```
'''
 Name : Elowan
 Creation : 30-06-2023 23:56:45
 Last modified : 21-05-2024 12:02:45
 File : consts.py
'''
NUMBER_OF_CHROMOSOME_TO_KEEP = 20   # Nombre de chromosomes à garder à
                        # chaque génération

def MAX_SCORE(xp):                  # Score théorique maximal pour un niveau
   return 15 + 15*xp/10            # d'xp donné

EPS = 0.5                           # Epsilon interval autour du score max
atteignable

INITIAL_POSITION = (7, 31)          # Position initiale de l'athlète
MAX_TICK_COUNT = 70                 # Nombre de tours(=secondes) maximum

ITERATION_NUMBER = 1                # Nombre d'itérations de l'algorithme

TICK_INTERVAL = 1                   # Interval entre 2 executions de la
partie

CROSSOVER_PROB = 1                  # Probabilité de croiser deux parents
MUTATION_PROB = 0.05                # Probabilité de mutation d'un enfant

SIZE_X = 10                         # Taille du terrain
SIZE_Y = 40

# Valeurs utilisées dans l'étude
POPULATIONS = [2, 5, 10, 20, 35, 60, 100, 200, 300,
     450, 700, 1000, 1200, 1400, 1800, 2000]

# Distance en mètre maximal qu'un être humain peut parcourir en courant
pendant 1s
DIST_MAX = 6

# L le nombre de variables représentant un gène
L = 6*70

PROBS_C = [0.0, 0.0, 0.0, 0.9, 0.9] # Probabilité de croisement
PROBS_M = [0.1, 0.5, 1.0, 0.0, 0.1] # Probabilité de mutation

NB_EVAL_MAX = 45_000                # = S
```

```
'''
 Name : Elowan
 Creation : 02-06-2023 10:59:30
 Last modified : 18-05-2024 12:32:24
 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 avec population = {}; Iteration =
{}".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)))
```

```
'''
 Name : Elowan
 Creation : 08-06-2023 10:00:40
 Last modified : 21-05-2024 21:31:11
 File : Genetic.py
'''
import random

class Chromosome:
    """
        Classe abstraite représentant un chromosome (une entitée)
        de l'algorithme génétique

        Params:
            genes (Polymorphique): Variable représentant les caractéristiques
                du Chromosome
            fitness (int): Score attribué du chromosome
            age (int): Nombre de générations du chromosome
            size (int): Taille du chromosome
    """
    def __init__(self, genes, fitness, age, size):
        self.genes = genes
        self.fitness = fitness
        self.age = age
        self.size = size

    def __repr__(self) -> str:
        return "Fitness : {}".format(self.fitness)
```

```python
class GeneticAlgorithm:
    """
        Algorithme génétique adapté à un problème donné

        Params:
            population (Chromosome): liste de chromosomes
            termination (function): fonction qui renvoie true/false
                selon le critère de terminaison de l'algorithme
            evaluate (function): fonction qui évalue la population
            selection (function): fonction qui sélectionne les parents
            crossover (function): fonction qui crée les enfants
            mutation (function): fonction qui fait des mutations sur eux
    """
    def __init__(self, population:list, termination, evaluate,
                 selection, crossover, mutation, save, dirname="") -> None:

        # Renvoie true/false selon le critere de terminaison
        self.termination = termination

        # Fonction d'algo genetique
        self.evaluate = evaluate        # Tri la population
        self.selection = selection      # Selectionne les parents
        self.crossover = crossover      # Crée les enfants
        self.mutation = mutation        # Fait des mutations sur eux
        self.save = save

        self.population = population
        self.population_len = len(population)

        self.populationOverTime = [self.population]

        self.dirname = dirname

    def run(self, iteration=lambda x: None, callback=lambda x: None):
        """
            Execute l'algorithme génétique

            Params:
                ?iteration (function (GeneticAlgorithm): None): fonction qui
s'execute à chaque itération
                    de la boucle principale (Prend en paramètre l'instance de
l'algorithme génétique et
                    renvoie None)
                ?callback (function (GeneticAlgorithm): None): fonction qui
s'execute à la fin de
                    l'algorithme (Prend en paramètre l'instance de l'algorithme
génétique et renvoie None)
        """
        self.population = self.evaluate(self.population)
        while not self.termination(self.population):
            iteration(self.population)

            self.population = self.selection(self.population)
            self.population = self.crossover(self.population)
            self.population = self.mutation(self.population)
            self.population = self.evaluate(self.population)

            # Sauvegarde des données pour la sérialisation
            self.populationOverTime.append(self.population)

        self.save(self)
        return callback(self.population)

    def getFilename(self):
        return self.filename

    def getDirname(self):
        return self.dirname

if __name__ == "__main__":
    random.seed(22)

    # Test de l'algorithme génétique avec le problème OneMax
    class OneMaxChromosome(Chromosome):
        def __init__(self, genes: list):
            self.genes = genes
            super().__init__(self.genes, self.calc_fitness(), 0,
len(self.genes))

    def calc_fitness(self):
        sum_gene = 0
        for i in range(len(self.genes)):
            sum_gene += int(self.genes[i])

        return sum_gene

    def mix(self, gene1, gene2):
        self.genes = gene1+gene2
        self.fitness = self.calc_fitness()
        self.age += 1

    def __repr__(self):
        string = ""
        for i in range(len(self.genes)):
            string += str(self.genes[i])

        return string

def sumlist(population):
    return [chro.fitness for chro in population]

def swap(i, j, liste):
    temp = liste[i]
    liste[i] = liste[j]
    liste[j] = temp

    return liste

def evaluate(population):
    # Evaluation de la population
    sums = sumlist(population)

    # Sort list decreasing
    for i in range(len(sums)-1):
        max = i
        for j in range(i+1, len(sums)):
            if sums[j] >= sums[max]:
                max = j

        if max != i:
            sums = swap(i, max, sums)
            population = swap(i, max, population)

    return population

def selection(population):
    # Selection des parents
    # On fait des pairs de chaque element
    liste = []
    for i in range(0, len(population)-1, 2):
        liste.append((population[i], population[i+1]))

    return liste

def crossover(population):
    # Creation des enfants
    cross_point = random.randint(0, 1000)
    liste = []
    for i in range(len(population)):
        chro1 = population[i][0]
        chro2 = population[i][1]
        chro1.mix(chro1.genes[:cross_point], chro2.genes[cross_point:])
        chro2.mix(chro1.genes[:cross_point], chro2.genes[cross_point:])

        liste.append(chro1)
        liste.append(chro2)

    # print("".join([ str(x) for x in liste[0] ]))
    return liste

def mutation(population):
    liste = []
    for pop in population:
```

```
        l = pop
        if random.randint(0, 100) < 5:
            random.shuffle(l.genes)

        liste.append(l)

    return liste

def termination(population):
    sums = sumlist(population)
    best = sums.index(max(sums))
    return population[best].fitness >= 1000

def iteration(population):
    sums = sumlist(population)
    best = sums.index(max(sums))

    print("Current best : {}".format(population[best].fitness))

def save(population): pass

populationChromosome = [ OneMaxChromosome([ random.randint(0, 1)
                            for x in range(1000) ])
                for y in range(100) ]
OneMaxProblem = GeneticAlgorithm(populationChromosome, termination,
evaluate,
                selection, crossover, mutation, save)

OneMaxProblem.run(iteration)

sums = sumlist(OneMaxProblem.population)
best = sums.index(max(sums))
print("Best overall : {}\nAge :
{}".format(OneMaxProblem.population[best],
OneMaxProblem.population[best].age))
```

```
'''

Name : Elowan
Creation : 02-06-2023 11:00:05
Last modified : 21-05-2024 21:31:06
File : Game.py
'''

from Terrain import Field
from Models import Athlete, FIGURES

from consts import INITIAL_POSITION, MAX_TICK_COUNT, TICK_INTERVAL

class Game:
    """
    Classe représentant un round de la compétition
    """

    instances = []
    def __init__(self, athlete):
        self.field = Field()
        self.field.createField()
        self.athlete = athlete
        self.state = 0          # Etat de la partie
        self.tickCount = 0      # 1 tick = 1 seconde

        self.athlete.setField(self.field)

        Game.instances.append(self)

    def start(self):
        """Initialisation des valeurs de depart de la competition"""
        self.tickCount = 0
        self.state = 1
        self.athlete.position = INITIAL_POSITION

    def update(self):
        """Met à jour l'état de l'athlète et retourne l'état de la
compétition"""
        if self.tickCount >= MAX_TICK_COUNT:
            self.end()
            return self.state

        self.athlete.takeAction(self.tickCount)

        self.tickCount += TICK_INTERVAL
        return self.state

    def end(self):
        """Fonction appelée lorque la competition termine"""
        self.state = 2
        # # Si la suite de combos ne rempli pas entièrement le nombre de combos
        # # disponible, on duplique le dernier combo (en supprimant la figure
puis le tick)
        # if len(self.athlete.combos) < 70:

        #     n = 70 - len(self.athlete.combos)

        #     pos, _, _ = self.athlete.combos[-1]

        #     for _ in range(n):
        #         self.athlete.combos.append((pos, FIGURES["run"], 0))

    def play(self, iterate=lambda x: None, callback=lambda x: None):
        """Fait faire une partie entière au jeu
        Params:
            iterate (function)  - Prend en paramètre l'instance du jeu et ne
                                  renvoie rien. Elle est executee a chaque
                                  tick de la partie

            callback (function) - Prend en paramètre l'instance du jeu et ne
                                  renvoie rien. Elle est executee a la fin de
                                  la partie

        """
        self.start()
        while self.update() == 1:
            iterate(self)

        callback(self)

    def _getGameByAthlete(athlete):
        for i in Game.instances:
            if i.athlete.id == athlete.id:
                return i

    def resetGames():
        """Supprime toutes les instances de Game"""
        Game.instances = []

if __name__ == "__main__":
    athlete = Athlete(5, FIGURES["backflip"])
    game = Game(athlete)
    def iterate(game):
        print("Athlete state (in the second {}) : ".format(game.tickCount))
        print("  - Position ({},{})".format(
            game.athlete.position[0], game.athlete.position[1]
        ))
        print("  - Case : {}".format(
            game.field.getCase(game.athlete.position).name
        ))
        print("  - Current movement : {} since {} seconds".format(
            game.athlete.state["movement"],
game.athlete.state["ticksSinceStartedMoving"]
        ))
        print()

    def callback(game):
        print("Game state : {}\nFor {} ticks".format(game.state,
game.tickCount))
        print("Combos : {}".format(athlete.combos))

    print("Game started !")
