

```

1  '''
2  Name : Elowan
3  Creation : 23-06-2023 11:42:17
4  Last modified : 21-05-2024 12:02:40
5  File : Chromosome.py
6  '''
7  from random import randint, seed, choice
8  import logging
9  from math import sqrt, ceil
10 import json
11 import os
12
13 from Terrain import FIGURES
14 from Models import Athlete, Figure
15 from Game import Game
16 from Genetic import Chromosome
17 from consts import INITIAL_POSITION, NUMBER_OF_CHROMOSOME_TO_KEEP,\
18     EPS, MAX_SCORE, L, SIZE_X, SIZE_Y, DIST_MAX
19
20 k = 0
21 i = 0
22
23 class AthleteChromosome(Chromosome):
24     """
25     Classe représentant un athlète (une entité) pour
l'algorithme
26     génétique
27
28     Params:
29         athlete (Athlete): Athlète représenté par le chromosome
30     """
31     def __init__(self, athlete):
32         self.athlete = athlete
33         self.genes = from_combo_to_string(athlete.combos)
34         self.detailedFitness = {}
35
36         super().__init__(self.genes, self.calc_fitness(),
37             0, len(self.genes))
38
39     def calc_fitness(self) -> int:
40         """Calcule le score de l'athlète"""
41         score = {
42             "execution": {
43                 "safety": 3,
44                 "flow": 0,
45                 "mastery": 0,
46             },
47             "composition": {
48                 "use_of_space": 0,
49                 "use_of_obstacles": 0,
50                 "connection": 0,
51             },
52             "difficulty": {
53                 "variety": 0,
54                 "single_trick": 0,
55                 "whole_run": 0,
56             },
57         }
58         self.genes = from_combo_to_string(self.athlete.combos)
59
60         # Liste des figures faites
61         nb_figure = len(self.genes)//6

```

```

62         tricks = [Figure.figures[int(self.genes[6*i+4: 6*i+6])]
63             for i in range(nb_figure)]
64         field = self.athlete.field
65         try:
66             cases = [field.getCase(
67                 (int(self.genes[6*i:6*i+2]), int(self.genes[6*i+2: 6*i+4]))
68                 for i in range(nb_figure)]
69         except IndexError:
70             for i in range(nb_figure):
71                 print((int(self.genes[6*i:6*i+2]), int(self.genes[6*i+2:
6*i+4]])))
72
73
74
75         # Calcul de la sureté des figures
76         # On coefficiente la sureté par l'xp de l'athlète
77         score["execution"]["safety"] = \
78             (score["execution"]["safety"])*self.athlete.xp/10
79
80         # Calcul du flow
81         # Compte le nb de fois qu'on s'est arrêté
82         score["execution"]["flow"] = 3 - tricks.count(FIGURES["do_nothing"])
83
84         # Calcul de la maitrise
85         # Max 4
86         score["execution"]["mastery"] = 4*self.athlete.xp/10
87
88         # Calcul de l'utilisation de l'espace
89         # Compte le nb de cases différentes utilisées
90         # Post-it : Identifiant d'une case est unique donc on ne compte que
91         #         les cases visitées (dans notre liste de cases)
92         # Max 3 (comme le nb de cases)
93         l = []
94         for case in cases:
95             if case.id not in l: l.append(case.id)
96         score["composition"]["use_of_space"] = len(l)
97
98
99         # Calcul de l'utilisation des obstacles
100        # Compte le nb de types de cases différents utilisés
101        # Max 3 (comme le nb de cases)
102        l = []
103        for case in cases:
104            if case.name not in l: l.append(case.name)
105        score["composition"]["use_of_obstacles"] = len(l)
106
107        # Calcul de la connexion entre les obstacles
108        # Max 4
109        score["composition"]["connection"] = 4*self.athlete.xp/10
110
111        # Calcul de la variété
112        # Ajoute 1 pts a chaque figure de complexité >= 2
113        # (Donc que le trick n'est pas ds la catégorie de parkour classique)
114        score["difficulty"]["variety"] = sum([1
115            for trick in tricks
116            if trick.complexity >= 2])
117
118        if score["difficulty"]["variety"] > 3:
119            score["difficulty"]["variety"] = 3
120
121        # Calcul de la difficulté d'un trick
122        # Ajoute 1 pt par trick de complexité >= 3

```

```

123         score["difficulty"]["single_trick"] = sum([1
124             for trick in tricks
125             if trick.complexity >= 3])
126
127         if score["difficulty"]["single_trick"] > 3:
128             score["difficulty"]["single_trick"] = 3
129
130         # Calcul de la difficulté d'un run
131         score["difficulty"]["whole_run"] = 4*self.athlete.xp/10
132
133         # Calcul du score final
134         self.detailedFitness = score
135         self.fitness = sum([sum(score["execution"].values()),
136             sum(score["composition"].values()),
137             sum(score["difficulty"].values())])
138
139         if self.fitness < 0:
140             self.fitness = 0
141
142         # print(self.fitness)
143         return self.fitness
144
145     def __repr__(self) -> str:
146         return "AthleteID {} de score {} d'age {} et de taille {} : \
n{}}"\
147
148         .format(self.athlete.id, round(self.fitness, 2),
149             self.age, self.size, self.athlete)
150
151     def evaluate(population:list) -> list:
152         """
153         Notation de chaque athlète de la population
154
155         Params:
156             population (AthleteChromosome list): liste d'athlètes
157
158         Returns:
159             population (AthleteChromosome list): liste d'athlètes triés
160             (décroissant) par score
161         """
162         population.sort(key=lambda x: x.calc_fitness(), reverse=True)
163         return population
164
165     def selection(population:list) -> list:
166         """
167         Selectionne les parents de la prochaine population
168         Parents = 10 premiers en score de la population actuelle
169
170         Params:
171             population (AthleteChromosome list): liste d'athlètes
172
173         Returns:
174             (AthleteChromosome list): liste d'athlètes sélectionnés
175         """
176         return population[:10]
177
178     def get_point_communs(a1, a2) -> tuple[int, int]:
179         """
180         Renvoie les indices où les deux athlètes sont au même point dans
leur course,
181         différent de (0, 0).
182         Si renvoie (-1, -1) alors il n'y en a pas

```

```

183
184 Params :
185     a1 (AthleteChromosome) : Athlète
186     a2 (AthleteChromosome) : Athlète
187
188 Returns:
189     int: indice
190     ""
191     for i in range(0, len(a1.genes), 6):
192         for j in range(0, len(a1.genes), 6):
193             if a1.genes[i: i+6] == a2.genes[j: j+6]: return (i, j)
194     return (-1, -1)
195
196 def copy_chromosome(parent):
197     ""
198     Duplique littéralement un chromosome en augmentant son age
199
200 Params :
201     parent (AthleteChromosome) : Parent
202
203 Returns:
204     AthleteChromosome: Duplica
205     ""
206     child = Athlete(parent.athlete.xp)
207     child.combos = parent.athlete.combos
208     child.setField(parent.athlete.field)
209
210     childChro = AthleteChromosome(child)
211     childChro.age = parent.age + 1
212     return childChro
213
214 def new_children_crossover(p1, p2, cross_prob):
215     ""
216     Renvoie deux nouveaux chromosomes enfants des deux parents p1 et
217     p2
218     selon la méthode de croisement et la probabilité de croisement
219     cross_prob
220
221 Params :
222     p1 (AthleteChromosome) : Parent
223     p2 (AthleteChromosome) : Parent
224
225 Returns:
226     (AthleteChromosome, AthleteChromosome): Les enfants
227     ""
228     c1, c2 = get_point_communs(p1, p2)
229
230     if c1 != -1 and c2 != -1\
231         and randint(0, 100)/100 < cross_prob:
232
233         # Premier enfant, avec un premier croisement des combos
234         child1 = Athlete(p1.athlete.xp)
235         child1.setField(p1.athlete.field)
236         child1.combos =
237         from_string_to_combos(p1.genes[:c1]+p2.genes[c2:])
238         childChro1 = AthleteChromosome(child1)
239
240     # Deuxieme enfant, avec le croisement complémentaire au
241     premier
242     child2 = Athlete(p2.athlete.xp)
243     child2.setField(p2.athlete.field)
244     child2.combos = from_string_to_combos(p2.genes[:c2] +
245     p1.genes[c1:])

```

```

242     return childChro1, childChro2
243
244 else:
245     return copy_chromosome(p1), copy_chromosome(p2)
246
247 def crossover(parents: list, probs) -> list:
248     ""
249     Crée les enfants de la prochaine population
250     On choisit 2 parents et on les on prend 2 moins communs aux deux
251     chemins (s'il y a) et on échange les chemins entre ces deux points
252
253     Si les deux parents sont en réalité le même, on le copie tel quel.
254
255 Params:
256     parents (AthleteChromosome list): liste d'athlètes
257
258 Returns:
259     children (AthleteChromosome list): liste d'athlètes enfants
260     ""
261     children = []
262     CROSSOVER_PROB, _ = probs
263     for i in range(0, len(parents)-1, 2):
264         c1, c2 = new_children_crossover(parents[i], parents[i+1],
265                                         CROSSOVER_PROB)
266         children.append(c1)
267         children.append(c2)
268
269     return children
270
271 def dist(x1, y1, x2, y2):
272     ""Calculé la distance entre 2 points dans le plan""
273     return sqrt((x2-x1)**2 + (y2-y1)**2)
274
275 def coherence_suite_etats(e1, e2, e3):
276     ""
277     Vérifie la cohérence des l'état e2 provenant de l'état e1 et allant
278     à l'état e3
279
280 Params:
281     e1/e2/e3 (str): String de 6 caractères représentant un état
282
283 Returns:
284     (bool): Valide ou non
285     ""
286     x1 = int(e1[0:2])
287     x2 = int(e2[0:2])
288     x3 = int(e3[0:2])
289
290     y1 = int(e1[2:4])
291     y2 = int(e2[2:4])
292     y3 = int(e3[2:4])
293
294     return dist(x1, y1, x2, y2) <= DIST_MAX and dist(x2, y2, x3, y3) <=
295     DIST_MAX
296
297 def mutation_individual(athleteChromosome: AthleteChromosome, k:int):
298     ""
299     Mutation en place de l'athlete `athleteChromosome` passé en paramètre.
300     Le caractère du gene à modifier est imposé par le paramètre `k` contenu
301     entre 0 et 419 inclus.
302     ""
303     # k = Indice du caractère à modifier
304     # i = Indice du gene contenant la variable

