**Algorithm Genetic Algorithm**

Import 🡪 QtCore 🡪 PyQt5

Import 🡪 Settings 🡪 components

import 🡪 itemgetter 🡪 operator

Import 🡪 Counter 🡪 collections

Import 🡪 copy

Import 🡪 itertools

Import 🡪 numpy as np

Class 🡨 GeneticAlgorithm(QtCore.QThread)

# Current phase of the algorithm

statusSignal 🡨 QtCore.pyqtSignal(str)

# Genetic algorithm variable details

detailsSignal 🡨 QtCore.pyqtSignal(list)

# Running process type

operationSignal 🡨 QtCore.pyqtSignal(int)

# List of chromosomes for preview

dataSignal 🡨 QtCore.pyqtSignal(list)

Function 🡨 \_\_init\_\_(self, data)

self.averageFitness 🡨 0

self.pastAverageFitness 🡨 0

self.running 🡨 True

self.chromosomes 🡨 []

self.data 🡨 {

'rooms' [],

'instructors' [],

'sections' [],

'sharings' [],

'subjects' []

}

self.stayInRoomAssignments 🡨 {}

self.tournamentSize 🡨 .04

self.elitePercent 🡨 .05

self.mutationRate 🡨 .10

self.lowVariety 🡨 55

self.highestFitness 🡨 0

self.lowestFitness 🡨 100

self.elites 🡨 []

self.matingPool 🡨 []

self.offsprings 🡨 []

self.tempChromosome 🡨 None

self.tempSections 🡨 None

self.data 🡨 data

self.settings 🡨 Settings.getSettings()

self.stopWhenMaxFitnessAt <- self.settings['maximum\_fitness']

super().\_\_init\_\_()

Function 🡨 initialization(self)

# Generate population based on minimum population

self.generateChromosome(self.settings['minimum\_population'])

Function 🡨 generateChromosome(self, quantity)

for i in range(quantity)

self.statusSignal.emit('Creating #{} of {} Chromosomes'.format(i, quantity))

self.tempChromosome 🡨 Chromosome(self.data)

# {id [[subjectIds](, stay|roomId = False)]}

self.tempSections 🡨 sections 🡨 {key [value[2], value[3]] for (key, value) in

copy.deepcopy(self.data['sections']).items()}

# {id [subjectId, [sections]]}

self.tempSharings 🡨 sharings 🡨 copy.deepcopy(self.data['sharings'])

# [roomIds]

self.rooms 🡨 rooms 🡨 list(self.data['rooms'].keys())

# Distributed Room selection for staying sections

IF not len(self.stayInRoomAssignments)

selectedRooms 🡨 []

for section in sections

IF sections[section][1]

room 🡨 False

attempts 🡨 0

while not room

attempts += 1

candidate 🡨 np.random.choice(rooms)

IF attempts == self.settings['generation\_tolerance']

room 🡨 candidate

IF self.data['rooms'][candidate][1] == 'lec'

IF candidate not in selectedRooms

selectedRooms.append(copy.deepcopy(candidate))

room 🡨 candidate

sections[section][1] 🡨 room

self.stayInRoomAssignments[section] 🡨 room

ELSE

for section, room in self.stayInRoomAssignments.items()

sections[section][1] 🡨 room

# Remove subjects from sections that are already in sharing

for sharing in sharings.values()

for section in sharing[1]

sections[section][0].remove(sharing[0])

self.generateSubjectPlacementsForSharings(sharings)

self.generateSubjectPlacementsForSections(sections)

self.chromosomes.append(self.tempChromosome)

Function 🡨 generateSubjectPlacementsForSharings(self, sharings)

sharingOrder 🡨 list(sharings.keys())

np.random.shuffle(sharingOrder)

for sharing in sharingOrder

result 🡨 self.generateSubjectPlacement(sharings[sharing][1], sharings[sharing][0], sharing)

IF not result

self.tempChromosome.data['unplaced']['sharings'].append(sharing)