    game.play(iterate=iterate, callback=callback)
```

```
'''

Name : Elowan
```

```
Creation : 23-06-2023 11:42:17
Last modified : 21-05-2024 12:02:40
File : Chromosome.py
'''
from random import randint, seed, choice
import logging
from math import sqrt, ceil
import json
import os

from Terrain import FIGURES
from Models import Athlete, Figure
from Game import Game
from Genetic import Chromosome
from consts import INITIAL_POSITION, NUMBER_OF_CHROMOSOME_TO_KEEP,\
   EPS, MAX_SCORE, L, SIZE_X, SIZE_Y, DIST_MAX

k = 0
i = 0

class AthleteChromosome(Chromosome):
   """
      Classe représentant un athlète (une entitée) pour l'algorithme
      génétique

      Params:
         athlete (Athlete): Athlète représenté par le chromosome
   """
   def __init__(self, athlete):
      self.athlete = athlete
      self.genes = from_combo_to_string(athlete.combos)
      self.detailedFitness = {}

      super().__init__(self.genes, self.calc_fitness(),
                  0, len(self.genes))

   def calc_fitness(self) -> int:
      """Calcule le score de l'athlète"""
      score =  {
         "execution": {
            "safety": 3,
            "flow": 0,
            "mastery": 0,
         },
         "composition": {
            "use_of_space": 0,
            "use_of_obstacles": 0,
            "connection": 0,
         },
         "difficulty": {
            "variety": 0,
            "single_trick": 0,
            "whole_run": 0,
         },
      }
      self.genes = from_combo_to_string(self.athlete.combos)

      # Liste des figures faites
      nb_figure = len(self.genes)//6
      tricks = [Figure.figures[int(self.genes[6*i+4: 6*i+6])]
            for i in range(nb_figure)]
      field = self.athlete.field
      try:
         cases = [field.getCase(
            (int(self.genes[6*i:6*i+2]), int(self.genes[6*i+2: 6*i+4])))
            for i in range(nb_figure)]
      except IndexError:
         for i in range(nb_figure):
            print((int(self.genes[6*i:6*i+2]), int(self.genes[6*i+2: 6*i+4])))
            return 0


      # Calcul de la sureté des figures
      # On coefficiente la sureté par l'xp de l'athlète
      score["execution"]["safety"] =\
         (score["execution"]["safety"])*self.athlete.xp/10

      # Calcul du flow
      # Compte le nb de fois qu'on s'est arreté
      score["execution"]["flow"] = 3 - tricks.count(FIGURES["do_nothing"])

      # Calcul de la maitrise
      # Max 4
      score["execution"]["mastery"] = 4*self.athlete.xp/10

      # Calcul de l'utilisation de l'espace
      # Compte le nb de cases différentes utilisées
      # Post-it : Identifiant d'une case est unique donc on ne compte que
      #           les cases visitées (dans notre liste de cases)
      # Max 3 (comme le nb de cases)
      l = []
      for case in cases:
         if case.id not in l: l.append(case.id)
      score["composition"]["use_of_space"] = len(l)


      # Calcul de l'utilisation des obstacles
      # Compte le nb de types de cases différents utilisés
      # Max 3 (comme le nb de cases)
      l = []
      for case in cases:
         if case.name not in l: l.append(case.name)
      score["composition"]["use_of_obstacles"] = len(l)

      # Calcul de la connexion entre les obstacles
      # Max 4
      score["composition"]["connection"] = 4*self.athlete.xp/10

      # Calcul de la variété
      # Ajoute 1 pts a chaque figure de complexité >= 2
      # (Donc que le trick n'est pas ds la catégorie de parkour classique)
      score["difficulty"]["variety"] = sum([1
                  for trick in tricks
                  if trick.complexity >= 2])

      if score["difficulty"]["variety"] > 3:
         score["difficulty"]["variety"] = 3

      # Calcul de la difficulté d'un trick
      # Ajoute 1 pt par trick de complexité >= 3
      score["difficulty"]["single_trick"] = sum([1
                  for trick in tricks
                  if trick.complexity >= 3])

      if score["difficulty"]["single_trick"] > 3:
         score["difficulty"]["single_trick"] = 3

      # Calcul de la difficulté d'un run
      score["difficulty"]["whole_run"] = 4*self.athlete.xp/10

      # Calcul du score final
      self.detailedFitness = score
      self.fitness = sum([sum(score["execution"].values()),
                  sum(score["composition"].values()),
                  sum(score["difficulty"].values())])

      if self.fitness < 0:
         self.fitness = 0

      # print(self.fitness)
      return self.fitness

   def __repr__(self) -> str:
      return "AthleteID {} de score {} d'age {} et de taille {} : \n{}"\
         .format(self.athlete.id, round(self.fitness, 2),
               self.age, self.size, self.athlete)

def evaluate(population:list) -> list:
   """
   Notation de chaque athlète de la population

   Params:
      population (AthleteChromosome list): liste d'athlètes
```

```
    Returns:
        population (AthleteChromosome list): liste d'athlètes triés
            (décroissant) par score
    """
    population.sort(key=lambda x: x.calc_fitness(), reverse=True)
    return population

def selection(population:list) -> list:
    """
    Selectionne les parents de la prochaine population
    Parents = 10 premiers en score de la population actuelle

    Params:
        population (AthleteChromosome list): liste d'athlètes

    Returns:
        (AthleteChromosome list): liste d'athlètes sélectionnés
    """
    return population[:10]

def get_point_communs(a1, a2) -> tuple[int, int]:
    """
    Renvoie les indices où les deux athlètes sont au même point dans leur
course,
    différent de (0, 0).