```

```

305     i = (k - k%6)//6
306
307     # Etat associé au gène
308     e = athleteChromosome.genes[i*6: (i+1)*6]
309
310     # Positions et figure associé à l'état
311     x = int(e[0:2])
312     y = int(e[2:4])
313     f = int(e[4:6])
314
315     modifieur = choice([-1, 1])
316
317     # Récupération des états précédant et succédant l'état à l'étude
318     if i == 0 :
319         e1 = e
320     else:
321         e1 = athleteChromosome.genes[(i-1)*6: i*6]
322
323     if i >= len(athleteChromosome.genes)//6 - 1:
324         e3 = e
325     else:
326         e3 = athleteChromosome.genes[(i+1)*6: (i+2)*6]
327
328     has_mutated = True
329
330     # Match sur la composante qui va être modifiée
331     match (k%6)//2:
332         case 0 : # Si on modifie la variable de l'abscisse
333             # Le modifieur étant choisi avant le match, on vérifie que
334             # la modification apporté à l'abscisse n'enfreint aucune
335             # des conditions de bons fonctionnements comme :
336             # 0 <= x < SIZE_X et que le déplacement à cette case
337             depuis
338             # la case précédente e1 est possible et le déplacement
339             vers e3,
340             # assuré par le renvoie "true" de la fonction
341             coherence_suite_etats
342             e2 = from_combo_to_string(
343                 [((x+modifieur, y), Figure.getFigureById(f), 0))]
344
345             if x + modifieur >= 0 and x + modifieur < SIZE_X:
346                 if coherence_suite_etats(e1, e2, e3):
347                     x += modifieur
348
349             else:
350                 if x-modifieur >= 0 :
351                     e2_recovery = from_combo_to_string(
352                         [((x-modifieur, y),
353                         Figure.getFigureById(f), 0))]
354
355             if coherence_suite_etats(e1, e2_recovery, e3):
356                 x -= modifieur
357
358             else: has_mutated = False
359
360         else:
361             if x-modifieur >= 0 and x-modifieur < SIZE_X:
362                 e2_recovery = from_combo_to_string(
363                     [((x-modifieur, y), Figure.getFigureById(f),
364                     0))]
365
366             if coherence_suite_etats(e1, e2_recovery, e3):
367                 x -= modifieur

```

```

364         else: has_mutated = False
365
366     case 1: # Sensiblement la même chose que précédemment mais
pour l'ordonné
367         e2 = from_combo_to_string(
368             [(x, y+modifieur), Figure.getFigureById(f), 0])
369
370         if y + modifieur >= 0 and y + modifieur < SIZE_Y:
371
372             if coherence_suite_etats(e1, e2, e3):
373                 y += modifieur
374
375             else:
376                 if y-modifieur >= 0 :
377                     e2_recovery = from_combo_to_string(
378                         [(x, y-modifieur),
379 Figure.getFigureById(f), 0])
380
381                 if coherence_suite_etats(e1, e2_recovery,
e3):
382                     y -= modifieur
383                 else: has_mutated = False
384
385             else:
386                 if y-modifieur >= 0 and y-modifieur < SIZE_Y:
387                     e2_recovery = from_combo_to_string(
388                         [(x, y-modifieur), Figure.getFigureById(f),
389 0])
390
391                 if coherence_suite_etats(e1, e2_recovery, e3):
392                     y -= modifieur
393                 else: has_mutated = False
394             else: has_mutated = False
395
396     case 2: # Cas du changement de la figure
397         if f + modifieur >= len(FIGURES) or f+modifieur < 0 :
398             f -= modifieur
399         else:
400             f += modifieur
401
402     # Reconstruction du gène
403     gene = athleteChromosome.genes[0: i*6] +\
404         from_combo_to_string([(x, y), Figure.getFigureById(f),
405 0]))+\
406         athleteChromosome.genes[(i+1)*6:]
407
408     # Modification en place de l'athlete
409     athleteChromosome.genes = gene
410     athleteChromosome.athlete.combos = from_string_to_combos(gene)
411
412     return has_mutated
413 def mutation(population:list, l: int) -> list:
414     """
415     Fait muter la `population`, en ajoutant 1 ou -1 à un gène
aléatoire
416     (position x, position y ou l'indentifiant de la figure) selon le
dernier muté
417     et du paramètre `l` (Mutation Clock operation) et la cohérence de
ce changement avec le modèle réel
418

```

```

419
420     Params:
421         population (AthleteChromosome list): liste d'athlètes
422         l (int): nombre associé à une probabilité selon la 2nd étude sur les
GAs
423
424     Returns:
425         population (AthleteChromosome list): liste d'athlètes enfants
426
427     """
428     global k, i
429     has_mutated = mutation_individual(population[i], k)
430
431     if not has_mutated:
432         i = (i+1)%len(population)
433
434     else:
435         k = int((k+1)%L)
436         i = (i + ceil((k+1)/L))%len(population)
437
438     return population
439
440 def termination(population:list, infos) -> bool:
441     """
442     Condition d'arrêt de l'algorithme génétique
443
444     Params:
445         population (AthleteChromosome list): liste des athlètes
446
447     Returns:
448         (bool): True si l'algorithme doit s'arrêter, False sinon
449     """
450     return infos["generationCount"] > infos["terminaison_age"] or \
451         MAX_SCORE(population[0].athlete.xp) - EPS <
infos["maxPopulationFitness"]
452
453 def getBestAthlete(population):
454     """
455     Affiche le meilleur athlète de la population
456
457     Params:
458         population (AthleteChromosome list): liste des athlètes
459     """
460     evalPop = evaluate(population)
461     logging.info(evalPop[0])
462
463 def from_combo_to_string(combos) -> str:
464     """
465     Changement de représentation de la suite de combos en une chaîne de
caractères pour la représentation des gènes d'un chromosome
466
467     Params:
468         combos (tuple list): liste des combos sous la forme
469         [(x, y), Figure, tickStarted])
470
471     Returns:
472         str: concaténation de chaque figure codée sur 6 caractères. Par
exemple
473
474         "xxyyii" pour la figure d'identifiant i en ligne y et colonne x
475         Attention : on code chaque nombre sur 2 chiffres (d'où la
longueur 6)
476     """

```

```

477     chaine = []
478     for combo in combos:
479         x = combo[0][0]
480         if x < 10: chaine.append("0")
481         chaine.append(str(x))
482         y = combo[0][1]
483         if y < 10: chaine.append("0")
484         chaine.append(str(y))
485         i = combo[1].id
486         if i < 10: chaine.append("0")
487         chaine.append(str(i))
488
489     return "".join(chaine)
490
491 def from_string_to_combos(genes: str) -> list:
492     """
493     Fonction réciproque de `from_combo_to_string` sans les ticks
494     """
495     combos = []
496     for i in range(len(genes)//6):
497         combos.append(
498             (
499                 (int(genes[6*i: 6*i+2]), int(genes[6*i+2: 6*i+4])),
500                 Figure.figures[int(genes[6*i+4: 6*i+6])],
501                 -1
502             )
503         )
504
505     return combos
506
507 def is_success(population):
508     """
509     Vrai si le meilleur score obtenu est dans l'epsilon interval
510     défini par la constante EPS et le meilleur score théorique
511     pour un niveau d'expérience donné. Faux sinon
512     """
513     return evaluate(population)[0].fitness > \
514         MAX_SCORE(population[0].athlete.xp) - EPS
515
516 def save(self, probs, population_number, infos):
517     """
518     Sauvegarde en Json les données de la population à chaque itération
519     en plus des informations sur l'athlète original
520     """
521     # Formatage des données
522     dataSerialized = []
523     CROSSOVER_PROB, MUTATION_PROB = probs
524     for i in range(len(self.populationOverTime)):
525         for j in range(min(NUMBER_OF_CHROMOSOME_TO_KEEP,
population_number-1)):
526
527             dataSerialized.append({
528                 "g": self.populationOverTime[i][j].genes,
529                 "f": self.populationOverTime[i][j].fitness,
530                 "a": self.populationOverTime[i][j].age,
531                 "s": self.populationOverTime[i][j].size
532             })
533
534     athleteSerialized = {
535         "xp": self.population[0].athlete.xp,
536         "InitialPosition": INITIAL_POSITION,
537     }

```

```

538
539 metaInfoSerialized = {
540     "is_success" : is_success(self.populationOverTime[-1]),
541     "crossover_prob": CROSSOVER_PROB,
542     "mutation_prob": MUTATION_PROB,
543     "population_size": population_number,
544     "terminaison_age": infos["terminaison_age"]
545 }
546
547 fieldCases = []
548 for i in range(len(self.population[0].athlete.field.grille)):
549     ligne = []
550     for j in
range(len(self.population[0].athlete.field.grille[i])):
551         caseId = self.population[0].athlete.field.grille[i][j].id
552         ligne.append(caseId)
553
554     fieldCases.append(ligne)
555
556 fieldSerialized = {
557     "cases": fieldCases,
558     "width": len(self.population[0].athlete.field.grille),
559     "height": len(self.population[0].athlete.field.grille[0])
560 }
561
562 data = {
563     "metaInfo": metaInfoSerialized,
564     "athlete": athleteSerialized,
565     "field": fieldSerialized,
566     "dataGenerations": dataSerialized
567 }
568
569 os.makedirs(self.dirname, exist_ok=True)
570
571 i=infos["start_filename"]
572 while os.path.exists("{}/{}.json".format(self.dirname, i)):
573     i += 1
574
575 self.filename = str(i)
576
577 with open("{}/{}.json".format(self.dirname, self.filename), "w")
as f:
578     json.dump(data, f)
579
580 logging.debug("Data saved in {}.json".format(self.filename))
581
582 if __name__ == "__main__":
583     ### Tests
584     seed(0)
585
586     # Vérification que les fonctions de traduction Genes <-> Combo
587     # est bijective et ne change pas le score final
588
589     # Initialisation d'athlètes
590     population_number = 8
591     population = [AthleteChromosome(Athlete(8))
592                   for _ in range(population_number)]
593
594     Game.resetGames()
595
596     for athleteChromosome in population:
597         game = Game(athleteChromosome.athlete)

```

```

598         game.play()
599
600         # Genes avec un score 27 normalement
601         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]]
602
603         # Transformation du genes sauvegardé en genes utilisable par le programme
604         # (Conversion des identifiants en Figure par exemple)
605         genes_2 = []
606         for coords, fig, tick in genes:
607             genes_2.append(((coords[0], coords[1]), Figure.getFigureById(fig),
tick))
608
609         s = from_combo_to_string(genes_2)
610         g = from_string_to_combos(s)
611
612         print("Echange string <-> combo bijectif (sans ticks) ?
"+str(g==genes_2))
613
614         # Vérification que les deux évaluations des genes ont le même score
615         population[4].athlete.combos = genes_2
616
617         a = AthleteChromosome(population[4].athlete)
618         a.calc_fitness()
619         print("A-t-on égalité après deux évaluations consécutives des mêmes
gènes ? %s" %
620               (a.fitness==a.calc_fitness()))
621
622         print()
623
624         # Test du croisement
625         print("Croisement de 073002-083107-073201 et 083107-012601-070002")
626
627         a1 = AthleteChromosome(population[4].athlete)
628         a2 = AthleteChromosome(population[4].athlete)
629
630         a1.genes = "073002083107073201"
631         a2.genes = "083107012601070002"
632
633         a1.athlete.combos = from_string_to_combos(a1.genes)
634         a2.athlete.combos = from_string_to_combos(a2.genes)
635
636         l = crossover([a1, a2], (1, 1))
637
638         # Tests
639         t1 = l[0].genes == "073002083107012601070002"
640         t2 = l[1].genes == "083107073201"
641
642
643         print("Donne-t-il 073002-083107-012601-070002 et 083107-073201 ? %s" %
644               (t1 and t2))
645
646         l = crossover(population, (0.2, 0.3))
647         print("Garde-t-on la même taille de population ? %s" %
648               (len(population) == len(l)))
649
650         a1.genes = "012506"
651         d = mutation([a1], 0)