# {id [[subjectIds](, stay|roomId <- False)]}

Function 🡨 generateSubjectPlacementsForSections(self, sections)

# Maximum length of section subjects

maxSubjects 🡨 max(len(subjects[0]) for subjects in sections.values())

# Put one random section subject per turn

for i in range(maxSubjects)

for section in sections

subjectList 🡨 sections[section][0]

IF not len(subjectList)

continue

subjectToPlace 🡨 np.random.randint(0, len(subjectList))

result 🡨 self.generateSubjectPlacement([section], subjectList[subjectToPlace])

IF not result

self.tempChromosome.data['unplaced']['sections'][section].append(subjectList[subjectToPlace])

sections[section][0].pop(subjectToPlace)

# Section = [id], Subject = int (id)

Function 🡨 generateSubjectPlacement(self, section, subject, sharing<-False)

generating 🡨 True

generationAttempt 🡨 0

error 🡨 None

stayInRoom 🡨 False

IF section[0] not in

self.stayInRoomAssignments.keys()

ELSE self.stayInRoomAssignments[

section[0]]

subjectDetails 🡨 self.data['subjects'][subject]

room 🡨 stayInRoom IF stayInRoom ELSE None

# [[day/s], startingTimeSlot, length]

timeDetails 🡨 []

instructor 🡨 None

while generating

# Control generation to avoid impossible combinations

generationAttempt += 1

IF generationAttempt > self.settings['generation\_tolerance']

generating 🡨 False

return False

# Allow random meeting patterns IF generation is taking long

forceRandomMeeting 🡨 True IF generationAttempt > self.settings['generation\_tolerance'] / 2 ELSE False

# First time generation

IF not error

IF not stayInRoom or (stayInRoom and subjectDetails[6] == 'lab')

room 🡨 self.selectRoom(subject)

IF len(subjectDetails[4]) > 1

instructor 🡨 self.selectInstructor(subject)

ELIF len(subjectDetails[4])

instructor 🡨 subjectDetails[4][0]

ELSE

instructor 🡨 False

timeDetails 🡨 self.selectTimeDetails(subject, forceRandomMeeting)

ELSE

# Randomly select IF choosing new entry or replacing subject time details

IF error == 1 or error == 2

IF np.random.randint(0, 2)

error 🡨 3

ELIF error == 1

IF not stayInRoom or (stayInRoom and subjectDetails[6] == 'lab')

room 🡨 self.selectRoom(subject)

ELSE

error 🡨 3

ELSE

IF len(subjectDetails[4]) > 1

instructor 🡨 self.selectInstructor(subject)

ELSE

error 🡨 3

# Select subject time details

ELIF error == 3

timeDetails 🡨 self.selectTimeDetails(subject, forceRandomMeeting)

# [roomId, [sectionId], subjectId, instructorID, [day / s], startingTS, length(, sharingId)]

scheduleToInsert 🡨 [room, section, subject, instructor, \*timeDetails]

IF sharing

scheduleToInsert.append(sharing)

error 🡨 self.tempChromosome.insertSchedule(scheduleToInsert)

IF error is False

return True

Function 🡨 selectRoom(self, subject)

room 🡨 None

while not room

candidate 🡨 np.random.choice(self.rooms)

IF self.data['subjects'][subject][6] == self.data['rooms'][candidate][1]

room 🡨 candidate

return room

Function 🡨 selectInstructor(self, subject)

instructor 🡨 None

subjectInstructors 🡨 self.data['subjects'][subject][4]

while not instructor and len(subjectInstructors)

instructor 🡨 np.random.choice(subjectInstructors)

return instructor

Function 🡨 selectTimeDetails(self, subject, forceRandomMeeting)

meetingPatterns 🡨 [[0, 2, 4], [1, 3]]

days 🡨 [0, 1, 2, 3, 4, 5]

np.random.shuffle(days)

hours 🡨 self.data['subjects'][subject][1]