    Si renvoie (-1, -1) alors il n'y en a pas

    Params :
        a1 (AthleteChromosome) : Athlète
        a2 (AthleteChromosome) : Athlète

    Returns:
        int: indice
    """
    for i in range(0, len(a1.genes), 6):
        for j in range(0, len(a1.genes), 6):
            if a1.genes[i: i+6] == a2.genes[j: j+6]: return (i, j)
    return (-1, -1)

def copy_chromosome(parent):
    """
    Duplique littéralement un chromosome en augmentant son age

    Params :
        parent (AthleteChromosome) : Parent

    Returns:
        AthleteChromosome: Duplica
    """
    child = Athlete(parent.athlete.xp)
    child.combos = parent.athlete.combos
    child.setField(parent.athlete.field)
    childChro = AthleteChromosome(child)
    childChro.age = parent.age + 1
    return childChro

def new_children_crossover(p1, p2, cross_prob):
    """
    Renvoie deux nouveaux chromosomes enfants des deux parents p1 et p2
    selon la méthode de croisement et la probabilité de croisement cross_prob

    Params :
        p1 (AthleteChromosome) : Parent
        p2 (AthleteChromosome) : Parent

    Returns:
        (AthleteChromosome, AthleteChromosome): Les enfants
    """
    c1, c2 = get_point_communs(p1, p2)

    if c1 != -1 and c2 != -1\
              and randint(0, 100)/100 < cross_prob:

        # Premier enfant, avec un premier croisement des combos
        child1 = Athlete(p1.athlete.xp)
        child1.setField(p1.athlete.field)
        child1.combos = from_string_to_combos(p1.genes[:c1]+p2.genes[c2:])
        childChro1 = AthleteChromosome(child1)

        # Deuxieme enfant, avec le croisement complémentaire au premier
        child2 = Athlete(p2.athlete.xp)
        child2.setField(p2.athlete.field)
        child2.combos = from_string_to_combos(p2.genes[:c2] + p1.genes[c1:])
        childChro2 = AthleteChromosome(child2)
        return childChro1, childChro2

    else:
        return copy_chromosome(p1), copy_chromosome(p2)

def crossover(parents: list, probs) -> list:
    """
    Crée les enfants de la prochaine population
    On choisit 2 parents et on les on prend 2 moins communs aux deux
    chemins (s'il y a) et on échange les chemins entre ces deux points

    Si les deux parents sont en réalité le même, on le copie tel quel.

    Params:
        parents (AthleteChromosome list): liste d'athlètes

    Returns:
        children (AthleteChromosome list): liste d'athlètes enfants
    """
    children = []
    CROSSOVER_PROB, _ = probs
    for i in range(0, len(parents)-1, 2):
        c1, c2 = new_children_crossover(parents[i], parents[i+1],
                                        CROSSOVER_PROB)
        children.append(c1)
        children.append(c2)

    return children

def dist(x1, y1, x2, y2):
    """Calcule la distance entre 2 points dans le plan"""
    return sqrt((x2-x1)**2 + (y2-y1)**2)

def coherence_suite_etats(e1, e2, e3):
    """
    Vérifie la cohérence des l'état e2 provenant de l'état e1 et allant
    à l'état e3

    Params:
        e1/e2/e3 (str): String de 6 caractères représentant un état

    Returns:
        (bool): Valide ou non
    """
    x1 = int(e1[0:2])
    x2 = int(e2[0:2])
    x3 = int(e3[0:2])

    y1 = int(e1[2:4])
    y2 = int(e2[2:4])
    y3 = int(e3[2:4])

    return dist(x1, y1, x2, y2) <= DIST_MAX and dist(x2, y2, x3, y3) <=
DIST_MAX

def mutation_individual(athleteChromosome: AthleteChromosome, k:int):
    """
    Mutation en place de l'athlete `athleteChromosome` passé en paramètre.
    Le caractère du gene à modifier est imposé par le paramètre `k` contenu
    entre 0 et 419 inclus.
    """
    # k = Indice du caractère à modifier
    # i = Indice du gene contenant la variable
    k = k%len(athleteChromosome.genes)
    i = (k - k%6)//6

    # Etat associé au gène
    e = athleteChromosome.genes[i*6: (i+1)*6]

    # Positions et figure associé à l'état
```

```
x = int(e[0:2])
y = int(e[2:4])
f = int(e[4:6])

modifieur = choice([-1, 1])

# Récupération des états précédant et succédant l'état à l'étude
if i == 0 :
   e1 = e
else:
   e1 = athleteChromosome.genes[(i-1)*6: i*6]

if i >= len(athleteChromosome.genes)//6 - 1:
   e3 = e
else:
   e3 = athleteChromosome.genes[(i+1)*6: (i+2)*6]

has_mutated = True

# Match sur la composante qui va être modifiée
match (k%6)//2:
   case 0 : # Si on modifie la variable de l'abscisse
      # Le modifieur étant choisi avant le match, on vérifie que
      # la modification apporté à l'abscisse n'enfreint aucune
      # des conditions de bons fonctionnements comme :
      # 0 <= x < SIZE_X et que le déplacement à cette case depuis
      # la case précédente e1 est possible et le deplacement vers e3,
      # assuré par le renvoie "true" de la fonction coherence_suite_etats
      e2 = from_combo_to_string(
         [((x+modifieur, y), Figure.getFigureById(f), 0)])

      if x + modifieur >= 0 and x + modifieur < SIZE_X:
         if coherence_suite_etats(e1, e2, e3):
            x += modifieur

         else:
            if x-modifieur >= 0 :
               e2_recovery = from_combo_to_string(
                     [((x-modifieur, y), Figure.getFigureById(f), 0)])

               if coherence_suite_etats(e1, e2_recovery, e3):
                  x -= modifieur

               else: has_mutated = False

      else:
         if x-modifieur >= 0 and x-modifieur < SIZE_X:
            e2_recovery = from_combo_to_string(
               [((x-modifieur, y), Figure.getFigureById(f), 0)])

            if coherence_suite_etats(e1, e2_recovery, e3):
               x -= modifieur
            else: has_mutated = False
         else: has_mutated = False


   case 1: # Sensiblement la même chose que précédemment mais pour l'ordonné
      e2 = from_combo_to_string(
         [((x, y+modifieur), Figure.getFigureById(f), 0)])

      if y + modifieur >= 0 and y + modifieur < SIZE_Y:

         if coherence_suite_etats(e1, e2, e3):
            y += modifieur

         else:
            if y-modifieur >= 0 :
               e2_recovery = from_combo_to_string(
                     [((x, y-modifieur), Figure.getFigureById(f), 0)])

               if coherence_suite_etats(e1, e2_recovery, e3):
                  y -= modifieur
               else: has_mutated = False


      else:
         if y-modifieur >= 0 and y-modifieur < SIZE_Y:
            e2_recovery = from_combo_to_string(
               [((x, y-modifieur), Figure.getFigureById(f), 0)])

            if coherence_suite_etats(e1, e2_recovery, e3):
               y -= modifieur
            else: has_mutated = False
         else: has_mutated = False

   case 2: # Cas du changement de la figure
      if f + modifieur >= len(FIGURES) or f+modifieur < 0 :
         f -= modifieur
      else:
         f += modifieur

# Reconstruction du gène
gene = athleteChromosome.genes[0: i*6] +\
      from_combo_to_string([((x, y), Figure.getFigureById(f), 0)])+\
      athleteChromosome.genes[(i+1)*6:]

# Modification en place de l'athlete
athleteChromosome.genes = gene
athleteChromosome.athlete.combos = from_string_to_combos(gene)

return has_mutated
```

```
def mutation(population:list, l: int) -> list:
   """
   Fait muter la `population`, en ajoutant 1 ou -1 à un gène aléatoire
   (position x, position y ou l'indentifiant de la figure) selon le dernier muté
   et du paramètre `l` (Mutation Clock operation) et la cohérence de
   ce changement avec le modèle réel

   Params:
      population (AthleteChromosome list): liste d'athlètes
      l (int): nombre associé à une probabilité selon la 2nd étude sur les GAs

   Returns:
      population (AthleteChromosome list): liste d'athlètes enfants

   """
   global k, i
   has_mutated = mutation_individual(population[i], k)

   if not has_mutated:
      i = (i+1)%len(population)

   else:
      k = int((k+l)%L)
      i = (i + ceil((k+l)/L))%len(population)

   return population

def termination(population:list, infos) -> bool:
   """
   Condition d'arrêt de l'algorithme génétique

   Params:
      population (AthleteChromosome list): liste des athlètes

   Returns:
      (bool): True si l'algorithme doit s'arrêter, False sinon
   """
   return infos["generationCount"] > infos["terminaison_age"] or \
      MAX_SCORE(population[0].athlete.xp) - EPS < infos["maxPopulationFitness"]

def getBestAthlete(population):
   """
   Affiche le meilleur athlète de la population

   Params:
      population (AthleteChromosome list): liste des athlètes
   """
   evalPop = evaluate(population)
```

```
    logging.info(evalPop[0])

def from_combo_to_string(combos) -> str:
    """
    Changement de représentation de la suite de combos en une chaîne de
    caractères pour la représentation des gènes d'un chromosome

    Params:
        combos (tuple list): liste des combos sous la forme
            [((x, y), Figure, tickStarted)]

    Returns:
        str: concaténation de chaque figure codée sur 6 caractères. Par exemple
            "xxyyii" pour la figure d'identifiant i en ligne y et colonne x
            Attention : on code chaque nombre sur 2 chiffres (d'où la longueur
6)
    """
    chaine = []
    for combo in combos:
        x = combo[0][0]
        if x < 10: chaine.append("0")
        chaine.append(str(x))
        y = combo[0][1]
        if y < 10: chaine.append("0")
        chaine.append(str(y))
        i = combo[1].id
        if i < 10: chaine.append("0")
        chaine.append(str(i))

    return "".join(chaine)

def from_string_to_combos(genes: str) -> list:
    """