```

```

652
653         print("Mutation de 012506 : %s (Valide si égal à 022506)" %
d["population"][0].genes)
654         print("Probs : (1, 1, 0, 0) devient (%s, %s, %s, %s)" %
d["probs"])
655
656         _____
657         '''
658         Name : Elowan
659         Creation : 30-06-2023 23:56:45
660         Last modified : 21-05-2024 12:02:45
661         File : consts.py
662         '''
663         NUMBER_OF_CHROMOSOME_TO_KEEP = 20 # Nombre de chromosomes à garder à
664                                             # chaque génération
665
666         def MAX_SCORE(xp): # Score théorique maximal pour un
niveau
667             return 15 + 15*xp/10 # d'xp donné
668
669         EPS = 0.5 # Epsilon interval autour du score
max atteignable
670
671         INITIAL_POSITION = (7, 31) # Position initiale de l'athlète
672         MAX_TICK_COUNT = 70 # Nombre de tours(=secondes)
maximum
673
674         ITERATION_NUMBER = 1 # Nombre d'itérations de
l'algorithme
675
676         TICK_INTERVAL = 1 # Interval entre 2 executions de
la partie
677
678         CROSSOVER_PROB = 1 # Probabilité de croiser deux
parents
679         MUTATION_PROB = 0.05 # Probabilité de mutation d'un
enfant
680
681         SIZE_X = 10 # Taille du terrain
682         SIZE_Y = 40
683
684         # Valeurs utilisées dans l'étude
685         POPULATIONS = [2, 5, 10, 20, 35, 60, 100, 200, 300,
686                       , 700, 1000, 1200, 1400, 1800, 2000]
687
688         # Distance en mètre maximal qu'un être humain peut parcourir en
courant pendant 1s
689         DIST_MAX = 6
690
691         # L le nombre de variables représentant un gène
692         L = 6*70
693
694         PROBS_C = [0.0, 0.0, 0.0, 0.9, 0.9] # Probabilité de croisement
695         PROBS_M = [0.1, 0.5, 1.0, 0.0, 0.1] # Probabilité de mutation
696
697         NB_EVAL_MAX = 45_000 # = S
698         _____
699         '''
700         Name : Elowan
701         Creation : 30-08-2023 15:03:52
702         Last modified : 21-05-2024 21:30:56
703         File : customAthletes.py

```



```

704 '''
705 from Models import Athlete, FIGURES
706 from Terrain import Field
707 from main import AthleteChromosome
708
709 # Lilou Ruel
710 lilou = Athlete(8)
711 lilou.combos = [
712     ((7, 31), FIGURES["double_cork"], 0),
713     ((7, 31), FIGURES["jump"], 4),
714     ((8, 33), FIGURES["180"], 5),
715     ((6, 35), FIGURES["cast_backflip"], 7),
716     ((7, 31), FIGURES["jump"], 10),
717     ((6, 29), FIGURES["cast_backflip_360"], 11),
718     ((5, 30), FIGURES["jump"], 13),
719     ((2, 31), FIGURES["double_cork"], 15),
720     ((1, 33), FIGURES["inward_flip"], 18),
721     ((2, 28), FIGURES["180"], 22),
722     ((4, 28), FIGURES["cork"], 23),
723     ((7, 27), FIGURES["gaet_flip"], 26),
724     ((8, 26), FIGURES["run"], 27),
725     ((8, 24), FIGURES["run"], 28),
726     ((8, 22), FIGURES["jump"], 29),
727     ((8, 20), FIGURES["jump"], 30),
728     ((6, 18), FIGURES["double_swing_gainer"], 31),
729     ((5, 18), FIGURES["run"], 35),
730     ((4, 16), FIGURES["180"], 38),
731     ((3, 15), FIGURES["jump"], 40),
732     ((3, 15), FIGURES["180"], 41),
733     ((3, 15), FIGURES["jump"], 43),
734     ((4, 15), FIGURES["run"], 44),
735     ((5, 15), FIGURES["jump"], 45),
736     ((6, 15), FIGURES["180"], 46),
737     ((7, 15), FIGURES["jump"], 47),
738     ((8, 15), FIGURES["cork"], 48),
739 ]
740
741 if __name__ == "__main__":
742     field = Field()
743     field.createField()
744
745     lilou.setField(field)
746     lilouAthehte = AthleteChromosome(lilou)
747     print(lilouAthehte.detailedFitness)
748     print(lilouAthehte.fitness)
749
750 '''
751 Name : Elowan
752 Creation : 02-06-2023 11:00:05
753 Last modified : 21-05-2024 21:31:06
754 File : Game.py
755 '''
756
757 from Terrain import Field
758 from Models import Athlete, FIGURES
759
760 from consts import INITIAL_POSITION, MAX_TICK_COUNT, TICK_INTERVAL
761
762 class Game:
763     """
764     Classe représentant un round de la compétition
765     """

```

```

766
767 instances = []
768 def __init__(self, athlete):
769     self.field = Field()
770     self.field.createField()
771     self.athlete = athlete
772     self.state = 0 # Etat de la partie
773     self.tickCount = 0 # 1 tick = 1 seconde
774
775     self.athlete.setField(self.field)
776
777     Game.instances.append(self)
778
779 def start(self):
780     """Initialisation des valeurs de depart de la competition"""
781     self.tickCount = 0
782     self.state = 1
783     self.athlete.position = INITIAL_POSITION
784
785 def update(self):
786     """Met à jour l'état de l'athlète et retourne l'état de la
787 compétition"""
788     if self.tickCount >= MAX_TICK_COUNT:
789         self.end()
790         return self.state
791
792     self.athlete.takeAction(self.tickCount)
793
794     self.tickCount += TICK_INTERVAL
795     return self.state
796
797 def end(self):
798     """Fonction appelée lorsque la competition termine"""
799     # # Si la suite de combos ne rempli pas entièrement le nombre de
800 combos
801     # # disponible, on duplique le dernier combo (en supprimant la figure
802 puis le tick)
803     # if len(self.athlete.combos) < 70:
804
805     #     n = 70 - len(self.athlete.combos)
806
807     #     pos, _, _ = self.athlete.combos[-1]
808
809     #     for _ in range(n):
810         self.athlete.combos.append((pos, FIGURES["run"], 0))
811
812 def play(self, iterate=lambda x: None, callback=lambda x: None):
813     """Fait faire une partie entière au jeu
814 Params:
815     iterate (function) - Prend en paramètre l'instance du jeu et ne
816 renvoie rien. Elle est executee a chaque
817 tick de la partie
818
819     callback (function) - Prend en paramètre l'instance du jeu et ne
820 renvoie rien. Elle est executee a la fin
821 de
822 la partie
823
824 """
825     self.start()
826     while self.update() == 1:

```

```

827         iterate(self)
828
829         callback(self)
830
831 def _getGameByAthlete(athlete):
832     for i in Game.instances:
833         if i.athlete.id == athlete.id:
834             return i
835
836 def resetGames():
837     """Supprime toutes les instances de Game"""
838     Game.instances = []
839
840 if __name__ == "__main__":
841     athlete = Athlete(5, FIGURES["backflip"])
842     game = Game(athlete)
843     def iterate(game):
844         print("Athlete state (in the second {}) :
845 ".format(game.tickCount))
846         print(" - Position ({}, {})".format(
847             game.athlete.position[0], game.athlete.position[1]
848         ))
849         print(" - Case : {}".format(
850             game.field.getCase(game.athlete.position).name
851         ))
852         print(" - Current movement : {} since {} seconds".format(
853             game.athlete.state["movement"],
854             game.athlete.state["ticksSinceStartedMoving"]
855         ))
856         print()
857
858     def callback(game):
859         print("Game state : {}\nFor {} ticks".format(game.state,
860 game.tickCount))
861         print("Combos : {}".format(athlete.combos))
862
863     print("Game started !")
864     game.play(iterate=iterate, callback=callback)
865
866 '''
867 Name : Elowan
868 Creation : 08-06-2023 10:00:40
869 Last modified : 21-05-2024 21:31:11
870 File : Genetic.py
871 '''
872 import random
873
874 class Chromosome:
875     """
876     Classe abstraite représentant un chromosome (une entitée)
877     de l'algorithme génétique
878
879 Params:
880     genes (Polymorphique): Variable représentant les
881 caractéristiques
882     du Chromosome
883     fitness (int): Score attribué du chromosome
884     age (int): Nombre de générations du chromosome
885     size (int): Taille du chromosome
886
887 """
888     def __init__(self, genes, fitness, age, size):
889         self.genes = genes

```

```

882     self.fitness = fitness
883     self.age = age
884     self.size = size
885
886     def __repr__(self) -> str:
887         return "Fitness : {}".format(self.fitness)
888
889 class GeneticAlgorithm:
890     """
891     Algorithme génétique adapté à un problème donné
892
893     Params:
894         population (Chromosome): liste de chromosomes
895         termination (function): fonction qui renvoie true/false
896             selon le critère de terminaison de l'algorithme
897         evaluate (function): fonction qui évalue la population
898         selection (function): fonction qui sélectionne les
parents
899         crossover (function): fonction qui crée les enfants
900         mutation (function): fonction qui fait des mutations sur
eux
901     """
902     def __init__(self, population:list, termination, evaluate,
903                 selection, crossover, mutation, save, dirname="") ->
None:
904
905         # Renvoie true/false selon le critere de terminaison
906         self.termination = termination
907
908         # Fonction d'algo genetique
909         self.evaluate = evaluate           # Tri la population
910         self.selection = selection         # Selectionne les parents
911         self.crossover = crossover         # Crée les enfants
912         self.mutation = mutation          # Fait des mutations sur eux
913         self.save = save
914
915         self.population = population
916         self.population_len = len(population)
917
918         self.populationOverTime = [self.population]
919
920         self.dirname = dirname
921
922     def run(self, iteration=lambda x: None, callback=lambda x: None):
923         """
924         Execute l'algorithme génétique
925
926         Params:
927             ?iteration (function (GeneticAlgorithm): None):
fonction qui s'exécute à chaque itération
928             de la boucle principale (Prend en paramètre
l'instance de l'algorithme génétique et
929             renvoie None)
930             ?callback (function (GeneticAlgorithm): None):
fonction qui s'exécute à la fin de
931             l'algorithme (Prend en paramètre l'instance de
l'algorithme génétique et renvoie None)
932         """
933         self.population = self.evaluate(self.population)
934         while not self.termination(self.population):
935             iteration(self.population)
936             self.population = self.selection(self.population)

```

```

937         self.population = self.crossover(self.population)
938         self.population = self.mutation(self.population)
939         self.population = self.evaluate(self.population)
940
941         # Sauvegarde des données pour la sérialisation
942         self.populationOverTime.append(self.population)
943
944         self.save(self)
945         return callback(self.population)
946
947     def getFilename(self):
948         return self.filename
949
950     def getDirname(self):
951         return self.dirname
952
953 if __name__ == "__main__":
954     random.seed(22)
955
956     # Test de l'algorithme génétique avec le problème OneMax
957     class OneMaxChromosome(Chromosome):
958         def __init__(self, genes: list):
959             self.genes = genes
960             super().__init__(self.genes, self.calc_fitness(), 0,
len(self.genes))
961
962         def calc_fitness(self):
963             sum_gene = 0
964             for i in range(len(self.genes)):
965                 sum_gene += int(self.genes[i])
966
967             return sum_gene
968
969         def mix(self, gene1, gene2):
970             self.genes = gene1+gene2
971             self.fitness = self.calc_fitness()
972             self.age += 1
973
974         def __repr__(self):
975             string = ""
976             for i in range(len(self.genes)):
977                 string += str(self.genes[i])
978
979             return string
980
981     def sumlist(population):
982         return [chro.fitness for chro in population]
983
984     def swap(i, j, liste):
985         temp = liste[i]
986         liste[i] = liste[j]
987         liste[j] = temp
988
989         return liste
990
991     def evaluate(population):
992         # Evaluation de la population
993         sums = sumlist(population)
994
995         # Sort list decreasing
996         for i in range(len(sums)-1):
997             max = i

