np.dot(self.a[l-1],self.w[l-1]))
                #éventuellement introduire une matrice de biais ici
            self.memo[tuple(entree)]=self.a[-1] #on ajoute la valeur calculee au dictionnaire pour une eventuelle utilisation future
            return self.a[-1]
        else:
            return self.memo[tuple(entree)]


def mutation(individu,nbmutat):
    for i in range(nbmutat): #on va modifier aléatoirement plusieurs poids du reseau neuronal de l'individu
        couche_random=random.randint(0,len(individu.reseau.w)-1)
        #print('len',len(individu.reseau.w)-1)
        #print('couche',couche_random)
        #print('poids',individu.reseau.w[couche_random])
        #print('type',type(individu.reseau.w[couche_random]))
        i_random=random.randint(0,len(individu.reseau.w[couche_random])-1)
        j_random=random.randint(0,len(individu.reseau.w[couche_random][i_random])-1)
        #rajout d'un nombre aleatoire decroissant a chaque gen pour affiner au fil des générations
        individu.reseau.w[couche_random][i_random,j_random]+= np.random.normal(scale=normalisation_poids/2)*.999**no_generation
        if individu.reseau.w[couche_random][i_random,j_random]>normalisation_poids:
            individu.reseau.w[couche_random][i_random,j_random]=normalisation_poids
        elif individu.reseau.w[couche_random][i_random,j_random]<-normalisation_poids:
            individu.reseau.w[couche_random][i_random,j_random]=-normalisation_poids


def reproduction(pere,mere):
    enfant=Vehicules(position_initiale)

    #on a deux opérateurs de reproduction choisis au hasard
    choix=random.random()
    #La reproduction barycentrique
    if choix > .5 :
        enfant.reseau.w = 0.5*(np.add(pere.reseau.w , mere.reseau.w ))
    #La reproduction par copie : l'enfant reçoit aléatoirement des poids du pere ou de la mere
    else:
        for i in range( len(enfant.reseau.w)):
            for j in range( len(enfant.reseau.w[i])):
                for k in range(len(enfant.reseau.w[i][j])):
                    p=random.choice([pere.reseau.w[i][j][k],mere.reseau.w[i][j][k]])
                    enfant.reseau.w[i][j][k]=p

    return enfant


def evolution(individus,distances):
```

```
118
119         distance_moyenne=sum(distances)/len(distances)
120         variance=sum([d**2-distance_moyenne**2 for d in distances])/len(distances)
121         print('moy=',distance_moyenne,'\n variance=',variance)
122
123         indices_tries=np.argsort(distances)
124         #argsort renvoie une liste contenant les indices des elements tries par ordre croissant sans modifier la liste
125         individus_tries=[individus[k] for k in indices_tries] #les individus sont ici mis dans l'ordre croissant
126         taux_mutation=0.2
127         taux_meilleurs=0.6 #proportion des meilleurs individus conservés pour la génération suivante
128         taux_sauvetage=0.05 #chances qu'un "mauvais" individu soit conservé
129
130         #on prend les meilleurs individus
131         nombre_meilleurs=int(taux_meilleurs*len(individus_tries))
132         parents=individus_tries[nombre_meilleurs:]
133
134         #on ajoute quelques individus moins bons au hasard pour la diversité
135         for k in range(len(individus_tries[:nombre_meilleurs])):
136             if random.random() < taux_sauvetage:
137                 parents.append(individus_tries[k])
138
139         enfants=[]
140
141         #on fait muter quelques parents au hasard
142         for k in range(len(parents)):
143             if random.random() < taux_mutation:
144                 mutation(parents[k],4)
145
146         while len(enfants)<len(individus):
147             enfants.append(reproduction(random.choice(parents),random.choice(parents)))
148
149         return enfants,distance_moyenne,variance
150
151     def sigmoide(x):
152         return 1/(1+np.exp(-x))
153
154     def tanh(x): #une autre fonction d'activation
155         a,b=np.exp(x),np.exp(-x)
156         return (a-b)/(a+b)
157
158     class Vehicules:
159         def __init__(self,position):
160             self.position=position
161             #vitesse maximale du vehicule , on multipliera cette valeur par la sortie du réseau
162             self.vmax=35
163             self.vitesse=0
164             self.angle=0.0
165             self.reseau=Neurones([angles_vision,9,2],sigmoide)#modifier le premier coefficient de la liste
166                                                               #en raccord avec le nombre de sorties de detect_entree
167             self.distance=0.0
168             self.vivant=True
169
170         def detect_entree(self,dmax,nbangles):
171             distances=[0.0 for i in range (nbangles)] #tableau qui contiendra la distance à un obstacle POUR CHAQUE ANGLE
172             angles=[i*np.pi/nbangles for i in range(-int((nbangles)/2),int((nbangles)/2))] #tableau des angles d'observation
173
174             for i in range(len(angles)): #on fixe un angle d'observation , on va regarder dans cette direction
175                 x,y=self.position       #on copie la position actuelle du véhicule
```