    Fonction réciproque de `from_combo_to_string` sans les ticks
    """
    combos = []
    for i in range(len(genes)//6):
        combos.append(
            (
                (int(genes[6*i: 6*i+2]), int(genes[6*i+2: 6*i+4])),
                Figure.figures[int(genes[6*i+4: 6*i+6])],
                -1
            )
        )

    return combos

def is_success(population):
    """
        Vrai si le meilleur score obtenu est dans l'epsilon interval
        défini par la constante EPS et le meilleur score théorique
        pour un niveau d'expérience donné. Faux sinon
    """
    return evaluate(population)[0].fitness >\
        MAX_SCORE(population[0].athlete.xp) - EPS

def save(self, probs, population_number, infos):
    """
    Sauvegarde en Json les données de la population à chaque itération
    en plus des informations sur l'athlète original
    """
    # Formatage des données
    dataSerialized = []
    CROSSOVER_PROB, MUTATION_PROB = probs
    for i in range(len(self.populationOverTime)):
        for j in range(min(NUMBER_OF_CHROMOSOME_TO_KEEP, population_number-1)):

            dataSerialized.append({
                "g": self.populationOverTime[i][j].genes,
                "f": self.populationOverTime[i][j].fitness,
                "a": self.populationOverTime[i][j].age,
                "s": self.populationOverTime[i][j].size
            })

    athleteSerialized = {
        "xp": self.population[0].athlete.xp,
        "InitialPosition": INITIAL_POSITION,
    }

    metaInfoSerialized = {
        "is_success" : is_success(self.populationOverTime[-1]),
        "crossover_prob": CROSSOVER_PROB,
        "mutation_prob": MUTATION_PROB,
        "population_size": population_number,
        "terminaison_age": infos["terminaison_age"]
    }

    fieldCases = []
    for i in range(len(self.population[0].athlete.field.grille)):
        ligne = []
        for j in range(len(self.population[0].athlete.field.grille[i])):
            caseId = self.population[0].athlete.field.grille[i][j].id
            ligne.append(caseId)

        fieldCases.append(ligne)

    fieldSerialized = {
        "cases": fieldCases,
        "width": len(self.population[0].athlete.field.grille),
        "height": len(self.population[0].athlete.field.grille[0])
    }

    data = {
        "metaInfo": metaInfoSerialized,
        "athlete": athleteSerialized,
        "field": fieldSerialized,
        "dataGenerations": dataSerialized
    }

    os.makedirs(self.dirname, exist_ok=True)

    i=infos["start_filenumber"]
    while os.path.exists("{}/{}.json".format(self.dirname, i)):
        i += 1

    self.filename = str(i)

    with open("{}/{}.json".format(self.dirname, self.filename), "w") as f:
        json.dump(data, f)

    logging.debug("Data saved in {}.json".format(self.filename))

if __name__ == "__main__":
    ### Tests
    seed(0)

    # Vérification que les fonctions de traduction Genes <-> Combo
    # est bijective et ne change pas le score final

    # Initialisation d'athlètes
    population_number = 8
    population = [AthleteChromosome(Athlete(8))
            for _ in range(population_number)]

    Game.resetGames()

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

    # Genes avec un score 27 normalement
    genes = [[[7, 30], 2, -1], [[8, 31], 7, -1], [[7, 32], 1, -1], [[8, 33],
7, -1], [[8, 34], 2, -1], [[8, 35], 7, -1], [[8, 36], 5, -1], [[8, 35], 7, -
1], [[9, 36], 5, -1], [[9, 37], 17, -1], [[8, 37], 2, -1], [[8, 36], 16, -
1], [[7, 37], 9, -1], [[8, 38], 5, -1], [[9, 39], 5, -1], [[8, 39], 17, -1],
[[7, 39], 1, -1], [[6, 38], 1, -1], [[5, 39], 10, -1], [[6, 38], 10, -1],
[[6, 37], 8, -1], [[5, 38], 6, -1], [[4, 37], 2, -1], [[5, 36], 13, -1],
[[6, 36], 8, -1]]

    # Transformation du genes sauvegardé en genes utilisable par le programme
    # (Conversion des identifiants en Figure par exemple)
    genes_2 = []
    for coords, fig, tick in genes:
        genes_2.append(((coords[0], coords[1]), Figure.getFigureById(fig),
tick))
```

```
    s = from_combo_to_string(genes_2)
    g = from_string_to_combos(s)

    print("Echange string <-> combo bijectif (sans ticks) ?
"+str(g==genes_2))

    # Vérification que les deux évaluations des genes ont le même score
    population[4].athlete.combos = genes_2

    a = AthleteChromosome(population[4].athlete)
    a.calc_fitness()
    print("A-t-on égalité après deux évalutations consécutives des mêmes
gènes ? %s" %
          (a.fitness==a.calc_fitness()))

    print()

    # Test du croisement
    print("Croisement de 073002-083107-073201 et 083107-012601-070002")

    a1 = AthleteChromosome(population[4].athlete)
    a2 = AthleteChromosome(population[4].athlete)

    a1.genes = "073002083107073201"
    a2.genes = "083107012601070002"

    a1.athlete.combos = from_string_to_combos(a1.genes)
    a2.athlete.combos = from_string_to_combos(a2.genes)

    l = crossover([a1, a2], (1, 1))

    # Tests
    t1 = l[0].genes == "07300208310701260107000 2"
    t2 = l[1].genes == "083107073201"


    print("Donne-t-il 073002-083107-012601-070002 et 083107-073201 ? %s" %
          (t1 and t2))

    l = crossover(population, (0.2, 0.3))
    print("Garde-t-on la même taille de population ? %s" %
          (len(population) == len(l)))

    a1.genes = "012506"
    d = mutation([a1], 0)

    print("Mutation de 012506 : %s (Valide si égal à 022506)" %
d["population"][0].genes)
    print("Probs : (1, 1, 0, 0) devient (%s, %s, %s, %s)" % d["probs"])
```

```
'''


 Name : Elowan
 Creation : 30-08-2023 15:03:52
 Last modified : 21-05-2024 21:30:56
 File : customAthletes.py
'''
from Models import Athlete, FIGURES
from Terrain import Field
from main import AthleteChromosome

# Lilou Ruel
lilou = Athlete(8)
lilou.combos = [
    ((7, 31), FIGURES["double_cork"], 0),
    ((7, 31), FIGURES["jump"], 4),
    ((8, 33), FIGURES["180"], 5),
    ((6, 35), FIGURES["cast_backflip"], 7),
    ((7, 31), FIGURES["jump"], 10),
    ((6, 29), FIGURES["cast_backflip_360"], 11),
    ((5, 30), FIGURES["jump"], 13),
    ((2, 31), FIGURES["double_cork"], 15),
    ((1, 33), FIGURES["inward_flip"], 18),
    ((2, 28), FIGURES["180"], 22),
    ((4, 28), FIGURES["cork"], 23),
    ((7, 27), FIGURES["gaet_flip"], 26),
    ((8, 26), FIGURES["run"], 27),
    ((8, 24), FIGURES["run"], 28),
    ((8, 22), FIGURES["jump"], 29),
    ((8, 20), FIGURES["jump"], 30),
    ((6, 18), FIGURES["double_swing_gainer"], 31),
    ((5, 18), FIGURES["run"], 35),
    ((4, 16), FIGURES["180"], 38),
    ((3, 15), FIGURES["jump"], 40),
    ((3, 15), FIGURES["180"], 41),
    ((3, 15), FIGURES["jump"], 43),
    ((4, 15), FIGURES["run"], 44),
    ((5, 15), FIGURES["jump"], 45),
    ((6, 15), FIGURES["180"], 46),
    ((7, 15), FIGURES["jump"], 47),
    ((8, 15), FIGURES["cork"], 48),
]

if __name__ == "__main__":
    field = Field()
    field.createField()

    lilou.setField(field)
    lilouAthelte = AthleteChromosome(lilou)
    print(lilouAthelte.detailedFitness)
    print(lilouAthelte.fitness)
```

```
'''


 Name : Elowan
 Creation : 02-06-2023 11:00:02
 Last modified : 21-05-2024 21:31:18
 File : Models.py
'''
from random import choice
from utils import weighted_random

class Figure:
    instanceCount = 0
    figures = {}

    def __init__(self, name, duration, complexity):
        self.id = self.instanceCount
        self.name = name
        self.duration = duration
        self.complexity = complexity
        Figure.figures[self.id] = self
        Figure.instanceCount += 1

    def getFigureById(id):
        """Retourne la figure en fonction de son id, None sinon"""
        for figure in FIGURES.values():
            if figure.id == id:
                return figure
        return None

    def __str__(self) -> str:
        return self.name

    def __repr__(self) -> str:
        return "{}: Points accordés : {} pour une durée de {}".format(
            self.name, self.complexity, self.duration)

class Athlete:
    instanceCount = 0

    def __init__(self, xp):
        self.id = self.instanceCount
        self.xp = xp
        self.combos = []                        # ((x, y), Figure, tickStarted)
        self.position = (0, 0)                  # Coordonnées en (x, y)
        self.state = {                          # Etat de l'athlete
            "isMoving": False,
            "ticksSinceStartedMoving": 0,
            "movement": FIGURES["do_nothing"],  # Pas en mouvement
        }
        self.field = None
```

```
    Athlete.instanceCount += 1

def _getFigureByTick(self, tick):
    """Retourne le combo de l'athlete en fonction du tick de départ

    Params:
        tick (int): Le tick en question
    """
    for combo in self.combos:
        if combo[2] == tick:
            return combo

    return None

def takeAction(self, tick):
    """Fait faire une figure à l'athlete

    Params:
        tick (int): Le tick actuel
    """

    if self.state["movement"] != FIGURES["do_nothing"]:
        if self.state["ticksSinceStartedMoving"]+1 >= \
         self.state["movement"].duration:
            self._endMovement()

        else:
            self.state["ticksSinceStartedMoving"] += 1

    else:
        figure = self._getFigureByTick(tick)

        # Choisit où l'action doit être faite, si figure n'est
        # pas None, alors on va aux coordonnées de la figure, sinon
        # on bouge aléatoirement autour de l'athlete
        self._moveAround(figure)

        # Fait la figure si le figure du combo n'est pas None sinon
        # on fait une figure aléatoire