```

```

998             for j in range(i+1, len(sums)):
999                 if sums[j] >= sums[max]:
1000                     max = j
1001
1002             if max != i:
1003                 sums = swap(i, max, sums)
1004                 population = swap(i, max, population)
1005
1006         return population
1007
1008     def selection(population):
1009         # Selection des parents
1010         # On fait des pairs de chaque element
1011         liste = []
1012
1013         for i in range(0, len(population)-1, 2):
1014             liste.append((population[i], population[i+1]))
1015
1016         return liste
1017
1018     def crossover(population):
1019         # Creation des enfants
1020         cross_point = random.randint(0, 1000)
1021         liste = []
1022         for i in range(len(population)):
1023             chro1 = population[i][0]
1024             chro2 = population[i][1]
1025             chro1.mix(chro1.genes[:cross_point],
chro2.genes[cross_point:])
1026             chro2.mix(chro1.genes[:cross_point],
chro2.genes[cross_point:])
1027
1028             liste.append(chro1)
1029             liste.append(chro2)
1030
1031         # print("".join([ str(x) for x in liste[0] ]))
1032         return liste
1033
1034     def mutation(population):
1035         liste = []
1036         for pop in population:
1037             l = pop
1038             if random.randint(0, 100) < 5:
1039                 random.shuffle(l.genes)
1040
1041             liste.append(l)
1042
1043         return liste
1044
1045     def termination(population):
1046         sums = sumlist(population)
1047         best = sums.index(max(sums))
1048         return population[best].fitness >= 1000
1049
1050     def iteration(population):
1051         sums = sumlist(population)
1052         best = sums.index(max(sums))
1053
1054         print("Current best : {}".format(population[best].fitness))
1055
1056     def save(population): pass
1057

```

```

1058     populationChromosome = [ OneMaxChromosome([ random.randint(0, 1)
1059                                     for x in
1060     range(1000) ])
1061                                     for y in range(100) ]
1062     OneMaxProblem = GeneticAlgorithm(populationChromosome,
1063     termination, evaluate,
1064     selection, crossover, mutation, save)
1065     OneMaxProblem.run(iteration)
1066     sums = sumlist(OneMaxProblem.population)
1067     best = sums.index(max(sums))
1068     print("Best overall : {}\nAge :
1069     {}".format(OneMaxProblem.population[best],
1070     OneMaxProblem.population[best].age))
1071
1072     _____
1073     '''
1074     Name : Elowan
1075     Creation : 02-06-2023 10:59:30
1076     Last modified : 18-05-2024 12:32:24
1077     File : main.py
1078     '''
1079     import datetime
1080     import logging
1081     from tqdm import tqdm
1082     from multiprocessing import Pool, Lock, Manager
1083     import sys
1084
1085     from Chromosome import *
1086     from Models import Athlete
1087     from Game import Game
1088     from Genetic import GeneticAlgorithm
1089     from utils import computeNextOccurrence
1090     from traitement import analyseStudy, analyseFolder, createStats
1091     from consts import NB_EVAL_MAX, PROBS_C, PROBS_M,\
1092     ITERATION_NUMBER, NUMBER_OF_CHROMOSOME_TO_KEEP,\
1093     INITIAL_POSITION, MAX_TICK_COUNT, SIZE_X, SIZE_Y,\
1094     POPULATIONS
1095
1096     # Variables communes à tous les processus pour connaître combien de
1097     ligne
1098     # il faut sauter pour afficher chaque barre de progression
1099     # Notamment utile pour empecher des sauts de lignes inopinés
1100     position_Lock = Lock()
1101     positionsBars = []
1102
1103     def playAllGames(population:list):
1104         """
1105         Joue toutes les parties associées aux athlètes de la population
1106
1107         Params:
1108             population (AthleteChromosome list): liste des athlètes à
1109             faire jouer
1110             """
1111         # Supprime les anciens jeux
1112         Game.resetGames()
1113
1114         for athleteChromosome in population:
1115             game = Game(athleteChromosome.athlete)

```

```

1114         game.play()
1115
1116     def logConstants(athleteLevel, seed):
1117         """
1118         Log les constantes de l'algorithmme
1119         """
1120
1121         logging.debug("Seed : {}".format(seed))
1122         logging.debug("Iteration number : {}".format(ITERATION_NUMBER))
1123         logging.debug("Athlete level : {}".format(athleteLevel))
1124         logging.debug("Number of chromosomes to keep :
1125         {}".format(NUMBER_OF_CHROMOSOME_TO_KEEP))
1126         logging.debug("Initial position : {}".format(INITIAL_POSITION))
1127         logging.debug("Size of the field : {}".format((SIZE_X, SIZE_Y)))
1128         logging.debug("Max tick count : {}".format(MAX_TICK_COUNT))
1129
1130     def replaceBars(i):
1131         """
1132         Descend les barres de progression d'une ligne dès qu'une des barres
1133         termine.
1134         """
1135         positionLock.acquire()
1136
1137         positionsBars[i] = 0
1138         for j in range(i+1, len(positionsBars)):
1139             positionsBars[j] -= 1
1140
1141         positionLock.release()
1142
1143     def process(args):
1144         """
1145         Fonction exécutant l'algorithmme génétique pour une population de
1146         `population_number` individus et avec toutes les probabilités définies
1147         par le fichier `const.py`.
1148
1149         Params:
1150             - args (tuple) : Contient `population_number` ainsi que `iteration`
1151             représentant le i-ième appel à process
1152
1153         """
1154         population_number, iteration = args
1155         count = 0
1156         total = len(PROBS_C)*ITERATION_NUMBER
1157
1158         text = "Tests des probs sur une population de {0:04}
1159         individus".format(population_number)
1160
1161         pbar = tqdm(total=len(PROBS_C)*ITERATION_NUMBER, unit="exec",
1162             desc=text, file=sys.stdout, position=positionsBars[i])
1163
1164         for probs in zip(PROBS_C, PROBS_M):
1165             for _ in range(ITERATION_NUMBER):
1166                 logging.debug("##### ITERATION {}/{} #####".format(count, total))
1167                 logging.debug("Population number : {}".format(population_number))
1168                 logging.debug("Probabilités : Crossover = {}% Mutation = {}%\
1169                 ".format(probs[0]*100, probs[1]*100))
1170                 logging.debug("Terminaison age :
1171                 {}".format(NB_EVAL_MAX/population_number))
1172
1173             ### Creation de la population
1174             # Chronométrage
1175             start_time = datetime.datetime.now()
1176

```

```

1177         # population_number de fois le meme athlete
1178         population = [AthleteChromosome(
1179             Athlete(athleteLevel))
1180             for _ in range(population_number)]
1181
1182         playAllGames(population)
1183
1184     ### Algorithme génétique
1185
1186     # Informations utilisées pour déterminer la terminaison
1187     # de l'algorithmme (quand le maximum n'a pas été modifié
1188     depuis
1189
1190     # un certain temps maxAge par exemple)
1191     infos = {
1192         "maxPopulationFitness": 0,
1193         "maxAge": 0,
1194         "generationCount": 0,
1195         "terminaison_age": NB_EVAL_MAX/population_number,
1196         "start_filename": iteration*total,
1197     }
1198
1199     # Crée la variable l comme dans l'étude sélectionnée
1200     u = randint(0, 99)/100
1201     l = computeNextOccurrence(u, probs[1])
1202
1203     # Ajout de paramètres supplémentaires
1204     def term(pop): return termination(pop, infos)
1205     def s(pop): return save(pop, probs, population_number,
1206     infos)
1207     def mut(pop): return mutation(pop, l)
1208     def cross(pop):
1209         children = crossover(pop, probs)
1210
1211         # Duplications des enfants pour generer une population
1212         entière
1213         popu = []
1214         for _ in
1215         range(population_number//len(children)):popu.extend(children)
1216         return popu[:population_number]
1217
1218     def iterate(population):
1219         evalPop = evaluate(population)
1220         infos["generationCount"] += 1
1221
1222         # Mise a jour du score max des athlètes
1223         # et le temps depuis quand c'est le max
1224         if evalPop[0].fitness > infos["maxPopulationFitness"]:
1225             infos["maxPopulationFitness"] = evalPop[0].fitness
1226             infos["maxAge"] = 1
1227
1228         else:
1229             infos["maxAge"] += 1
1230
1231     parkourGenetic = GeneticAlgorithm(population, term,
1232     evaluate,
1233     selection, cross, mut, s,
1234     "data/{}".format(dirnameSaves))
1235
1236     try:

```

```

1228         parkourGenetic.run(iteration=iterate)
1229         logging.debug("\nMeilleur athlète de la dernière
génération: {}".format(evaluate(parkourGenetic.population)[0]))
1230         logging.debug("Temps d'execution :
{}".format(datetime.datetime.now() - start_time))
1231
1232         count+=1
1233         pbar.pos = positionsBars[iteration]
1234         pbar.update(1)
1235         pbar.refresh()
1236
1237     except Exception as e :
1238         logging.error("Erreur de l'appel avec population =
{}; Iteration = {}".format(population_number, iteration))
1239         logging.error(e)
1240         pbar.close()
1241         replaceBars(iteration)
1242         return
1243
1244     replaceBars(iteration)
1245
1246
1247
1248 if __name__ == "__main__":
1249     s = 1713449159 # Pour avoir des résultats reproductibles
1250     # s = int(datetime.datetime.now().timestamp())
1251     seed(s)
1252
1253     athleteLevel = 8
1254     dirNameSaves = "{}xp/{}".format(athleteLevel,
datetime.datetime.now()
1255                                     .strftime("%d-%m-%Y %Hh%Mm
%Ss"))
1256
1257
1258     dirs = "data/{}".format(dirNameSaves)
1259     os.makedirs('logs', exist_ok=True)
1260
1261     # Initialisation des logs
1262     logging.basicConfig(level=logging.DEBUG,
1263                         format='%(asctime)s - %(levelname)s - %
(message)s',
1264                         datefmt='%d-%m-%Y %H:%M:%S',
1265                         filename='logs/Main -
{}.txt'.format(str(athleteLevel) + "xp - "
1266                                     +
datetime.datetime.now()
1267                                     .strftime("%d-%m-%Y %H:
%M:%S")),
1268                         filemode='w')
1269
1270     # Affichage dans la console
1271     console = logging.StreamHandler()
1272     console.setLevel(logging.INFO)
1273     formatter = logging.Formatter('%(asctime)s - %(levelname)s - %
(message)s')
1274     console.setFormatter(formatter)
1275     logging.getLogger('').addHandler(console)
1276
1277     logConstants(athleteLevel, s)
1278
1279     # Multi-Processing pour accélérer le temps d'exécution
1280     init_time = datetime.datetime.now()

