```
'''
 Name : Elowan
 Creation : 02-06-2023 10:59:30
 Last modified : 27-04-2024 20:13:19
 File : main.py
'''
import datetime
import logging
from tqdm import tqdm
from multiprocessing import Pool, Lock, Manager
import sys

from Chromosome import *
from Models import Athlete
from Game import Game
from Genetic import GeneticAlgorithm
from utils import computeNextOccurrence
from traitement import analyseStudy, analyseFolder, createStats
from consts import NB_EVAL_MAX, PROBS_C, PROBS_M,\
   ITERATION_NUMBER, NUMBER_OF_CHROMOSOME_TO_KEEP,\
   INITIAL_POSITION, MAX_TICK_COUNT, SIZE_X, SIZE_Y,\
   POPULATIONS

# Variables communes à tous les processus pour connaître combien de ligne
# il faut sauter pour afficher chaque barre de progression
# Notamment utile pour empecher des sauts de lignes inopinés
position_Lock = Lock()
positions_bars = []

def playAllGames(population:list):
   """
   Joue toutes les parties associées aux athlètes de la population

   Params:
      population (AthleteChromosome list): liste des athlètes à faire jouer
   """
   # Supprime les anciens jeux
   Game.resetGames()

   for athleteChromosome in population:
      game = Game(athleteChromosome.athlete)
      game.play()


def logConstants(athleteLevel, seed):
   """
   Log les constantes de l'algorithme
   """
   logging.debug("Seed : {}".format(seed))
   logging.debug("Iteration number : {}".format(ITERATION_NUMBER))
   logging.debug("Athlete level : {}".format(athleteLevel))
   logging.debug("Number of chromosomes to keep : {}".format(NUMBER_OF_CHROMOSOME_TO_KEEP))
   logging.debug("Initial position : {}".format(INITIAL_POSITION))
   logging.debug("Size of the field : {}".format((SIZE_X, SIZE_Y)))
   logging.debug("Max tick count : {}".format(MAX_TICK_COUNT))
```

```
def replace_bars(i):
    """
    Descend les barres de progression d'une ligne dès qu'une des barres termine.
    """
    position_Lock.acquire()

    positions_bars[i] = 0
    for j in range(i+1, len(positions_bars)):
        positions_bars[j] -= 1

    position_Lock.release()

def process(args):
    """
    Fonction exécutant l'algorithme génétique pour une population de
    `population_number` individus et avec toutes les probabilités définies
    par le fichier `const.py`.

    Params:
        - args (tuple) : Contient `population_number` ainsi que `iteration`
                 représentant le i-ième appel à process
    """
    population_number, iteration = args
    count = 0
    total = len(PROBS_C)*ITERATION_NUMBER

    text = "Tests des probs sur une population de {0:04} individus".format(population_number)

    pbar = tqdm(total=len(PROBS_C)*ITERATION_NUMBER, unit="exec",
            desc=text, file=sys.stdout, position=positions_bars[i])

    for probs in zip(PROBS_C, PROBS_M):
        for _ in range(ITERATION_NUMBER):
            logging.debug("##### ITERATION {}/{} #####".format(count, total))
            logging.debug("Population number : {}".format(population_number))
            logging.debug("Probabilitées : Crossover = {}% Mutation = {}%"\
                    .format(probs[0]*100, probs[1]*100))
            logging.debug("Terminaison age : {}".format(NB_EVAL_MAX/population_number))

            ### Creation de la population
            # Chronométrage
            start_time = datetime.datetime.now()

            # population_number de fois le meme athlete
            population = [AthleteChromosome(
                       Athlete(athleteLevel))
                    for _ in range(population_number)]

            playAllGames(population)


            ### Algorithme génétique
```

```
        # Informations utilisées pour déterminer la terminaison
        # de l'algorithme (quand le maximum n'a pas été modifié depuis
        # un certain temps maxAge par exemple)
        infos = {
           "maxPopulationFitness": 0,
           "maxAge": 0,
           "generationCount": 0,
           "terminaison_age": NB_EVAL_MAX/population_number,