        self._startMovement(tick, figure = figure)

def _moveAround(self, figure=None):
    """Fait bouger l'athlete sur une case collée"""
    # Si combo n'est pas None, alors on va aux coordonnées du combo
    if figure != None:
        self.position = figure[0]
        return

    # Note les cases adjacentes de 0 à 8 (0 = haut gauche
    # et croissant dans le sens horaire) et
    # supprime celles ou l'athlete ne peut aller
    possibleNextPosition = \
        self._removeImpossibleNextCases([x for x in range(8)])

    # Choisit aléatoirement parmis ces cases possibles
    nextCase = choice(possibleNextPosition)

    # Met a jour les coordonnees
    self._setNewCoords(nextCase)


def _setNewCoords(self, nextCase):
    """
    En partant d'un nombre entre 0 et 7 inclus, on met a jour
    les nouvelles coordonées. On a 0 dans le coin haut gauche et
    c'est croissant dans le sens horaire (Ex case bas gauche = 6)
    """
    match nextCase:
        case 0:
            self.position = (
                self.position[0]-1,
                self.position[1]-1,
            )
        case 1:
            self.position = (
                self.position[0],
                self.position[1]-1,
            )
        case 2:
            self.position = (
                self.position[0]+1,
                self.position[1]-1,
            )
        case 3:
            self.position = (
                self.position[0]+1,
                self.position[1],
            )
        case 4:
            self.position = (
                self.position[0]+1,
                self.position[1]+1,
            )
        case 5:
            self.position = (
                self.position[0],
                self.position[1]+1,
            )
        case 6:
            self.position = (
                self.position[0]-1,
                self.position[1]+1,
            )
        case 7:
            self.position = (
                self.position[0]-1,
                self.position[1],
            )

def _removeImpossibleNextCases(self, cases):
    positionToRemove = []

    # Dernier x/y qui est encore sans le terrain
    lastCoordPossibleY = len(self.field.grille[0])-1
    lastCoordPossibleX = len(self.field.grille)-1

    if self.position[0] == 0:
        if 0 not in positionToRemove: positionToRemove.append(0)
        if 7 not in positionToRemove: positionToRemove.append(7)
        if 6 not in positionToRemove: positionToRemove.append(6)

    elif self.position[0] == lastCoordPossibleY:
        if 2 not in positionToRemove: positionToRemove.append(2)
        if 3 not in positionToRemove: positionToRemove.append(3)
        if 4 not in positionToRemove: positionToRemove.append(4)

    if self.position[1] == 0:
        if 0 not in positionToRemove: positionToRemove.append(0)
        if 1 not in positionToRemove: positionToRemove.append(1)
        if 2 not in positionToRemove: positionToRemove.append(2)

    elif self.position[1] == lastCoordPossibleX:
        if 4 not in positionToRemove: positionToRemove.append(4)
        if 5 not in positionToRemove: positionToRemove.append(5)
        if 6 not in positionToRemove: positionToRemove.append(6)

    # Retire toutes les cases impossibles
    for i in range(len(cases)-1, -1, -1):
        if i in positionToRemove:
            cases.pop(i)

    return cases


def _startMovement(self, tick, figure = None):
    """
    Regarde sur quelle case est l'athlete et commence la figure
    associée

    Params:
        tick (int): Tick actuel
    """
    # Si combo n'est pas None, alors on fait la figure du combo
```

```
    if figure != None:
      self.state["movement"] = figure[1]

    else:
      # Choisit aléatoirement le mouvement à faire parmis la liste
possible
      figures = self.field.getCase(self.position).figuresPossible
      self.state["movement"] = figures[weighted_random(
        0, len(figures)-1, 20, self.xp)]

      # Choisit une figure possible avec un tel niveau d'xp
      while self.state["movement"].complexity > (self.xp/2):
        self.state["movement"] = figures[weighted_random(
          0, len(figures)-1, 20, self.xp)]

      self.combos.append((self.position, self.state["movement"], tick))

    self.state["isMoving"] = True
    self.state["ticksSinceStartedMoving"] = 0


  def _endMovement(self):
    self.state["isMoving"] = False
    self.state["movement"] = FIGURES["do_nothing"]
    self.state["ticksSinceStartedMoving"] = 0

  def setField(self, field):
    self.field = field

  def __repr__(self) -> str:
    return "{} :\n    - xp : {}\n    - Combos : {}".format(
      self.id, self.xp, self.combos
    )

FIGURES = {
  "do_nothing": Figure("do_nothing", 1, 0),       # Ne rien faire pendant
1s

  "run": Figure("run", 1, 0),                     # Courir pendant 1s
  "jump": Figure("jump", 1, 0),                   # Sauter pendant 1s

  "180": Figure("180", 1, 0.5),                   # Faire un 180 pendant 1s
  "frontflip": Figure("frontflip", 3, 0.5),       # Faire un frontflip
pendant 3s
  "backflip": Figure("backflip", 2, 0.5),         # Faire un backflip
pendant 2s
  "gaet_flip": Figure("gaet_flip", 2, 0.5),       # Faire un gaet flip
(back
                                # en appui sur un coin de mur)
                                # pendant 2s
  "cork": Figure("cork", 3, 1),                   # Faire un cork pendant
3s
  "cast_backflip": Figure("cast_backflip", 1, 1), # Faire un cast backflip
(
                                # backflip en appui sur un
                                # mur) pendant 1s
  "gainer": Figure("gainer", 3, 1),               # Faire un gainer pendant
3s
  "inward_flip": Figure("inward_flip", 2, 1),     # Faire un inward flip (
                                # front qui te fait reculer)
                                # pendant 2s
  "540": Figure("540", 1, 1.5),                   # Faire un 540 pendant 1s
  "double_cork": Figure("double_cork", 4, 2),     # Faire un double cork
  "kong_gainer": Figure("kong_gainer", 2, 2),     # Faire un kong gainer
  "cast_backflip_360": Figure("cast_backflip_360",
                    2, 2.5),          # Faire un cast backflip 360
  "double_swing_gainer": Figure("double_swing_gainer", 2, 3), # Back sur
une barre

  "double_frontflip": Figure("double_frontflip",
                   2, 4),               # Faire un double front
  "double_backflip": Figure("double_backflip",
                   2, 4),               # Faire un double back

  "double_flip_360": Figure("double_flip_360", 3, 4.5), # Faire un double
flip 360
}

if __name__ == "__main__":
  athlete = Athlete(5, FIGURES["frontflip"])
  print(athlete)
```

```
'''

Name : Elowan
Creation : 23-06-2023 10:35:11
Last modified : 21-05-2024 21:31:31
File : traitement.py
'''

from json import dump, load
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as mticker
import matplotlib.patches as mpatches
from matplotlib.transforms import Bbox
import matplotlib as mpl
import os
import datetime
import logging
from tqdm import tqdm

from Models import Figure, FIGURES
from Terrain import Case
from Chromosome import from_string_to_combos

from consts import SIZE_X, SIZE_Y, NUMBER_OF_CHROMOSOME_TO_KEEP,\
  POPULATIONS, PROBS_M, PROBS_C, INITIAL_POSITION,\
  MAX_TICK_COUNT, ITERATION_NUMBER, TICK_INTERVAL

def unserializeJson(filename):
  """
  Take a json file and return a dict following this structure :
  {
    athlete : {
      xp,
    },
    field : {
      case: [[Case, Case], [Case, Case]],
      width
      height
      },
    dataGenerations : [
      {
        genes : "xxyyii",
        fitness,
        age,
        size,
      }
    ]
    meta : {
      is_success
    }
  }
  """
  with open(filename, "r") as file:
    data = load(file)

    parsed_data = {
      "athlete": {
        "xp": data["athlete"]["xp"],
      },
      "field": {
        "cases": [],
        "width": data["field"]["width"],
        "height": data["field"]["height"]
      },
      "meta": {
        "is_success": data["metaInfo"]["is_success"],
        "crossover_prob": data["metaInfo"]["crossover_prob"],
        "mutation_prob": data["metaInfo"]["mutation_prob"],
        "population_size": data["metaInfo"]["population_size"],
        "terminaison_age": data["metaInfo"]["terminaison_age"],
      },
      "dataGenerations": []
```

```
    }

    for lines in data["field"]["cases"]:
        parsed_line = []
        for case in lines:
            parsed_line.append(Case.getCaseById(case))

        parsed_data["field"]["cases"].append(parsed_line)


    for generation in data["dataGenerations"]:
        # Récupération de la fitness détaillée
        parsed_generation = {
            "genes": [],
            "fitness": generation["f"],
            "age": generation["a"],
            "size": generation["s"]
        }

        parsed_generation["genes"] = from_string_to_combos(generation["g"])
        parsed_data["dataGenerations"].append(parsed_generation)

    return parsed_data

def analyse(filename):
    """
    Prend en paramètre un dictionnaire généré par la fonction unserializeJson
    et renvoie un dictionnaire contenant les données analysées
    """
    logging.debug("Désérialisation du fichier {}...".format(filename))
    # Récupère les données
    data = unserializeJson(filename)

    logging.debug("Désérialisation terminée !\n")

    ### Histogramme des figures les plus utilisées
    logging.debug("Création de l'histogramme des figures les plus
utilisées...")