```

```

1281     logging.info("Exécutions des algorithmes avec différentes tailles de
population")
1282
1283     # Initialisation des positions des barres
1284     positionsBars = Manager().list([i for i in range(len(POPULATIONS))])
1285
1286     # Lancement des processus
1287     args = [(POPULATIONS[i], i) for i in range(len(POPULATIONS))]
1288     with Pool(initializer=tqdm.set_lock, initargs=(tqdm.get_lock(),)) as p :
1289         p.map(process, args)
1290
1291     logging.info("Fin des exécutions. Créations des graphiques")
1292
1293     # Analyse du dossier (moyenne sur toutes les itérations)
1294     data = analyseFolder(dirs)
1295     createStats(path="{}all".format(dirs), data=data)
1296
1297     # Dessine un graphe semblable à l'étude
1298     analyseStudy(dirNameSaves)
1299
1300     logging.info("Temps d'execution total : {}".format(
1301         (datetime.datetime.now() - init_time)))
1302
1303     '''
1304     Name : Elowan
1305     Creation : 02-06-2023 11:00:02
1306     Last modified : 21-05-2024 21:31:18
1307     File : Models.py
1308     '''
1309     from random import choice
1310     from utils import weighted_random
1311
1312     class Figure:
1313         instanceCount = 0
1314         figures = {}
1315
1316         def __init__(self, name, duration, complexity):
1317             self.id = self.instanceCount
1318             self.name = name
1319             self.duration = duration
1320             self.complexity = complexity
1321             Figure.figures[self.id] = self
1322             Figure.instanceCount += 1
1323
1324         def getFigureById(id):
1325             """Retourne la figure en fonction de son id, None sinon"""
1326             for figure in FIGURES.values():
1327                 if figure.id == id:
1328                     return figure
1329
1330             return None
1331
1332         def __str__(self) -> str:
1333             return self.name
1334
1335         def __repr__(self) -> str:
1336             return "{}: Points accordés : {} pour une durée de {}".format(
1337                 self.name, self.complexity, self.duration)
1338
1339     class Athlete:
1340         instanceCount = 0
1341

```

```

1342     def __init__(self, xp):
1343         self.id = self.instanceCount
1344         self.xp = xp
1345         self.combos = [] # ((x, y), Figure,
tickStarted)
1346         self.position = (0, 0) # Coordonnées en (x,
y)
1347         self.state = { # Etat de l'athlete
1348             "isMoving": False,
1349             "ticksSinceStartedMoving": 0,
1350             "movement": FIGURES["do_nothing"], # Pas en mouvement
1351         }
1352         self.field = None
1353
1354         Athlete.instanceCount += 1
1355
1356     def _getFigureByTick(self, tick):
1357         """Retourne le combo de l'athlete en fonction du tick de
départ
1358
1359         Params:
1360             tick (int): Le tick en question
1361         """
1362         for combo in self.combos:
1363             if combo[2] == tick:
1364                 return combo
1365
1366         return None
1367
1368     def takeAction(self, tick):
1369         """Fait faire une figure à l'athlete
1370
1371         Params:
1372             tick (int): Le tick actuel
1373         """
1374
1375         if self.state["movement"] != FIGURES["do_nothing"]:
1376             if self.state["ticksSinceStartedMoving"]+1 >= \
1377                 self.state["movement"].duration:
1378                 self._endMovement()
1379
1380             else:
1381                 self.state["ticksSinceStartedMoving"] += 1
1382
1383         else:
1384             figure = self._getFigureByTick(tick)
1385
1386             # Choisit où l'action doit être faite, si figure n'est
1387             # pas None, alors on va aux coordonnées de la figure,
1388             # sinon
1389             # on bouge aléatoirement autour de l'athlete
1390             self._moveAround(figure)
1391
1392             # Fait la figure si le figure du combo n'est pas None
1393             # sinon
1394             # on fait une figure aléatoire
1395             self._startMovement(tick, figure = figure)
1396
1397     def _moveAround(self, figure=None):
1398         """Fait bouger l'athlete sur une case collée"""
1399         # Si combo n'est pas None, alors on va aux coordonnées du
combo

```



```

1398     if figure != None:
1399         self.position = figure[0]
1400         return
1401
1402     # Note les cases adjacentes de 0 à 8 (0 = haut gauche
1403     # et croissant dans le sens horaire) et
1404     # supprime celles ou l'athlete ne peut aller
1405     possibleNextPosition = \
1406         self._removeImpossibleNextCases([x for x in range(8)])
1407
1408     # Choisit aléatoirement parmi ces cases possibles
1409     nextCase = choice(possibleNextPosition)
1410
1411     # Met a jour les coordonnees
1412     self._setNewCoords(nextCase)
1413
1414
1415 def _setNewCoords(self, nextCase):
1416     """
1417     En partant d'un nombre entre 0 et 7 inclus, on met a jour
1418     les nouvelles coordonnées. On a 0 dans le coin haut gauche et
1419     c'est croissant dans le sens horaire (Ex case bas gauche = 6)
1420     """
1421     match nextCase:
1422         case 0:
1423             self.position = (
1424                 self.position[0]-1,
1425                 self.position[1]-1,
1426             )
1427         case 1:
1428             self.position = (
1429                 self.position[0],
1430                 self.position[1]-1,
1431             )
1432         case 2:
1433             self.position = (
1434                 self.position[0]+1,
1435                 self.position[1]-1,
1436             )
1437         case 3:
1438             self.position = (
1439                 self.position[0]+1,
1440                 self.position[1],
1441             )
1442         case 4:
1443             self.position = (
1444                 self.position[0]+1,
1445                 self.position[1]+1,
1446             )
1447         case 5:
1448             self.position = (
1449                 self.position[0],
1450                 self.position[1]+1,
1451             )
1452         case 6:
1453             self.position = (
1454                 self.position[0]-1,
1455                 self.position[1]+1,
1456             )
1457         case 7:
1458             self.position = (
1459                 self.position[0]-1,

```

```

1460                 self.position[1],
1461             )
1462
1463     def _removeImpossibleNextCases(self, cases):
1464         positionToRemove = []
1465
1466         # Dernier x/y qui est encore sans le terrain
1467         lastCoordPossibleY = len(self.field.grille[0])-1
1468         lastCoordPossibleX = len(self.field.grille)-1
1469
1470         if self.position[0] == 0:
1471             if 0 not in positionToRemove: positionToRemove.append(0)
1472             if 7 not in positionToRemove: positionToRemove.append(7)
1473             if 6 not in positionToRemove: positionToRemove.append(6)
1474
1475         elif self.position[0] == lastCoordPossibleY:
1476             if 2 not in positionToRemove: positionToRemove.append(2)
1477             if 3 not in positionToRemove: positionToRemove.append(3)
1478             if 4 not in positionToRemove: positionToRemove.append(4)
1479
1480         if self.position[1] == 0:
1481             if 0 not in positionToRemove: positionToRemove.append(0)
1482             if 1 not in positionToRemove: positionToRemove.append(1)
1483             if 2 not in positionToRemove: positionToRemove.append(2)
1484
1485         elif self.position[1] == lastCoordPossibleX:
1486             if 4 not in positionToRemove: positionToRemove.append(4)
1487             if 5 not in positionToRemove: positionToRemove.append(5)
1488             if 6 not in positionToRemove: positionToRemove.append(6)
1489
1490         # Retire toutes les cases impossibles
1491         for i in range(len(cases)-1, -1, -1):
1492             if i in positionToRemove:
1493                 cases.pop(i)
1494
1495         return cases
1496
1497
1498     def _startMovement(self, tick, figure = None):
1499         """
1500         Regarde sur quelle case est l'athlete et commence la figure
1501         associée
1502
1503         Params:
1504             tick (int): Tick actuel
1505         """
1506         # Si combo n'est pas None, alors on fait la figure du combo
1507         if figure != None:
1508             self.state["movement"] = figure[1]
1509
1510         else:
1511             # Choisit aléatoirement le mouvement à faire parmi la liste
1512             possible
1513             figures = self.field.getCase(self.position).figuresPossible
1514             self.state["movement"] = figures[weighted_random(
1515                 0, len(figures)-1, 20, self.xp)]
1516
1517             # Choisit une figure possible avec un tel niveau d'xp
1518             while self.state["movement"].complexity > (self.xp/2):
1519                 self.state["movement"] = figures[weighted_random(
1520                     0, len(figures)-1, 20, self.xp)]
1521

```

```

1521             self.combos.append((self.position, self.state["movement"],
1522 tick))
1523
1524             self.state["isMoving"] = True
1525             self.state["ticksSinceStartedMoving"] = 0
1526
1527         def _endMovement(self):
1528             self.state["isMoving"] = False
1529             self.state["movement"] = FIGURES["do_nothing"]
1530             self.state["ticksSinceStartedMoving"] = 0
1531
1532         def setField(self, field):
1533             self.field = field
1534
1535         def __repr__(self) -> str:
1536             return "{} :\n    - xp : {}\n    - Combos : {}".format(
1537                 self.id, self.xp, self.combos
1538             )
1539
1540     FIGURES = {
1541         "do_nothing": Figure("do_nothing", 1, 0),          # Ne rien faire
1542         pendant 1s
1543         "run": Figure("run", 1, 0),                        # Courir pendant
1544         1s
1545         "jump": Figure("jump", 1, 0),                      # Sauter pendant
1546         1s
1547         "180": Figure("180", 1, 0.5),                      # Faire un 180
1548         pendant 1s
1549         "frontflip": Figure("frontflip", 3, 0.5),          # Faire un
1550         frontflip pendant 3s
1551         "backflip": Figure("backflip", 2, 0.5),            # Faire un
1552         backflip pendant 2s
1553         "gaet_flip": Figure("gaet_flip", 2, 0.5),          # Faire un gaet
1554         flip (back
1555         # en appui sur un
1556         coin de mur)
1557         # pendant 2s
1558         "cork": Figure("cork", 3, 1),                      # Faire un cork
1559         pendant 3s
1560         "cast_backflip": Figure("cast_backflip", 1, 1),    # Faire un cast
1561         backflip (
1562         # backflip en
1563         appui sur un
1564         # mur) pendant 1s
1565         "gainer": Figure("gainer", 3, 1),                  # Faire un gainer
1566         pendant 3s
1567         "inward_flip": Figure("inward_flip", 2, 1),        # Faire un inward
1568         flip (
1569         # front qui te
1570         fait reculer)
1571         # pendant 2s
1572         "540": Figure("540", 1, 1.5),                      # Faire un 540
1573         pendant 1s
1574         "double_cork": Figure("double_cork", 4, 2),        # Faire un double
1575         cork
1576         "kong_gainer": Figure("kong_gainer", 2, 2),        # Faire un kong
1577         gainer
1578         "cast_backflip_360": Figure("cast_backflip_360",