# Check IF hours can be splitted with minimum session of 1 hour or 2 timeslot

IF hours > 1.5 and ((hours / 3) % .5 == 0 or (hours / 2) % .5 == 0) and self.data['subjects'][subject][5]

# IF hours is divisible by two and three

IF (hours / 3) % .5 == 0 and (hours / 2) % .5 == 0

meetingPattern 🡨 np.random.choice(meetingPatterns)

IF len(meetingPattern) == 3

meetingPattern 🡨 days[03] IF forceRandomMeeting ELSE meetingPattern

hours 🡨 hours / 3

ELSE

meetingPattern <- days[02] IF forceRandomMeeting ELSE meetingPattern

hours 🡨 hours / 2

ELIF (hours / 3) % .5 == 0

meetingPattern 🡨 days[03] IF forceRandomMeeting ELSE meetingPatterns[0]

hours 🡨 hours / 3

ELSE

meetingPattern 🡨 days[02] IF forceRandomMeeting ELSE meetingPatterns[1]

hours 🡨 hours / 2

# Select random day slot

ELSE

meetingPattern 🡨 [np.random.randint(0, 6)]

# To convert hours into timetable timeslots

hours 🡨 hours / .5

startingTimeslot 🡨 False

# Starting slot selection

startingTime 🡨 self.settings['starting\_time']

endingTime 🡨 self.settings['ending\_time']

while not startingTimeslot

candidate 🡨 np.random.randint(0, endingTime - startingTime + 1)

# Validate IF subject will not overpass operation time

IF (candidate + hours) < endingTime - startingTime

startingTimeslot 🡨 candidate

return [meetingPattern, startingTimeslot, int(hours)]

Function 🡨 evaluate(self)

totalChromosomeFitness 🡨 0

self.pastAverageFitness 🡨 copy.deepcopy(self.averageFitness)

self.lowestFitness 🡨 100

self.highestFitness 🡨 0

for index, chromosome in enumerate(self.chromosomes)

self.statusSignal.emit('Evaluating #{} of {} Chromosomes'.format(index + 1, len(self.chromosomes)))

chromosome.fitness 🡨 self.evaluateAll(chromosome)

totalChromosomeFitness += chromosome.fitness

self.averageFitness 🡨 totalChromosomeFitness / len(self.chromosomes)

self.highestFitness 🡨 chromosome.fitness IF chromosome.fitness > self.highestFitness ELSE self.highestFitness

self.lowestFitness 🡨 chromosome.fitness IF chromosome.fitness < self.lowestFitness ELSE self.lowestFitness

chromosomeFitness 🡨 sorted(enumerate(map(lambda chromosome chromosome.fitness, self.chromosomes)),

key 🡨 itemgetter(1))

# Emit top five chromosomes

self.dataSignal.emit(

list(map(lambda chromosome [self.chromosomes[chromosome[0]], chromosome[1]], chromosomeFitness[-5])))

# Evaluation weight depends on settings

Function 🡨 evaluateAll(self, chromosome)

subjectPlacement 🡨 self.evaluateSubjectPlacements(chromosome)

lunchBreak 🡨 self.evaluateLunchBreak(chromosome) IF self.settings['lunchbreak'] ELSE 100

studentRest 🡨 self.evaluateStudentRest(chromosome)

instructorRest 🡨 self.evaluateInstructorRest(chromosome)

idleTime 🡨 self.evaluateStudentIdleTime(chromosome)

meetingPattern 🡨 self.evaluateMeetingPattern(chromosome)

instructorLoad 🡨 self.evaluateInstructorLoad(chromosome)

chromosome.fitnessDetails 🡨 copy.deepcopy([subjectPlacement, lunchBreak, studentRest, instructorRest, idleTime,

meetingPattern, instructorLoad])

matrix 🡨 self.settings['evaluation\_matrix']

return round(

(subjectPlacement \* matrix['subject\_placement'] / 100) +

(lunchBreak \* matrix['lunch\_break'] / 100) +

(studentRest \* matrix['student\_rest'] / 100) +

(instructorRest \* matrix['instructor\_rest'] / 100) +

(idleTime \* matrix['idle\_time'] / 100) +

(meetingPattern \* matrix['meeting\_pattern'] / 100) +

(instructorLoad \* matrix['instructor\_load'] / 100),

2

)