    # Comptage le nombre de fois que chaque figure est utilisée
    count = {}
    for generation in data["dataGenerations"]:
        for gene in generation["genes"]:
            if str(gene[1]) in count:
                count[str(gene[1])] += 1
            else:
                count[str(gene[1])] = 1

    # Ramène les valeurs sous forme de fréquence
    list_figures = []
    list_count = []
    for key, value in count.items():
        list_figures.append(key)
        list_count.append(value)

    for key in FIGURES.keys():
        if key not in list_figures:
            list_figures.append(key)
            list_count.append(0)

    list_figures = np.array(list_figures)
    list_count = np.array(list_count)
    nb_generations =
len(data["dataGenerations"])/NUMBER_OF_CHROMOSOME_TO_KEEP

    list_count = list_count/list_count.sum()

    # Tri par insertion des deux listes par ordre lexicographique croissant
    for i in range(1, len(list_figures)):
        j = i
        while j > 0 and list_figures[j-1] > list_figures[j]:
            # Swaping
            list_figures[j-1], list_figures[j] = list_figures[j],
list_figures[j-1]
            list_count[j-1], list_count[j] = list_count[j], list_count[j-1]
            j -= 1

    logging.debug("Histogramme des figures utilisées créé !\n")

    ### Evolution de la fitness au cours des générations
    logging.debug("Création de l'évolution de la fitness au cours des
générations...")

    list_fitness = []
    for generation in data["dataGenerations"]:
        list_fitness.append(generation["fitness"])
    list_fitness = np.array(list_fitness)


    # Fait une liste de la moyenne des fitness par génération
    list_fitness_moy = [sum(list_fitness[i:i+NUMBER_OF_CHROMOSOME_TO_KEEP])
                /NUMBER_OF_CHROMOSOME_TO_KEEP
                for i in range(0, len(list_fitness),
NUMBER_OF_CHROMOSOME_TO_KEEP)]

    list_fitness_moy = np.array(list_fitness_moy)
    logging.debug("Evolution de la fitness au cours des générations créée !\
n")

    ### Utilisation des cases au cours des générations
    logging.debug("Création de l'utilisation des cases au cours des
générations...")
    cases = [[0 for _ in range(data["field"]["height"])]
             for _ in range(data["field"]["width"])]

    # Comptage du nombre de fois que chaque case est utilisée
    for generation in data["dataGenerations"]:
        for gene in generation["genes"]:
            cases[gene[0][1]][gene[0][0]] += 1

    # Récupération la case la plus utilisée
    max_case = max(case for line in cases for case in line)

    # Ramène les valeurs à une fréquence d'utilisation
    for i in range(len(cases)):
        for j in range(len(cases[i])):
            cases[i][j] = cases[i][j]/max_case

    # Création de la matrice d'utilisation des cases
    freq_matrice = np.zeros((len(cases), len(cases[0])))
    for i in range(len(cases)):
        for j in range(len(cases[i])):
            freq_matrice[i][j] = cases[i][j]

    # Creation d'une matrice représentant le mobilier du terrain
    terrain_matrice = np.zeros((data["field"]["width"],
                        data["field"]["height"]))
    for i in range(data["field"]["width"]):
        for j in range(data["field"]["height"]):
            terrain_matrice[i][j] = data["field"]["cases"][i][j].id

    logging.debug("Utilisation des cases au cours des générations créée !\n")

    # Trouve le chemin utilisé par l'athlete avec la meilleur fitness
    best_athlete = {}
    best_fitness = -1
    for generation in data["dataGenerations"]:
        if generation["fitness"] > best_fitness:
            best_fitness = generation["fitness"]
            best_athlete = generation

    return {
        "freq_matrice": freq_matrice,
        "terrain_matrice": terrain_matrice,
        "fitness": list_fitness,
        "fitness_moy": list_fitness_moy,
        "figures": list_figures,
        "count": list_count,
        "nb_generations": nb_generations,
        "athlete": data["athlete"],
        "best_athlete": best_athlete,
        "nb_executions": 1,
        "is_success": data["meta"]["is_success"],
        "crossover_prob": data["meta"]["crossover_prob"],
        "mutation_prob": data["meta"]["mutation_prob"],
```

```
    "population_size": data["meta"]["population_size"],
    "terminaison_age": data["meta"]["terminaison_age"],
  }

def analyseFolder(foldername):
  """
  Analyse tous les fichiers d'un dossier en concaténant les données
  """
  # Récupération des noms des fichiers
  filenames = [f for f in os.listdir(foldername) if
os.path.isfile(os.path.join(foldername, f))]

  filenames.sort(key=lambda x : int(x.split(".")[0]))

  file_number = len(filenames)

  # Initialisation des données
  data = {
    "nb_generations": 0,
    "fitness": [],
    "fitness_moy": [],
    "freq_matrice": np.zeros((SIZE_Y, SIZE_X)),
    "terrain_matrice": np.zeros((SIZE_Y, SIZE_X)),
    "figures": [],
    "count": [],
    "best_athlete": {
      "fitness": 0,
      "genes": []
    },
    "athlete": {},
    "nb_executions": file_number,
    "performance": 0,
  }

  fitness_temp = []

  # Analyse de chaque fichier
  count = 0
  progress_bar = tqdm(total=len(filenames), desc="Analyse des fichiers",
unit="file")
  for filename in filenames:
    # Analyse du fichier
    file_data = analyse(os.path.join(foldername, filename))

    # Ajout des données
    data["nb_generations"] += file_data["nb_generations"]
    data["freq_matrice"] += file_data["freq_matrice"]
    fitness_temp.append(file_data["fitness"])
    data["performance"] += 1 if file_data["is_success"] else 0

    # Variables invariantes face aux excécutions de l'algorithme
    data["terrain_matrice"] = file_data["terrain_matrice"]
    data["figures"] = file_data["figures"]
    data["count"] = file_data["count"]
    data["athlete"] = file_data["athlete"]

    # Valeurs sans cohérence face aux exécutions
    data["population_size"] = -1
    data["crossover_prob"] = -1
    data["mutation_prob"] = -1
    data["terminaison_age"] = -1

    # Meilleur athlète
    if file_data["best_athlete"]["fitness"] > data["best_athlete"]
["fitness"]:
      data["best_athlete"] = file_data["best_athlete"]

    count += 1
    progress_bar.update()
    logging.debug("Analyse de {} terminée ({}%)".format(filename,
      round((count/file_number)*100, 2)))

  progress_bar.close()

  # Moyenne des données
  data["freq_matrice"] /= len(filenames)