```

```

1565         2, 2.5),          # Faire un cast backflip 360
1566     "double_swing_gainer": Figure("double_swing_gainer", 2, 3), #
Back sur une barre
1567
1568     "double_frontflip": Figure("double_frontflip",
1569         2, 4),          # Faire un double front
1570     "double_backflip": Figure("double_backflip",
1571         2, 4),          # Faire un double back
1572
1573     "double_flip_360": Figure("double_flip_360", 3, 4.5), # Faire un
double flip 360
1574 }
1575
1576 if __name__ == "__main__":
1577     athlete = Athlete(5, FIGURES["frontflip"])
1578     print(athlete)
1579
1580 '''
1581 Name : Elowan
1582 Creation : 02-06-2023 11:01:13
1583 Last modified : 21-05-2024 21:31:26
1584 File : Terrain.py
1585 '''
1586 from Models import FIGURES
1587 from consts import SIZE_X, SIZE_Y
1588
1589 class Case:
1590     instanceCount = 0
1591
1592     def __init__(self, name, figuresPossible):
1593         self.id = self.instanceCount
1594         self.name = name
1595         self.figuresPossible = figuresPossible
1596         Case.instanceCount += 1
1597
1598     def getCaseById(id):
1599         """Retourne la case en fonction de son id, None sinon"""
1600         for case in CASES.values():
1601             if case.id == id:
1602                 return case
1603
1604         return None
1605
1606     def __repr__(self) -> str:
1607         return str(self.id)
1608
1609     def __str__(self) -> str:
1610         return self.__repr__()
1611
1612 class Field:
1613     def __init__(self, grille = [[None for j in range(SIZE_X)]
1614                                   for i in range(SIZE_Y)]):
1615         self.grille = grille
1616
1617     def createField(self):
1618         """Crée un terrain aléatoire"""
1619         # for i in range(SIZE_Y):
1620         #     for j in range(SIZE_X):
1621         #         self.grille[i][j] = choice(list(CASES.values()))
1622
1623         # Terrain fixe :

```

```

1624         # field = [[2, 2, 2, 1, 0, 1, 1, 0, 1, 0], [1, 0, 2, 0, 1, 0, 1, 0,
1, 1], [2, 1, 2, 1, 2, 1, 0, 1, 1, 2], [1, 2, 0, 0, 2, 0, 0, 1, 2], [2, 2, 2, 2,
2, 1, 1, 0, 0, 1], [0, 0, 1, 1, 0, 1, 1, 1, 1, 0], [1, 2, 2, 1, 1, 0, 0, 1, 2, 0],
[0, 2, 1, 0, 2, 0, 1, 1, 0, 0], [1, 2, 2, 0, 0, 2, 1, 2, 0, 1], [2, 0, 0, 0, 0, 0, 1,
0, 1, 1, 0], [2, 0, 2, 0, 2, 0, 0, 2, 1, 2], [0, 2, 2, 0, 1, 0, 0, 0, 1, 2], [2, 1,
1, 0, 1, 0, 2, 2, 2, 0], [0, 0, 0, 2, 1, 1, 1, 2, 1, 2], [0, 1, 2, 1, 0, 0, 1, 1,
2, 1]]
1625
1626         field = sofiaField
1627
1628         for i in range(SIZE_Y):
1629             for j in range(SIZE_X):
1630                 self.grille[i][j] = Case.getCaseById(field[i][j])
1631
1632
1633     def getCase(self, positions) -> Case:
1634         """Retourne la case en coordonnée x y"""
1635         x = positions[0]
1636         y = positions[1]
1637         return self.grille[y][x]
1638
1639     def __len__(self) -> int:
1640         return len(self.grille)
1641
1642     def __repr__(self) -> str:
1643         # Représente le terrain comme une grille
1644         result = ""
1645         for i in range(len(self.grille)):
1646             result += "| "
1647             for case in self.grille[i]:
1648                 result += str(case) + " | "
1649
1650         # Empeche le dernier saut a la ligne
1651         result += "\n"
1652
1653         return result
1654
1655     def __str__(self) -> str:
1656         return self.__repr__()
1657
1658 CASES = {
1659     "empty": Case("empty", [
1660         FIGURES["do_nothing"],
1661         FIGURES["run"],
1662         FIGURES["jump"],
1663         FIGURES["180"],
1664         FIGURES["backflip"],
1665         FIGURES["frontflip"],
1666         FIGURES["gaet_flip"],
1667         FIGURES["cork"],
1668         FIGURES["inward_flip"],
1669         FIGURES["540"],
1670         FIGURES["double_cork"],
1671         FIGURES["double_frontflip"],
1672         FIGURES["double_backflip"],
1673         FIGURES["double_flip_360"]
1674     ]),
1675     "wall": Case("wall", [
1676         FIGURES["do_nothing"],
1677         FIGURES["jump"],
1678         FIGURES["run"],
1679         FIGURES["cast_backflip"],

```

```

1680         FIGURES["gainer"],
1681         FIGURES["kong_gainer"],
1682         FIGURES["cast_backflip_360"],
1683     ]),
1684     "bar": Case("bar", [
1685         FIGURES["do_nothing"],
1686         FIGURES["jump"],
1687         FIGURES["run"],
1688         FIGURES["cast_backflip"],
1689         FIGURES["gainer"],
1690         FIGURES["cast_backflip_360"],
1691         FIGURES["double_swing_gainer"],
1692         FIGURES["double_backflip"],
1693     ]),
1694 }
1695
1696 # Terrain Sofia (Bulgarie, voir https://www.youtube.com/watch?v=ubQ1w7awah8) :
1697 # 1 case = 1 m^2
1698 sofiaField = [
1699     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1700     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1701     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1702     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1703     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1704     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1705     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1706     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1707     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1708     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1709     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1710     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1711     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1712     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1713     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1714     [0, 0, 1, 1, 0, 0, 1, 1, 0, 0],
1715     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1716     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1717     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1718     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1719     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1720     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1721     [0, 0, 2, 2, 2, 2, 2, 2, 0, 0],
1722     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1723     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1724     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1725     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1726     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1727     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1728     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1729     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1730     [0, 0, 1, 1, 2, 2, 1, 1, 0, 0],
1731     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1732     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1733     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1734     [0, 0, 1, 1, 2, 2, 1, 1, 0, 0],
1735     [0, 0, 1, 1, 1, 1, 1, 1, 0, 0],
1736     [0, 0, 2, 2, 1, 1, 2, 2, 0, 0],
1737     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1738     [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
1739 ]
1740

```

```

1741
1742 if __name__ == "__main__":
1743     # Grille 3x3
1744     field = Field()
1745     field.createField()
1746     print(len(field.grille), len(field.grille[0]))
1747     print(field.grille)
1748
1749 '''
1750 Name : Elowan
1751 Creation : 23-06-2023 10:35:11
1752 Last modified : 21-05-2024 21:31:31
1753 File : traitement.py
1754 '''
1755
1756 from json import dump, load
1757 import numpy as np
1758 import matplotlib.pyplot as plt
1759 import matplotlib.ticker as mticker
1760 import matplotlib.patches as mpatches
1761 from matplotlib.transforms import Bbox
1762 import matplotlib as mpl
1763 import os
1764 import datetime
1765 import logging
1766 from tqdm import tqdm
1767
1768 from Models import Figure, FIGURES
1769 from Terrain import Case
1770 from Chromosome import from_string_to_combos
1771
1772 from consts import SIZE_X, SIZE_Y, NUMBER_OF_CHROMOSOME_TO_KEEP,\
1773     POPULATIONS, PROBS_M, PROBS_C, INITIAL_POSITION,\
1774     MAX_TICK_COUNT, ITERATION_NUMBER, TICK_INTERVAL
1775
1776 def unserializeJson(filename):
1777     """
1778     Take a json file and return a dict following this structure :
1779     {
1780         athlete : {
1781             xp,
1782         },
1783         field : {
1784             case: [[Case, Case], [Case, Case]],
1785             width
1786             height
1787         },
1788         dataGenerations : [
1789             {
1790                 genes : "xxyyii",
1791                 fitness,
1792                 age,
1793                 size,
1794             }
1795         ]
1796         meta : {
1797             is_success
1798         }
1799     }
1800     """
1801     with open(filename, "r") as file:
1802         data = load(file)

```

```

1803
1804     parsed_data = {
1805         "athlete": {
1806             "xp": data["athlete"]["xp"],
1807         },
1808         "field": {
1809             "cases": [],
1810             "width": data["field"]["width"],
1811             "height": data["field"]["height"]
1812         },
1813         "meta": {
1814             "is_success": data["metaInfo"]["is_success"],
1815             "crossover_prob": data["metaInfo"]["crossover_prob"],
1816             "mutation_prob": data["metaInfo"]["mutation_prob"],
1817             "population_size": data["metaInfo"]["population_size"],
1818             "terminaison_age": data["metaInfo"]["terminaison_age"],
1819         },
1820         "dataGenerations": []
1821     }
1822
1823     for lines in data["field"]["cases"]:
1824         parsed_line = []
1825         for case in lines:
1826             parsed_line.append(Case.getCaseById(case))
1827
1828     parsed_data["field"]["cases"].append(parsed_line)
1829
1830
1831     for generation in data["dataGenerations"]:
1832         # Récupération de la fitness détaillée
1833         parsed_generation = {
1834             "genes": [],
1835             "fitness": generation["f"],
1836             "age": generation["a"],
1837             "size": generation["s"]
1838         }
1839
1840         parsed_generation["genes"] =
1841         from_string_to_combos(generation["g"])
1842         parsed_data["dataGenerations"].append(parsed_generation)
1843
1844     return parsed_data
1845
1846 def analyse(filename):
1847     """
1848     Prend en paramètre un dictionnaire généré par la fonction unserializeJson
1849     et renvoie un dictionnaire contenant les données analysées
1850     """
1851     logging.debug("Désérialisation du fichier {}...".format(filename))
1852     # Récupère les données
1853     data = unserializeJson(filename)
1854
1855     logging.debug("Désérialisation terminée !\n")
1856
1857     ### Histogramme des figures les plus utilisées
1858     logging.debug("Création de l'histogramme des figures les plus
1859     utilisées...")
1860     # Comptage le nombre de fois que chaque figure est utilisée
1861     count = {}
1862     for generation in data["dataGenerations"]:
1863         for gene in generation["genes"]:
1864             if str(gene[1]) in count:

```

```

1863             count[str(gene[1])] += 1
1864         else:
1865             count[str(gene[1])] = 1
1866
1867     # Ramène les valeurs sous forme de fréquence
1868     list_figures = []
1869     list_count = []
1870     for key, value in count.items():
1871         list_figures.append(key)
1872         list_count.append(value)
1873
1874     for key in FIGURES.keys():
1875         if key not in list_figures:
1876             list_figures.append(key)
1877             list_count.append(0)
1878
1879     list_figures = np.array(list_figures)
1880     list_count = np.array(list_count)
1881     nb_generations =
1882     len(data["dataGenerations"])/NUMBER_OF_CHROMOSOME_TO_KEEP
1883
1884     list_count = list_count/list_count.sum()
1885
1886     # Tri par insertion des deux listes par ordre lexicographique
1887     croissant
1888     for i in range(1, len(list_figures)):
1889         j = i
1890         while j > 0 and list_figures[j-1] > list_figures[j]:
1891             # Swaping
1892             list_figures[j-1], list_figures[j] = list_figures[j],
1893             list_figures[j-1]
1894             list_count[j-1], list_count[j] = list_count[j],
1895             list_count[j-1]
1896             j -= 1
1897
1898     logging.debug("Histogramme des figures utilisées créé !\n")
1899
1900     ### Evolution de la fitness au cours des générations
1901     logging.debug("Création de l'évolution de la fitness au cours des
1902     générations...")
1903
1904     list_fitness = []
1905     for generation in data["dataGenerations"]:
1906         list_fitness.append(generation["fitness"])
1907         list_fitness = np.array(list_fitness)
1908
1909     # Fait une liste de la moyenne des fitness par génération
1910     list_fitness_moy =
1911     [sum(list_fitness[i:i+NUMBER_OF_CHROMOSOME_TO_KEEP])
1912     /NUMBER_OF_CHROMOSOME_TO_KEEP
1913     for i in range(0, len(list_fitness),
1914     NUMBER_OF_CHROMOSOME_TO_KEEP)]
1915
1916     list_fitness_moy = np.array(list_fitness_moy)
1917     logging.debug("Evolution de la fitness au cours des générations
1918     créée !\n")
1919
1920     ### Utilisation des cases au cours des générations
1921     logging.debug("Création de l'utilisation des cases au cours des
1922     générations...")
1923     cases = [[0 for _ in range(data["field"]["height"])]

```

```

1916         for _ in range(data["field"]["width"])]
1917
1918     # Comptage du nombre de fois que chaque case est utilisée
1919     for generation in data["dataGenerations"]:
1920         for gene in generation["genes"]:
1921             cases[gene[0][1]][gene[0][0]] += 1
1922
1923     # Récupération la case la plus utilisée
1924     max_case = max(case for line in cases for case in line)
1925
1926     # Ramène les valeurs à une fréquence d'utilisation
1927     for i in range(len(cases)):
1928         for j in range(len(cases[i])):
1929             cases[i][j] = cases[i][j]/max_case
1930
1931     # Création de la matrice d'utilisation des cases
1932     freq_matrice = np.zeros((len(cases), len(cases[0])))
1933     for i in range(len(cases)):
1934         for j in range(len(cases[i])):
1935             freq_matrice[i][j] = cases[i][j]
1936
1937     # Creation d'une matrice représentant le mobilier du terrain
1938     terrain_matrice = np.zeros((data["field"]["width"],
1939                                data["field"]["height"])))
1940     for i in range(data["field"]["width"]):
1941         for j in range(data["field"]["height"]):
1942             terrain_matrice[i][j] = data["field"]["cases"][i][j].id
1943
1944     logging.debug("Utilisation des cases au cours des générations
1945 créée !\n")
1946
1947     # Trouve le chemin utilisé par l'athlete avec la meilleur fitness
1948     best_athlete = {}
1949     best_fitness = -1
1950     for generation in data["dataGenerations"]:
1951         if generation["fitness"] > best_fitness:
1952             best_fitness = generation["fitness"]
1953             best_athlete = generation
1954
1955     return {
1956         "freq_matrice": freq_matrice,
1957         "terrain_matrice": terrain_matrice,
1958         "fitness": list_fitness,
1959         "fitness_moy": list_fitness_moy,
1960         "figures": list_figures,
1961         "count": list_count,
1962         "nb_generations": nb_generations,
1963         "athlete": data["athlete"],
1964         "best_athlete": best_athlete,
1965         "nb_executions": 1,
1966         "is_success": data["meta"]["is_success"],
1967         "crossover_prob": data["meta"]["crossover_prob"],
1968         "mutation_prob": data["meta"]["mutation_prob"],
1969         "population_size": data["meta"]["population_size"],
1970         "terminaison_age": data["meta"]["terminaison_age"],
1971     }
1972
1973 def analyseFolder(foldername):
1974     """
1975     Analyse tous les fichiers d'un dossier en concaténant les données
1976     """
1977     # Récupération des noms des fichiers

```

```

1977     filenames = [f for f in os.listdir(foldername) if
1978 os.path.isfile(os.path.join(foldername, f))]
1979
1980     filenames.sort(key=lambda x : int(x.split(".")[0]))
1981
1982     file_number = len(filenames)
1983
1984     # Initialisation des données
1985     data = {
1986         "nb_generations": 0,
1987         "fitness": [],
1988         "fitness_moy": [],
1989         "freq_matrice": np.zeros((SIZE_Y, SIZE_X)),
1990         "terrain_matrice": np.zeros((SIZE_Y, SIZE_X)),
1991         "figures": [],
1992         "count": [],
1993         "best_athlete": {
1994             "fitness": 0,
1995             "genes": []
1996         },
1997         "athlete": {},
1998         "nb_executions": file_number,
1999         "performance": 0,
2000     }
2001
2002     fitness_temp = []
2003
2004     # Analyse de chaque fichier
2005     count = 0
2006     progress_bar = tqdm(total=len(filenames), desc="Analyse des fichiers",
2007 unit="file")
2008
2009     for filename in filenames:
2010         # Analyse du fichier
2011         file_data = analyse(os.path.join(foldername, filename))
2012
2013         # Ajout des données
2014         data["nb_generations"] += file_data["nb_generations"]
2015         data["freq_matrice"] += file_data["freq_matrice"]
2016         fitness_temp.append(file_data["fitness"])
2017         data["performance"] += 1 if file_data["is_success"] else 0
2018
2019         # Variables invariantes face aux exécutions de l'algorithme
2020         data["terrain_matrice"] = file_data["terrain_matrice"]
2021         data["figures"] = file_data["figures"]
2022         data["count"] = file_data["count"]
2023         data["athlete"] = file_data["athlete"]
2024
2025         # Valeurs sans cohérence face aux exécutions
2026         data["population_size"] = -1
2027         data["crossover_prob"] = -1
2028         data["mutation_prob"] = -1
2029         data["terminaison_age"] = -1
2030
2031         # Meilleur athlète
2032         if file_data["best_athlete"]["fitness"] > data["best_athlete"]
2033 ["fitness"]:
2034             data["best_athlete"] = file_data["best_athlete"]
2035
2036         count += 1
2037         progress_bar.update()
2038         logging.debug("Analyse de {} terminée ({}%)".format(filename,
2039 round((count/file_number)*100, 2)))

```

```

2036
2037     progress_bar.close()
2038
2039     # Moyenne des données
2040     data["freq_matrice"] /= len(filenames)
2041     data["nb_generations"] /= len(filenames)
2042     data["performance"] /= len(filenames)
2043
2044     # Moyenne de toutes les fitness par exécution
2045     # de l'algorithme génétique
2046     max_size = max(len(x) for x in fitness_temp)
2047     for i in range(max_size):
2048         moy_cur_fitness = [x[i] for x in fitness_temp if len(x) > i]
2049
2050     data["fitness"].append(sum(moy_cur_fitness)/len(moy_cur_fitness))
2051
2052     data["fitness"] = np.array(data["fitness"])
2053
2054     # Moyenne par génération
2055     data["fitness_moy"] = [sum(data["fitness"]
2056 [i:i+NUMBER_OF_CHROMOSOME_TO_KEEP])
2057 /NUMBER_OF_CHROMOSOME_TO_KEEP
2058 for i in range(0, len(data["fitness"]),
2059 NUMBER_OF_CHROMOSOME_TO_KEEP)]
2060
2061     data["fitness_moy"] = np.array(data["fitness_moy"])
2062
2063     return data
2064
2065 def makeEvolFitnessImg(list_fitness, nb_executions=1):
2066     """
2067     Crée l'image de l'évolution de la fitness au cours des générations
2068     """
2069     plt.subplot(1, 2, 1)
2070     mean_fitness = list_fitness.mean()
2071
2072     name = "Evolution du score au cours des générations ({}
2073 exécutions)" \
2074 .format(nb_executions)
2075
2076     # Liste de la moyenne des fitness par génération
2077     fitness_moy_by_gen =
2078 [sum(list_fitness[i:i+NUMBER_OF_CHROMOSOME_TO_KEEP])
2079 /NUMBER_OF_CHROMOSOME_TO_KEEP
2080 for i in range(0, len(list_fitness),
2081 NUMBER_OF_CHROMOSOME_TO_KEEP)]
2082
2083     # Affichage de la courbe
2084     plt.plot(fitness_moy_by_gen, color="blue", label="Score",
2085 linewidth=2)
2086
2087     plt.xlabel("Génération ({} athlètes/génération)" \
2088 .format(NUMBER_OF_CHROMOSOME_TO_KEEP))
2089     plt.ylabel("Score")
2090     plt.title(name)
2091
2092     # Affichage de la fitness maximale
2093     plt.axhline(y=mean_fitness, color="red", linestyle="--",
2094 label="Moyenne : {}".format(round(float(mean_fitness),
2095 2)),

```



```

2091         zorder = 3)
2092
2093     plt.legend()
2094
2095     def makeCasesImg(freq_matrice, terrain_matrice, best_athlete,
2096 filename):
2097         """
2098         Crée l'image de l'utilisation des cases au cours des générations
2099         """
2100         # Création de la figure et des axes
2101         ax = plt.subplot(1, 2, 1, aspect='equal')
2102
2103         cmap = mpl.colormaps['OrRd']
2104
2105         # Création des rectangles avec les valeurs de la matrice
2106         frequence
2107         for i in range(len(freq_matrice)):
2108             for j in range(len(freq_matrice[i])):
2109                 rect = mpatches.Rectangle((j, i), 1, 1,
2110 fc=cmap(freq_matrice[i, j]), lw=2)
2111                 ax.add_patch(rect)
2112
2113             # Affichage du chemin de l'athlete avec la meilleur fitness avec
2114             # des chiffres croissants
2115             for i in range(len(best_athlete["genes"])):
2116                 plt.text(best_athlete["genes"][i][0][0] + 0.5,
2117                     best_athlete["genes"][i][0][1] + 0.5,
2118                     str(i+1), color="black", ha="center", va="center")
2119
2120             # Mise en forme de l'image
2121             max_x, max_y, diff = len(freq_matrice[0]), len(freq_matrice), 1.
2122
2123             plt.title("Utilisation des cases au cours\ndes générations")
2124
2125             plt.colorbar(mpl.cm.ScalarMappable(norm=mpl.colors.Normalize(vmin=0,
2126 vmax=1), cmap=cmap), ax=ax)
2127             ax.set_xlim(0, max_x)
2128             ax.set_ylim(max_y, 0)
2129             ax.set_xticks(np.arange(max_x))
2130             ax.set_yticks(np.arange(max_y))
2131             ax.xaxis.tick_top()
2132             ax.grid()
2133
2134             # Création de l'affichage représentant le terrain
2135             ax2 = plt.subplot(1, 2, 2, aspect='equal')
2136
2137             # Couleurs représentant les différents types de cases
2138             colors = {
2139                 "black",
2140                 "grey",
2141                 "white"
2142             }
2143
2144             # Création des rectangles avec les cases de la matrice terrain
2145             for i in range(len(terrain_matrice)):
2146                 for j in range(len(terrain_matrice[i])):
2147                     rect = mpatches.Rectangle((j, i), 1, 1,
2148 fc=colors[int(terrain_matrice[i, j])],
2149 lw=2)