# = ((subjects - unplacedSubjects) / subjects) \* 100

Function 🡨 evaluateSubjectPlacements(self, chromosome)

sections 🡨 copy.deepcopy({key value[2] for key, value in self.data['sections'].items()})

sharings 🡨 self.data['sharings']

chromosomeUnplacedData 🡨 chromosome.data['unplaced']

# Number of subjects that are in sharing

sharingSubjects 🡨 0

# Remove section subjects that are shared

for sharing in sharings.values()

# Sharing subjects is increased based on number of sections sharing the subject

sharingSubjects += len(sharing[1])

for section in sharing[1]

sections[section].remove(sharing[0])

# Combined list of section subjects

sectionSubjects 🡨 len(list(itertools.chain.from\_iterable(sections.values())))

# Combined list of subjects

totalSubjects 🡨 sectionSubjects + sharingSubjects

# Number of shared subjects that are not placed

unplacedSharingSubjects 🡨 0

for sharing in chromosomeUnplacedData['sharings']

# Sharing subjects is increased based on number of sections sharing the subject

unplacedSharingSubjects += len(sharings[sharing][1])

# Length of unplaced section subjects

unplacedSectionSubjects 🡨 len(list(itertools.chain.from\_iterable(chromosomeUnplacedData['sections'].values())))

totalUnplacedSubjects 🡨 unplacedSharingSubjects + unplacedSectionSubjects

return round(((totalSubjects - totalUnplacedSubjects) / totalSubjects) \* 100, 2)

# = ((sectionDays - noLunchDays) / sectionDays) \* 100

Function 🡨 evaluateLunchBreak(self, chromosome)

sectionDays 🡨 0

noLunchDays 🡨 0

for section in chromosome.data['sections'].values()

# [roomId, instructorId, [day / s], startingTS, length]

details 🡨 section['details']

# A temporary map for days and lunch period

# {day [22, 23, 24, 25]}

# TS 22-25 11 AM - 1 PM

tempScheduleMap 🡨 {key [22, 23, 24, 25] for key in range(6)}

# Days that the section used

tempSectionDays 🡨 []

# Loop through each subject and remove lunch period timeslots that are occupied.

for subject in details.values()

IF not len(subject)

continue

for day in subject[2]

IF day not in tempSectionDays

tempSectionDays.append(day)

# Check IF subject is in lunch period

for timeslot in range(subject[3], subject[3] + subject[4])

IF timeslot in tempScheduleMap[day]

tempScheduleMap[day].remove(timeslot)

# IF whole day's lunch period is taken, count it as no lunch break

for day in tempScheduleMap.values()

IF not len(day)

noLunchDays +=1

sectionDays += len(tempSectionDays)

return round(((sectionDays - noLunchDays) / sectionDays) \* 100, 2)

# = ((sectionDays - noRestDays) / sectionDays) \* 100

Function 🡨 evaluateStudentRest(self, chromosome)

sectionDays 🡨 0

noRestDays 🡨 0

for section in chromosome.data['sections'].values()

# Sections week

week 🡨 {day [] for day in range(6)}

for subject in section['details'].values()

IF not len(subject)

continue

# Add section subject timeslots to sections week

for day in subject[2]

for timeslot in range(subject[3], subject[3] + subject[4])

week[day].append(timeslot)

week[day].sort()

for day in week.values()

IF not len(day)

continue

sectionDays += 1

IF len(day) < 6

continue

hasViolated 🡨 False

# Steps of how many three hours per day a section has (Increments of 30 minutes)

for threeHours in range(6, len(day) + 1)