  data["nb_generations"] /= len(filenames)
  data["performance"] /= len(filenames)

  # Moyenne de toutes les fitness par exécution
  # de l'algorithme génétique
  max_size = max(len(x) for x in fitness_temp)
  for i in range(max_size):
    moy_cur_fitness = [x[i] for x in fitness_temp if len(x) > i]
    data["fitness"].append(sum(moy_cur_fitness)/len(moy_cur_fitness))

  data["fitness"] = np.array(data["fitness"])

  # Moyenne par génération
  data["fitness_moy"] = [sum(data["fitness"]
[i:i+NUMBER_OF_CHROMOSOME_TO_KEEP])
            /NUMBER_OF_CHROMOSOME_TO_KEEP
            for i in range(0, len(data["fitness"]),
                    NUMBER_OF_CHROMOSOME_TO_KEEP)]

  data["fitness_moy"] = np.array(data["fitness_moy"])

  return data


def makeEvolFitnessImg(list_fitness, nb_executions=1):
  """
  Crée l'image de l'évolution de la fitness au cours des générations
  """
  plt.subplot(1, 2, 1)
  mean_fitness = list_fitness.mean()

  name = "Evolution du score au cours des générations ({} exécutions)"\
    .format(nb_executions)

  # Liste de la moyenne des fitness par génération
  fitness_moy_by_gen = [sum(list_fitness[i:i+NUMBER_OF_CHROMOSOME_TO_KEEP])
              /NUMBER_OF_CHROMOSOME_TO_KEEP
             for i in range(0, len(list_fitness),
NUMBER_OF_CHROMOSOME_TO_KEEP)]


  # Affichage de la courbe
  plt.plot(fitness_moy_by_gen, color="blue", label="Score", linewidth=2)

  plt.xlabel("Génération ({} athlètes/génération)"\
        .format(NUMBER_OF_CHROMOSOME_TO_KEEP))
  plt.ylabel("Score")
  plt.title(name)

  # Affichage de la fitness maximale
  plt.axhline(y=mean_fitness, color="red", linestyle="--",
        label="Moyenne : {}".format(round(float(mean_fitness), 2)),
        zorder = 3)

  plt.legend()

def makeCasesImg(freq_matrice, terrain_matrice, best_athlete, filename):
  """
  Crée l'image de l'utilisation des cases au cours des générations
  """

  # Création de la figure et des axes
  ax = plt.subplot(1, 2, 1, aspect='equal')

  cmap = mpl.colormaps['OrRd']

  # Création des rectangles avec les valeurs de la matrice frequence
  for i in range(len(freq_matrice)):
    for j in range(len(freq_matrice[i])):
      rect = mpatches.Rectangle((j, i), 1, 1, fc=cmap(freq_matrice[i, j]),
lw=2)
      ax.add_patch(rect)

  # Affichage du chemin de l'athlete avec la meilleur fitness avec
  # des chiffres croissants
  for i in range(len(best_athlete["genes"])):
    plt.text(best_athlete["genes"][i][0][0] + 0.5,
```

```
            best_athlete["genes"][i][0][1] + 0.5,
            str(i+1), color="black", ha="center", va="center")

    # Mise en forme de l'image
    max_x, max_y, diff = len(freq_matrice[0]), len(freq_matrice), 1.

    plt.title("Utilisation des cases au cours\ndes générations")
    plt.colorbar(mpl.cm.ScalarMappable(norm=mpl.colors.Normalize(vmin=0,
vmax=1), cmap=cmap), ax=ax)
    ax.set_xlim(0, max_x)
    ax.set_ylim(max_y, 0)
    ax.set_xticks(np.arange(max_x))
    ax.set_yticks(np.arange(max_y))
    ax.xaxis.tick_top()
    ax.grid()


    # Création de l'affichage représentant le terrain
    ax2 = plt.subplot(1, 2, 2, aspect='equal')

    # Couleurs représentant les différents types de cases
    colors = {
        0: "black",
        1: "grey",
        2: "white"
    }

# Création des rectangles avec les cases de la matrice terrain
for i in range(len(terrain_matrice)):
    for j in range(len(terrain_matrice[i])):
        rect = mpatches.Rectangle((j, i), 1, 1,
                                  fc=colors[int(terrain_matrice[i, j])],
                                  lw=2)
        ax2.add_patch(rect)

# Création de la légende
empty_patch = mpatches.Patch(color='black', label='Case sol')
wall_patch = mpatches.Patch(color='grey', label='Case mur')
hole_patch = mpatches.Patch(color='white', label='Case barre')

plt.legend(handles=[empty_patch, wall_patch, hole_patch],
           loc='center left', bbox_to_anchor=(1, 0.5))

plt.title("Mobilier du terrain")

ax2.set_xlim(0, max_x)
ax2.set_ylim(max_y, 0)
ax2.set_xticks(np.arange(max_x))
ax2.set_yticks(np.arange(max_y))
ax2.xaxis.tick_top()
ax2.grid()
```

```
    # Marges
    plt.subplots_adjust(right=1.2, bottom=-0.8)

    # Sauvegarde
    plt.savefig("traitement/{}_images/cases.png".format(filename),
                bbox_inches=Bbox([[0, -4], [9, 5]]),dpi=100)
    plt.close()

def makeFreqImg(list_figures, list_count, nb_executions=1):
    """
    Crée l'histogramme de la fréquence des figures
    """
    plt.subplot(1, 2, 2)
    title = "Fréquence des figures utilisées".format(nb_executions)

    # Affichage de l'histogramme
    plt.bar(list_figures, list_count)
    plt.xticks(rotation="vertical")
    plt.xlabel("Figures")
    plt.ylabel("Fréquence")
    plt.title(title)

    plt.autoscale(tight=False)

def constListImage(filename, const_dict):
    """
    Crée l'image de la liste des constantes utilisées pour les données
    """
    plt.axis('off')

    # Converti un dictionnaire vers un tableau 2D
    const_array = []
    for key, value in const_dict.items():
        const_array.append([str(key), str(value)])

    table = plt.table(cellText=const_array, colLabels=["Constante",
"Valeur"], loc='center')
    table.auto_set_font_size(False)
    table.set_fontsize(8)

    plt.tight_layout()
    plt.savefig("traitement/{}_images/constantes.png".format(filename),
dpi=100)
    plt.close()

def createStats(path=None, data=None):
    """
    Crée les images statistiques à partir d'un fichier ou de données
fournies.

    Args:
        path (str, optional): Le chemin du fichier à analyser.
            Si non spécifié, les données doivent être fournies.
        data (dict, optional): Les données à analyser.
            Si non spécifié, le fichier sera analysé en utilisant le chemin
spécifié.

    Returns:
        None: Cette fonction ne retourne aucune valeur.
    """
    if path is None and data is None:
        logging.error("Veuillez entrer un nom de fichier ou des données à
analyser.")
        return

    # Chronométrage du programme
    start_time = datetime.datetime.now()

    # Récupération des données
    if data is None:
        data = analyse(path)

    # Création du dossier de sauvegarde
    filename = path.split("data/")[-1] # Nom du fichier sans la partie
"data/"
    os.makedirs("traitement/{}_images".format(filename), exist_ok=True)

    logging.debug("Traitement des données...")