```

```

2147         ax2.add_patch(rect)
2148
2149         # Création de la légende
2150         empty_patch = mpatches.Patch(color='black', label='Case sol')
2151         wall_patch = mpatches.Patch(color='grey', label='Case mur')
2152         hole_patch = mpatches.Patch(color='white', label='Case barre')
2153
2154         plt.legend(handles=[empty_patch, wall_patch, hole_patch],
2155             loc='center left', bbox_to_anchor=(1, 0.5))
2156
2157         plt.title("Mobilier du terrain")
2158
2159         ax2.set_xlim(0, max_x)
2160         ax2.set_ylim(max_y, 0)
2161         ax2.set_xticks(np.arange(max_x))
2162         ax2.set_yticks(np.arange(max_y))
2163         ax2.xaxis.tick_top()
2164         ax2.grid()
2165
2166         # Marges
2167         plt.subplots_adjust(right=1.2, bottom=-0.8)
2168
2169         # Sauvegarde
2170         plt.savefig("traitement/{}_images/cases.png".format(filename),
2171             bbox_inches=Bbox([[0, -4], [9, 5]]),dpi=100)
2172         plt.close()
2173
2174     def makeFreqImg(list_figures, list_count, nb_executions=1):
2175         """
2176         Crée l'histogramme de la fréquence des figures
2177         """
2178         plt.subplot(1, 2, 2)
2179         title = "Fréquence des figures utilisées".format(nb_executions)
2180
2181         # Affichage de l'histogramme
2182         plt.bar(list_figures, list_count)
2183         plt.xticks(rotation="vertical")
2184         plt.xlabel("Figures")
2185         plt.ylabel("Fréquence")
2186         plt.title(title)
2187
2188         plt.autoscale(tight=False)
2189
2190     def constListImage(filename, const_dict):
2191         """
2192         Crée l'image de la liste des constantes utilisées pour les données
2193         """
2194         plt.axis('off')
2195
2196         # Converti un dictionnaire vers un tableau 2D
2197         const_array = []
2198         for key, value in const_dict.items():
2199             const_array.append([str(key), str(value)])
2200
2201         table = plt.table(cellText=const_array, collabels=["Constante",
2202 "Valeur"], loc='center')
2203         table.auto_set_font_size(False)
2204         table.set_fontsize(8)
2205
2206         plt.tight_layout()
2207         plt.savefig("traitement/{}_images/constantes.png".format(filename),
2208             dpi=100)

```

```

2207         plt.close()
2208
2209     def createStats(path=None, data=None):
2210         """
2211         Crée les images statistiques à partir d'un fichier ou de données
2212         fournies.
2213         Args:
2214             path (str, optional): Le chemin du fichier à analyser.
2215             Si non spécifié, les données doivent être fournies.
2216             data (dict, optional): Les données à analyser.
2217             Si non spécifié, le fichier sera analysé en utilisant le
2218             chemin spécifié.
2219         Returns:
2220             None: Cette fonction ne retourne aucune valeur.
2221         """
2222         if path is None and data is None:
2223             logging.error("Veuillez entrer un nom de fichier ou des
2224 données à analyser.")
2225             return
2226
2227         # Chronométrage du programme
2228         start_time = datetime.datetime.now()
2229
2230         # Récupération des données
2231         if data is None:
2232             data = analyse(path)
2233
2234         # Création du dossier de sauvegarde
2235         filename = path.split("data/")[-1] # Nom du fichier sans la partie
2236 "data/"
2237         os.makedirs("traitement/{}_images".format(filename),
2238             exist_ok=True)
2239
2240         logging.debug("Traitement des données...")
2241
2242         # Création des images
2243         makeEvolFitnessImg(data["fitness"],data["nb_executions"])
2244         makeFreqImg(data["figures"], data["count"], data["nb_executions"])
2245
2246         perf = " performance {%}%".format(round(data["performance"]*100,2))
2247         if "performance" in data.keys() \
2248             else " succès ? {}".format(data["is_success"])
2249
2250         name = "{}xp, {} générations/exécution en moy, {}
2251 individus/génération ({} exécutions)" \
2252             .format(
2253             data["athlete"]["xp"],
2254             round(data["nb_generations"]), data["population_size"],
2255             data["nb_executions"]
2256         )
2257         name += "\n" + perf
2258
2259         plt.suptitle(name)
2260         plt.subplots_adjust(bottom=0.1, left=-0.2, right=1.3, top=0.85,
2261 hspace=2)
2262
2263         plt.savefig("traitement/{}_images/freq&fitness.png".format(filename),
2264             bbox_inches=Bbox([[ -2, -1.3], [9, 5]]),dpi=100)

```

```

2260 plt.close()
2261
2262 constListImage(filename=filename, const_dict={
2263     "ATHLETE_XP": data["athlete"]["xp"],
2264     "CROSSOVER_PROB": data["crossover_prob"],
2265     "MUTATION_PROB": data["mutation_prob"],
2266     "POPULATION_SIZE": data["population_size"],
2267     "NUMBER_OF_CHROMOSOME_TO_KEEP":
2268         min(NUMBER_OF_CHROMOSOME_TO_KEEP,
data["population_size"]-1),
2269     "TERMINAISON_AGE": data["terminaison_age"],
2270     "INITIAL_POSITION": INITIAL_POSITION,
2271     "MAX_TICK_COUNT": MAX_TICK_COUNT,
2272     "ITERATION_NUMBER": ITERATION_NUMBER,
2273     "SIZE_X": SIZE_X,
2274     "SIZE_Y": SIZE_Y,
2275     "TICK_INTERVAL": TICK_INTERVAL,
2276     "MAX_FITNESS_GOTTEN": data["best_athlete"]["fitness"],
2277 })
2278
2279 makeCasesImg(data["freq_matrice"], data["terrain_matrice"],
2280             data["best_athlete"], filename)
2281
2282 logging.debug("Traitement terminé (en {})\n".format(
2283     datetime.datetime.now()-start_time))
2284
2285 def analyseStudy(foldername):
2286     """
2287     Analyse et création des images pour la comparaison avec l'étude
2288     """
2289     dataFolder = "data/"+foldername
2290
2291     # Récupération des noms des fichiers
2292     filenames = [f for f in os.listdir(dataFolder)
2293                 if os.path.isfile(os.path.join(dataFolder, f))]
2294     filenames.sort(key=lambda x : int(x.split(".")[0]))
2295
2296     # Construction du dictionnaire :
2297     # perfs = {
2298     #     p_c-p_m-population_size : [0, 1, 1, 0]
2299     #     (0 pour un échec, 1 pour un succès)
2300     #     Longueur = Nombre d'executions
2301     # }
2302     #
2303     perfs = {}
2304
2305     def pc_pmToString(file_data):
2306         pc = file_data["crossover_prob"]
2307         pm = file_data["mutation_prob"]
2308         popu = file_data["population_size"]
2309         return "{}-{}-{}".format(pc, pm, popu)
2310
2311     # Analyse de chaque fichier
2312     count = 0
2313     progress_bar = tqdm(total=len(filenames), desc="Analyse des
fichiers", unit="file")
2314     for filename in filenames:
2315         # Analyse du fichier
2316         file_data = analyse(os.path.join(dataFolder, filename))
2317
2318         category = pc_pmToString(file_data)
2319         if category not in perfs:

```

```

2320             perfs[category] = []
2321
2322             perfs[category].append(1 if file_data["is_success"] else 0)
2323
2324             count += 1
2325             progress_bar.update()
2326             logging.debug("Analyse de {} terminée ({}%)".format(filename,
2327                 round((count/len(filenames))*100, 2)))
2328
2329     progress_bar.close()
2330
2331     # Construction du dictionnaire :
2332     # perfsFinales = {
2333     #     "pc-pm": [float]
2334     # }
2335     perfsFinales = {}
2336
2337     logging.debug("Construction du graphique des succès...")
2338     for pc, pm in zip(PROBS_C, PROBS_M):
2339         perfsFinales["{}|{}".format(pc, pm)] = []
2340         for popu in POPULATIONS:
2341             perf = 0
2342             for success in perfs["{}-{}-{}".format(pc, pm, popu)]:
2343                 perf += success
2344             perf /= ITERATION_NUMBER
2345
2346             perfsFinales["{}|{}".format(pc, pm)].append(perf)
2347
2348     # Affichage des différentes courbes
2349     for key, value in perfsFinales.items():
2350         pc, pm = key.split("|")
2351
2352         plt.plot(POPULATIONS, value, label="p_c = {}; p_m = {}/1".format(pc,
2353             pm))
2354
2355     # Sauvegarde du tableau final
2356     saveDir = "traitement/study/{}".format(foldername)
2357     os.makedirs(saveDir, exist_ok=True)
2358
2359     plt.legend(prop={'size': 10})
2360     plt.xlabel("Taille de la population")
2361     plt.ylabel("Performances")
2362     plt.ylim((0, 1))
2363     plt.xscale('log')
2364     ax = plt.gca()
2365     ax.xaxis.set_major_formatter(mticker.ScalarFormatter())
2366     ax.set_xticks([2, 10, 100, 500, 1000, 2000])
2367     plt.xlim((2, 2000))
2368     plt.savefig("{}performances.png".format(saveDir), dpi=100)
2369     plt.close()
2370
2371     logging.info("Fichier sauvegardé : {}/performances.png".format(
2372         saveDir
2373     ))
2374
2375 if __name__ == "__main__":
2376     # Afficher les logs dans un fichier
2377     logging.basicConfig(level=logging.DEBUG,
2378         format='%(asctime)s - %(levelname)s - %(message)s',
2379         datefmt='%d-%m-%Y %H:%M:%S',
2380         filename='logs/Traitement - {}.txt'.format(

```

```

2381     datetime.datetime.now()).strftime(
2382         "%d-%m-%Y %H:%M:%S"
2383     )),
2384     filemode='w')
2385
2386     # Affichage dans la console
2387     console = logging.StreamHandler()
2388     console.setLevel(logging.INFO)
2389     formatter = logging.Formatter('%(asctime)s - %(levelname)s - %
(message)s')
2390     console.setFormatter(formatter)
2391     logging.getLogger('').addHandler(console)
2392
2393     # folder = "8xp/27-03-2024 11h32m49s/"
2394     # data = analyseFolder("data/" + folder)
2395     # createStats(path=folder+"/all", data=data)
2396
2397     analyseStudy("8xp/26-04-2024 18h45m31s")
2398
2399     _____
2400     '''
2401     Name : Elowan
2402     Creation : 30-06-2023 23:57:04
2403     Last modified : 21-05-2024 21:31:35
2404     File : utils.py
2405     '''
2406     import random
2407     import numpy as np
2408
2409     def weighted_random(mn, mx, mnweight, mxweight):
2410         """
2411         Exécute un random entre mn et mx avec une probabilité de mnweight
2412         d'avoir la plus basse valeur et mxweight d'avoir la plus haute
2413         valeur
2414         """
2415         return random.choices(range(mn, mx+1), \
2416             weights=np.linspace(mnweight,mxweight,(mx-mn)+1))[0]
2417
2418     def computeNextOccurrence(u: float, pm: float)->int:
2419         """
2420         Calcule la variable l de l'étude qui permet de réduire le
2421         nombre de calcul de variable aléatoire sans changer le succès
2422         de l'algorithme génétique.
2423         """
2424         if pm == 0.0: return 0
2425         else:
2426             val = (1/pm)*np.log(1-u)
2427             return int(val)
2428     _____

```