```
176                 for k in range(dmax):   #on itere jusqu'a dmax
177                     if x>=0 and x<xmax and y>=0 and y<ymax and not
                        circuit[int(x)][int(y)]==1: #conditions a laquelle on continue de
                        regarder dans la direction choisie
178
    # en gros tant qu'il n'y a pas d'obstacle ou qu'on ne tombe pas sur un mur
179                         x+=np.cos(self.angle+angles[i])     # x devient x+ projection
                            selon x dans la direction choisie
180                         y+=np.sin(self.angle+angles[i])     # y devient y+ projection
                            selon y dans la direction choisie
181                         distances[i]=k/dmax                 # la distance pour l'angle
                            d'observation choisi devient k/dmax pour avoir un quotient
                            entre 0 et 1
182
183             return np.array(distances) #on convertit en tableau numpy pour que la
                fonction de propagation puisse calculer rapidement dessus
184
185         def deplacement(self):
186             if self.vivant:
187                 #le véhicule "regarde" autour de lui : on veut lui donner des entrées
188                 entree=self.detect_entree(distance_vision,angles_vision)
189                 #print(entree)
190
191                 x,y=self.position
192
193                 #on calcule le résultat par le réseau de neurones
194                 resultat_reseau = self.reseau.propagation(entree)
195                 #print("res=",resultat_reseau)
196
197                 self.vitesse,self.angle=(resultat_reseau[0])*self.vmax,
                    (resultat_reseau[1]-0.5)*2*(np.pi/2)
198
199                 dx,dy=int(self.vitesse*dt*np.cos(self.angle)) ,
                    int(self.vitesse*dt*np.sin(self.angle))
200                 #print("d=",dx,dy)
201
202                 if x<0 or x>=xmax or y<0 or y>=ymax:
203                     self.distance -=200 #punir les individus qui rentrent dans les murs
204                 if x<0 or x>=xmax or y<0 or y>=ymax or circuit[x][y] or
                    (dx,dy)==(0,0) or self.distance< -3000:
205                     #dx,dy==0,0 est unecondition pour éviter les blocages,
206                     #on préferera que les individus soient constamment en mouvement
207                     if circuit[x][y]==2:
208                         self.distance+=10000
209                     else:
210                         self.distance-=4000
211                     self.vivant=False
212                 else:
213                     self.distance+=np.sqrt(dx**2+dy**2)
214                     self.position= x+dx,y+dy
215                     #xinit,yinit= position_initiale
216                     #self.distance = np.sqrt((xinit-(x+dx))**2+(yinit-(y+dy))**2)
217
218         def mort(self): #servira de condition de boucle pour le programme complet
219             return not self.vivant
220
221     def generation(la_horde,tracer):
222         nbvoit_vivantes=len(la_horde)
223         distances_par_vehicule=[]
224
225         tps=time.time()
226         for k in range(len(la_horde)):
227             vuatur=la_horde[k]
228             positions=[vuatur.position]
229             while nbvoit_vivantes>0 and not vuatur.mort():
230                 if vuatur.mort() :
231                     nbvoit_vivantes-=1
232                 elif (time.time()-tps)>3.0: #introduire une duree de vie limite,
                    certains individus sont sinon capables de ne jamais mourir
```