IF hasViolated

continue

# Compare consecutive timeslot to section's day timeslot

IF [timeslot for timeslot in range(day[threeHours - 6], day[threeHours - 6] + 6)] == day[

threeHours - 6 threeHours]

hasViolated 🡨 True

noRestDays += 1

return round(((sectionDays - noRestDays) / sectionDays) \* 100, 2)

# = ((instructorTeachingDays - noRestDays) / instructorTeachingDays) \* 100

Function 🡨 evaluateInstructorRest(self, chromosome)

instructorTeachingDays 🡨 0

noRestDays 🡨 0

for instructor in chromosome.data['instructors'].values()

# Instructor week

week 🡨 {day [] for day in range(6)}

for timeslot, timeslotRow in enumerate(instructor)

for day, value in enumerate(timeslotRow)

# Add timeslot to instructor week IF teaching

IF value

week[day].append(timeslot)

for day in week.values()

IF not len(day)

continue

instructorTeachingDays += 1

IF len(day) < 6

continue

hasViolated 🡨 False

# Steps of how many three hours per day a section has (Increments of 30 minutes)

for threeHours in range(6, len(day) + 1)

IF hasViolated

continue

# Compare consecutive timeslot to section's day timeslot

IF [timeslot for timeslot in range(day[threeHours - 6], day[threeHours - 6] + 6)] == day[

threeHours - 6 threeHours]

hasViolated 🡨 True

noRestDays += 1

IF not instructorTeachingDays

return 100.00

return round(((instructorTeachingDays - noRestDays) / instructorTeachingDays) \* 100, 2)

# = ((sectionDays - idleDays) / sectionDays) \* 100

Function 🡨 evaluateStudentIdleTime(self, chromosome)

sectionDays 🡨 0

idleDays 🡨 0

for section in chromosome.data['sections'].values()

week 🡨 {day [] for day in range(6)}

for subject in section['details'].values()

IF not len(subject)

continue

# Add section subject timeslots to sections week

for day in subject[2]

week[day].append([timeslot for timeslot in range(subject[3], subject[3] + subject[4])])

week[day].sort()

for day in week.values()

IF not len(day)

continue

sectionDays += 1

# For every 6 TS that the day occupies, there is 1 TS allowable break

allowedBreaks 🡨 round((len(list(itertools.chain.from\_iterable(day))) / 6), 2)

# IF the decimal of allowed breaks is greater than .6, consider it as an addition

IF (allowedBreaks > 1 and allowedBreaks % 1 > 0.60) or allowedBreaks % 1 > .80

allowedBreaks += 1

for index, timeslots in enumerate(day)

IF index == len(day) - 1 or allowedBreaks < 0

continue

# Consume the allowable breaks with the gap between each subject of the day

IF timeslots[-1] != day[index + 1][0] - 1

allowedBreaks -= timeslots[-1] + day[index + 1][0] - 1

IF allowedBreaks < 0

idleDays += 1

return round(((sectionDays - idleDays) / sectionDays) \* 100, 2)

# = ((placedSubjects - badPattern) / placedSubjects) \* 100

Function 🡨 evaluateMeetingPattern(self, chromosome)

placedSubjects 🡨 0

badPattern 🡨 0

for section in chromosome.data['sections'].values()

for subject in section['details'].values()

IF not len(subject) or len(subject[2]) == 1

continue

placedSubjects += 1

# Check IF subject has unusual pattern

IF subject[2] not in [[0, 2, 4], [1, 3]]

badPattern += 1

return round(((placedSubjects - badPattern) / placedSubjects) \* 100, 2)

Function 🡨 evaluateInstructorLoad(self, chromosome)

activeInstructors 🡨 {}

activeSubjects 🡨 []

# Get list of active subjects

for section in self.data['sections'].values()

activeSubjects += section[2]

subjects 🡨 self.data['subjects']

sharings 🡨 self.data['sharings']