           "start_filenumber": iteration*total,
        }

        # Crée la variable l comme dans l'étude sélectionnée
        u = randint(0, 99)/100
        l = computeNextOccurrence(u, probs[1])

        # Ajout de paramètres supplémentaires
        def term(pop): return termination(pop, infos)
        def s(pop): return save(pop, probs, population_number, infos)
        def mut(pop): return mutation(pop, l)
        def cross(pop):
           children = crossover(pop, probs)

           # Duplications des enfants pour generer une population entière
           popu = []
           for _ in range(population_number//len(children)):popu.extend(children)
           return popu[:population_number]

        def iterate(population):
           evalPop = evaluate(population)
           infos["generationCount"] += 1

           # Mise a jour du score max des athlètes
           # et le temps depuis quand c'est le max
           if evalPop[0].fitness > infos["maxPopulationFitness"]:
              infos["maxPopulationFitness"] = evalPop[0].fitness
              infos["maxAge"] = 1

           else:
              infos["maxAge"] += 1


        parkourGenetic = GeneticAlgorithm(population, term, evaluate,
                             selection, cross, mut, s,
                             "data/{}".format(dirnameSaves))

        try:
           parkourGenetic.run(iteration=iterate)
           logging.debug("\nMeilleur athlète de la dernière génération:
{}".format(evaluate(parkourGenetic.population)[0]))
           logging.debug("Temps d'execution : {}".format(datetime.datetime.now() - start_time))

           count+=1
```

```
            pbar.pos = positions_bars[iteration]
            pbar.update(1)
            pbar.refresh()

        except Exception as e :
            logging.error("Erreur de l'appel à {} {}".format(population_number, iteration))
            logging.error(e)
            pbar.close()
            replace_bars(iteration)
            return

    replace_bars(iteration)



if __name__ == "__main__":
    s = 1713449159 # Pour avoir des résultats reproductibles
    # s = int(datetime.datetime.now().timestamp())
    seed(s)

    athleteLevel = 8
    dirnameSaves = "{}xp/{}".format(athleteLevel,
                          datetime.datetime.now()
                          .strftime("%d-%m-%Y %Hh%Mm%Ss"))

    dirs = "data/{}".format(dirnameSaves)
    os.makedirs('logs', exist_ok=True)

    # Initialisation des logs
    logging.basicConfig(level=logging.DEBUG,
                format='%(asctime)s - %(levelname)s - %(message)s',
                datefmt='%d-%m-%Y %H:%M:%S',
                filename='logs/Main - {}.txt'.format(str(athleteLevel) + "xp - "
                                + datetime.datetime.now()
                                .strftime("%d-%m-%Y %H:%M:%S")),
                filemode='w')

    # Affichage dans la console
    console = logging.StreamHandler()
    console.setLevel(logging.INFO)
    formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
    console.setFormatter(formatter)
    logging.getLogger('').addHandler(console)

    logConstants(athleteLevel, s)

    # Multi-Processing pour accélérer le temps d'exécution
    init_time = datetime.datetime.now()
    logging.info("Exécutions des algorithmes avec différentes tailles de population")

    # Initialisation des positions des barres
    positions_bars = Manager().list([i for i in range(len(POPULATIONS))])
```

```
# Lancement des processus
args = [(POPULATIONS[i], i) for i in range(len(POPULATIONS))]
with Pool(initializer=tqdm.set_lock, initargs=(tqdm.get_lock(),)) as p :
   p.map(process, args)

logging.info("Fin des exécutions. Créations des graphiques")

# Analyse du dossier (moyenne sur toutes les itérations)
data = analyseFolder(dirs)
createStats(path="{}/all".format(dirs), data=data)

# Dessine un graphe semblable à l'étude
analyseStudy(dirnameSaves)

logging.info("Temps d'execution total : {}".format(
   (datetime.datetime.now() - init_time)))
```