```
233                     print('je suis mort patron')
234                     vuatur.vivant=False
235                 else:
236                     vuatur.deplacement()
237                     #print(vuatur.reseau.w)
238                 positions.append(vuatur.position)
239                 #print(vuatur.distance)
240 
241             distances_par_vehicule.append(vuatur.distance)
242             x_val = [x[0] for x in positions]
243             y_val = [x[1] for x in positions]
244             if tracer or (time.time()-tps)>3.0:
245                 ax1.plot(x_val,y_val,marker='o')
246                 ax1.set_title("Deplacements")
247                 ax1.set_xlabel("x")
248                 ax1.set_ylabel("y")
249         #(sorted(distances_par_vehicule)[-nb_meilleurs:])
250         return la_horde,distances_par_vehicule,[x_val,y_val]
251 
252 
253         #plt.show()
254 
255 #class Generation_parallele(Process): #necessite de creer une sous classe de Process pour la parallelisation
256 #   def __init__(self,nb_individus,nb_meilleurs):
257 #       Process.__init__(self)
258 #       self.nb_individus = nb_individus
259 #       self.nb_meilleurs = nb_meilleurs
260 #       self.individus= []
261 #       self.pos_x, self.pos_y=[],[]
262 #
263 #   def run(self): #limitations de Process: run() effectue la tache du programme mais ne doit rien retourner
264 #                   # c'est pourquoi on utilise une classe dans laquelle on va modifier des propriétés
265 #       nb_individus=self.nb_individus
266 #       nb_meilleurs=self.nb_meilleurs
267 #
268 #       #N=Neurones([2,3,4,2],sigmoide,drv_sigmoide)
269 #       #print("resultat de la propagation",N.propagation(np.array([18,2])),"activations=",N.a)
270 #       la_horde=[Vehicules(position_initiale)for i in range(nb_individus)]
271 #       nbvoit_vivantes=len(la_horde)
272 #
273 #       for k in range(len(la_horde)):
274 #           vuatur=la_horde[k]
275 #           self.individus.append(vuatur)
276 #           positions=[vuatur.position]
277 #           while nbvoit_vivantes>0 and not vuatur.mort():
278 #               if vuatur.mort() :
279 #                   nbvoit_vivantes-=1
280 #               else:
281 #                   vuatur.deplacement()
282 #                   #print(vuatur.reseau.w)
283 #               positions.append(vuatur.position)
284 #
285 #               #print(vuatur.distance)
286 #
287 #           self.pos_x.append([x[0] for x in positions])
288 #           self.pos_y.append([x[1] for x in positions])
289 #       #plt.plot(x_val,y_val)
290 #       #print(sorted(distances_par_vehicule)[-nb_meilleurs:])
291 #       #plt.show()
292 #
293 #def generation_multithread(nbindividus,nbthreads):
294 #   threads=[Generation_parallele(nbindividus,5) for k in range(nbthreads)]
295 #   for thread in threads:
296 #       thread.start()
297 #
```

```
298  #   for thread in threads:
299  #       print(thread.individus)
300  #       thread.join()
301  #   individus_combines=[]
302  #   for thread in threads:
303  #       print(thread.individus)
304  #       individus_combines+=thread.individus
305  #
306  #   print(individus_combines)
307  #   """for thread in threads:
308  #       for k in range(len(thread.pos_x)):
309  #           plt.plot(thread.pos_x[k],thread.pos_y[k])
310  #   plt.show()"""
311  #
312  #generation(10,3)
313  #debut_tps=time.time()
314  #generation_multithread(40,3)
315  #print(time.time()-debut_tps)
316
317  def sauvegarde_generation(chemin,gen):
318      with open(chemin,'wb') as fichier :
319          pickle.dump(gen,fichier)
320          fichier.close()
321
322  def lecture_generation(chemin):
323      with open(chemin,'rb') as fichier:
324          generation_enreg=pickle.load(fichier)
325          return generation_enreg
326
327  debut_tps=time.time()
328  nbind=70
329  nbgen=100
330  individus,distances,plot= generation([Vehicules(position_initiale)for i in
     range(nbind)],5)
331  print(time.time()-debut_tps)
332  drap=False
333  liste_dist=[]
334  liste_var=[]
335  for k in range(nbgen):
336      no_generation=k
337      if k==nbgen-1:
338          drap=True
339      debut_tps=time.time()
340      new_gen,dmoy,variance=evolution(individus,distances)
341      individus,distances,plot=generation(new_gen,drap)
342      liste_dist.append(dmoy)
343      liste_var.append(variance)
344      #liste_temps.append(time.time()-debut_tps)
345      #if time.time()-debut_tps>3:
346      #   break
347      print('generation_numero:',no_generation,'temps=',time.time()-debut_tps)
348  sauvegarde_generation('generation',individus)
349  #plt.plot(plot[0],plot[1])
350  ax1.grid(True)
351  #fig2.subplot(211)
352  ax2.plot(range(no_generation+1),liste_dist)
353  ax2.set_title("Distance moyenne par génération")
354  ax2.set_xlabel("Génération")
355  ax2.set_ylabel("Distance moyenne (pixels)")
356  #fig2.subplot(212)
357  #ax2.plot(range(no_generation+1),liste_var)
358  plt.show()
359
360  #debut_tps=time.time()
361  #generation(new_gen,40,5)
362  #print(time.time()-debut_tps)
363  """ordres de grandeur de l'interet de la parallelisation:
364  Processeur: i5-2520m
365  120 individus
```

```
366   Programme classique: 9.26s
367   Programme parallele: 5.05s
368   """
369
```