# Get list of active instructors and their potential load

for subject in activeSubjects

# Exclude subjects that have less than 1 candidate instructor

IF len(subjects[subject][4]) <= 1

continue

for instructor in subjects[subject][4]

IF instructor not in activeInstructors.keys()

activeInstructors[instructor] 🡨 [0, 0]

activeInstructors[instructor][0] += int(subjects[subject][1] / .5)

# Remove load from instructors that is duplicated due to sharing

for sharing in sharings.values()

subject 🡨 subjects[sharing[0]]

IF len(subject[4]) <= 1

continue

for instructor in subject[4]

activeInstructors[instructor][0] -= int(subject[1] / .5) \* (len(sharing[1]) - 1)

# Fill up active instructors with actual load

for instructor, details in chromosome.data['instructors'].items()

for timeslotRow in details

for day in timeslotRow

IF day and instructor in activeInstructors.keys()

activeInstructors[instructor][1] += 1

instructorLoadAverage 🡨 0

# Calculate the average instructor load. Closer to 50% means equal distribution which is better

for instructor in activeInstructors.values()

instructorLoadAverage += (instructor[1] / instructor[0]) \* 100

IF not len(activeInstructors)

return 100.00

instructorLoadAverage 🡨 round(instructorLoadAverage / len(activeInstructors), 2)

return instructorLoadAverage

Function 🡨 getAllFitness(self)

return [chromosome.fitness for chromosome in self.chromosomes]

Function 🡨 adapt(self)

deviation 🡨 self.getFitnessDeviation()

self.alignPopulation(deviation[0], deviation[1])

self.adjustMutationRate()

# sigma = [sigma], sigmaInstances = {sigma instance%}

Function 🡨 getFitnessDeviation(self)

populationCount 🡨 len(self.chromosomes)

fitnesses 🡨 [chromosome.fitness for chromosome in self.chromosomes]

mean 🡨 np.mean(fitnesses)

sigmas 🡨 [int(fitness - mean) for fitness in fitnesses]

sigmaInstances 🡨 {sigma (instance / populationCount) \* 100 for sigma, instance in

dict(Counter(sigmas)).items()}

return [sigmas, sigmaInstances]

Function 🡨 alignPopulation(self, sigmas, sigmaInstances)

populationCount 🡨 len(self.chromosomes)

sigmaStartingInstance 🡨 list(sigmaInstances.values())[0]

IF sigmaStartingInstance > self.lowVariety

# Add the excess percentage of instances on first sigma to population

generate 🡨 int((int(sigmaStartingInstance - self.lowVariety) / 100) \* populationCount)

while generate + populationCount > self.settings['maximum\_population']

generate -= 1

self.generateChromosome(generate)

ELSE

# Remove the excess percentage of instances on first sigma to population

sortedSigmas <-- sorted(enumerate(sigmas), key<-itemgetter(1))

remove 🡨 int((int(self.lowVariety - sigmaStartingInstance) / 100) \* populationCount)

while populationCount - remove < self.settings['minimum\_population']

remove -🡨 1

remove 🡨 [sortedSigmas[index][0] for index in range(remove)]

self.chromosomes 🡨 [chromosome for index, chromosome in enumerate(self.chromosomes) IF index not in remove]

# Increase mutation rate for low performing generations and decrease for good performance

Function 🡨 adjustMutationRate(self)

IF (self.averageFitness - self.pastAverageFitness < 0) or (

abs(self.averageFitness - self.pastAverageFitness) <= self.settings[

'mutation\_rate\_adjustment\_trigger']) and not self.mutationRate >= 100

self.mutationRate += .05

ELIF self.mutationRate > .10

self.mutationRate -= .05

self.mutationRate 🡨 round(self.mutationRate, 2)

# Selects top 5% of population and performs tournament to generate remaining candidates

Function 🡨 selection(self)

population 🡨 len(self.chromosomes)

chromosomeFitness 🡨 [self.chromosomes[chromosome].fitness for chromosome in range(len(self.chromosomes))]