    # Création des images
    makeEvolFitnessImg(data["fitness"],data["nb_executions"])
    makeFreqImg(data["figures"], data["count"], data["nb_executions"])

    perf = " performance {}%".format(round(data["performance"]*100,2)) if
"performance" in data.keys() \
        else " succès ? {}".format(data["is_success"])

    name = "{}xp, {} générations/exécution en moy, {} individus/génération
({} exécutions)"\
        .format(
            data["athlete"]["xp"],
            round(data["nb_generations"]), data["population_size"],
            data["nb_executions"]
        )

    name += "\n" + perf

    plt.suptitle(name)
    plt.subplots_adjust(bottom=0.1, left=-0.2, right=1.3, top=0.85, hspace=2)

    plt.savefig("traitement/{}_images/freq&fitness.png".format(filename),
                bbox_inches=Bbox([[-2, -1.3], [9, 5]]),dpi=100)
    plt.close()
```

```
  constListImage(filename=filename, const_dict={
    "ATHLETE_XP": data["athlete"]["xp"],
    "CROSSOVER_PROB": data["crossover_prob"],
    "MUTATION_PROB": data["mutation_prob"],
    "POPULATION_SIZE": data["population_size"],
    "NUMBER_OF_CHROMOSOME_TO_KEEP":
         min(NUMBER_OF_CHROMOSOME_TO_KEEP, data["population_size"]-1),
    "TERMINAISON_AGE": data["terminaison_age"],
    "INITIAL_POSITION": INITIAL_POSITION,
    "MAX_TICK_COUNT": MAX_TICK_COUNT,
    "ITERATION_NUMBER": ITERATION_NUMBER,
    "SIZE_X": SIZE_X,
    "SIZE_Y": SIZE_Y,
    "TICK_INTERVAL": TICK_INTERVAL,
    "MAX_FITNESS_GOTTEN": data["best_athlete"]["fitness"],
  })

  makeCasesImg(data["freq_matrice"], data["terrain_matrice"],
          data["best_athlete"], filename)

  logging.debug("Traitement terminé (en {})!\n".format(
    datetime.datetime.now()-start_time))

def analyseStudy(foldername):
  """
  Analyse et création des images pour la comparaison avec l'étude
  """
  dataFolder = "data/"+foldername

  # Récupération des noms des fichiers
  filenames = [f for f in os.listdir(dataFolder)
          if os.path.isfile(os.path.join(dataFolder, f))]
  filenames.sort(key=lambda x : int(x.split(".")[0]))

  # Construction du dictionnaire :
  # perfs = {
  #   p_c-p_m-population_size : [0, 1, 1, 0]
  #       (0 pour un échec, 1 pour un succès)
  #       Longueur = Nombre d'executions
  # }
  #
  perfs = {}

  def pc_pmToString(file_data):
    pc = file_data["crossover_prob"]
    pm = file_data["mutation_prob"]
    popu = file_data["population_size"]
    return "{}-{}-{}".format(pc, pm, popu)

  # Analyse de chaque fichier
  count = 0
  progress_bar = tqdm(total=len(filenames), desc="Analyse des fichiers",
unit="file")
  for filename in filenames:
    # Analyse du fichier
    file_data = analyse(os.path.join(dataFolder, filename))

    category = pc_pmToString(file_data)
    if category not in perfs:
      perfs[category] = []

    perfs[category].append(1 if file_data["is_success"] else 0)

    count += 1
    progress_bar.update()
    logging.debug("Analyse de {} terminée ({}%)".format(filename,
      round((count/len(filenames))*100, 2)))

  progress_bar.close()

  # Construction du dictionnaire :
  # perfsFinales = {
  #     "pc-pm": [float]
  # }
  perfsFinales = {}

  logging.debug("Construction du graphique des succés...")
  for pc, pm in zip(PROBS_C, PROBS_M):
    perfsFinales["{}|{}".format(pc, pm)] = []
    for popu in POPULATIONS:
      perf = 0
      for success in perfs["{}-{}-{}".format(pc, pm, popu)]:
        perf += success
      perf /= ITERATION_NUMBER

      perfsFinales["{}|{}".format(pc, pm)].append(perf)

  # Affichage des différentes courbes
  for key, value in perfsFinales.items():
    pc, pm = key.split("|")

    plt.plot(POPULATIONS, value, label="p_c = {}; p_m = {}/l".format(pc,
pm))

  # Sauvegarde du tableau final
  saveDir = "traitement/study/{}".format(foldername)
  os.makedirs(saveDir, exist_ok=True)

  plt.legend(prop={'size': 10})
  plt.xlabel("Taille de la population")
  plt.ylabel("Performances")
  plt.ylim((0, 1))
  plt.xscale('log')
  ax = plt.gca()
  ax.xaxis.set_major_formatter(mticker.ScalarFormatter())
  ax.set_xticks([2, 10, 100, 500, 1000, 2000])
  plt.xlim((2, 2000))
  plt.savefig("{}/performances.png".format(saveDir), dpi=100)
  plt.close()

  logging.info("Fichier sauvegardé : {}/performances.png".format(
    saveDir
  ))


if __name__ == "__main__":
  # Afficher les logs dans un fichier
  logging.basicConfig(level=logging.DEBUG,
            format='%(asctime)s - %(levelname)s - %(message)s',
            datefmt='%d-%m-%Y %H:%M:%S',
            filename='logs/Traitement - {}.txt'.format(
                    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)

  # folder = "8xp/27-03-2024 11h32m49s/"
  # data = analyseFolder("data/" + folder)
  # createStats(path=folder+"/all", data=data)

  analyseStudy("8xp/26-04-2024 18h45m31s")
```

```
'''

Name : Elowan
Creation : 30-06-2023 23:57:04
Last modified : 21-05-2024 21:31:35
File : utils.py
'''
import random
import numpy as np

def weighted_random(mn, mx, mnweight, mxweight):
```

```
    """
    Exécute un random entre mn et mx avec une probabilité de mnweight
    d'avoir la plus basse valeur et mxweight d'avoir la plus haute valeur
    """
    return random.choices(range(mn, mx+1), \
     weights=np.linspace(mnweight,mxweight,(mx-mn)+1))[0]

def computeNextOccurrence(u: float, pm: float)->int:
    """
    Calcule la variable l de l'étude qui permet de réduire le
    nombre de calcul de variable aléatoire sans changer le succés
    de l'algorithme génétique.
    """
    if pm == 0.0: return 0
    else:
        val = (1/pm)*np.log(1-u)
        return int(val)
```

```
'''

 Name : Elowan
 Creation : 02-06-2023 11:01:13
 Last modified : 21-05-2024 21:31:26
 File : Terrain.py
'''
from Models import FIGURES
from consts import SIZE_X, SIZE_Y

class Case:
    instanceCount = 0

    def __init__(self, name, figuresPossible):
        self.id = self.instanceCount
        self.name = name
        self.figuresPossible = figuresPossible
        Case.instanceCount += 1

    def getCaseById(id):
        """Retourne la case en fonction de son id, None sinon"""
        for case in CASES.values():
            if case.id == id:
                return case

        return None

    def __repr__(self) -> str:
        return str(self.id)

    def __str__(self) -> str:
        return self.__repr__()


class Field:
    def __init__(self, grille = [[None for j in range(SIZE_X)]
                                  for i in range(SIZE_Y)]):
        self.grille = grille

    def createField(self):
        """Crée un terrain aléatoire"""
        field = sofiaField

        for i in range(SIZE_Y):
            for j in range(SIZE_X):
                self.grille[i][j] = Case.getCaseById(field[i][j])


    def getCase(self, positions) -> Case:
        """Retourne la case en coordonée x y"""
        x = positions[0]
        y = positions[1]
        return self.grille[y][x]

    def __len__(self) -> int:
        return len(self.grille)

    def __repr__(self) -> str:
        # Représente le terrain comme une grille
        result = ""
        for i in range(len(self.grille)):
            result += "| "
            for case in self.grille[i]:
                result += str(case) + " | "

            # Empeche le dernier saut a la ligne
            result += "\n"

        return result

    def __str__(self) -> str:
        return self.__repr__()

CASES = {
    "empty": Case("empty", [                    # Sol plat vide
        FIGURES["do_nothing"],
        FIGURES["run"],
        FIGURES["jump"],
        FIGURES["180"],
        FIGURES["backflip"],
        FIGURES["frontflip"],
        FIGURES["gaet_flip"],
        FIGURES["cork"],
        FIGURES["inward_flip"],
        FIGURES["540"],
        FIGURES["double_cork"],
        FIGURES["double_frontflip"],
        FIGURES["double_backflip"],
        FIGURES["double_flip_360"]
    ]),
    "wall":  Case("wall", [                         # Mur
        FIGURES["do_nothing"],
        FIGURES["jump"],
        FIGURES["run"],
        FIGURES["cast_backflip"],
        FIGURES["gainer"],
        FIGURES["kong_gainer"],
        FIGURES["cast_backflip_360"],
    ]),
    "bar":  Case("bar", [                           # Trou
        FIGURES["do_nothing"],
        FIGURES["jump"],
        FIGURES["run"],
        FIGURES["cast_backflip"],
        FIGURES["gainer"],
        FIGURES["cast_backflip_360"],
        FIGURES["double_swing_gainer"],
        FIGURES["double_backflip"],
    ]),
}

# Terrain Sofia (Bulgarie, voir https://www.youtube.com/watch?v=ubQ1w7awah8)
:
# 1 case = 1 m^2
sofiaField = [
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
```

```
    [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 2, 2, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 2, 2, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
    [0, 0, 2, 2, 1, 1, 2, 2, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
]


if __name__ == "__main__":
    # Grille 3x3
    field = Field()
    field.createField()
    print(len(field.grille), len(field.grille[0]))
    print(field.grille)
```