# Select number of elites that will ensure there will be even offspring to be generated

eliteCount 🡨 round(population \* self.elitePercent)

IF population % 2 == 0

eliteCount 🡨 eliteCount IF eliteCount % 2 == 0 ELSE eliteCount + 1

ELSE

eliteCount 🡨 eliteCount IF eliteCount % 2 != 0 ELSE eliteCount + 1

self.statusSignal.emit('Selecting {} Elites'.format(eliteCount))

sortedFitness 🡨 sorted(enumerate(chromosomeFitness), key=itemgetter(1))

elites 🡨 list(map(lambda chromosome chromosome[0], sortedFitness[eliteCount \* -1]))

matingPool 🡨 []

matingPoolSize 🡨 int((population - eliteCount) / 2)

tournamentSize 🡨 int(self.tournamentSize \* population)

IF tournamentSize > 25

tournamentSize 🡨 25

# Fill mating pool with couples selected by multiple tournaments

for i in range(matingPoolSize)

self.statusSignal.emit('Creating #{} of {} Couples'.format(i + 1, matingPoolSize))

couple 🡨 []

while len(couple) != 2

winner 🡨 self.createTournament(tournamentSize, chromosomeFitness)

IF winner not in couple

couple.append(winner)

matingPool.append(couple)

self.elites 🡨 elites

self.matingPool 🡨 matingPool

# size = int, population = [fitness]

Function 🡨 createTournament(self, size, population)

participants 🡨 []

# Select participants

for i in range(size)

candidate 🡨 False

while candidate is False

candidate 🡨 np.random.randint(0, len(population))

IF candidate in participants

candidate 🡨 False

continue

participants.append(candidate)

winner 🡨 participants[0]

for participant in participants

IF population[participant] > population[winner]

winner 🡨 participant

return winner

Function 🡨 crossover(self)

offspringCount 🡨 1

self.offsprings 🡨 []

for couple in self.matingPool

self.statusSignal.emit(

'Creating #{} of {} Offsprings'.format(offspringCount, len(self.chromosomes) - len(self.elites)))

self.offsprings.append(self.createOffspring(couple))

offspringCount += 1

couple.reverse()

self.statusSignal.emit(

'Creating #{} of {} Offsprings'.format(offspringCount, len(self.chromosomes) - len(self.elites)))

self.offsprings.append(self.createOffspring(couple))

offspringCount += 1

self.elites 🡨 list(map(lambda elite copy.deepcopy(self.chromosomes[elite]), self.elites))

self.chromosomes 🡨 self.offsprings + self.elites

# Returns a chromosome containing a mix of parents genes

Function 🡨 createOffspring(self, parent)

self.tempChromosome 🡨 offspring 🡨 Chromosome(self.data)

parentA 🡨 self.chromosomes[parent[0]]

parentB 🡨 self.chromosomes[parent[1]]

parentAShareables 🡨 {

'sharings' {},

'sections' {}

}

# Parent A shall provide half of its genes

parentASharings 🡨 parentA.data['sharings']

IF len(parentASharings) > 1

# Amount of sharings to get

sharingCarve 🡨 round(len(parentASharings) / 3)

# Middlemost element with bias to left

startingPoint 🡨 int(len(parentASharings) / 2) - (sharingCarve - 1)

for index in range(startingPoint, startingPoint + sharingCarve)

# Take note that index does not mean it is the key of the sharings

# [{sharingId details}]

sharings 🡨 [id for id in parentASharings.keys()]

for sharing in sharings[startingPointstartingPoint + sharingCarve]

parentAShareables['sharings'][sharing] 🡨 parentASharings[sharing]

# Raw list of parent A sections with reduced subjects from sharings

parentASections 🡨 {}

for section, value in copy.deepcopy(parentA.data['sections']).items()

parentASections[section] 🡨 value['details']

for sharing in self.data['sharings'].values()

for section in sharing[1]

parentASections[section].pop(sharing[0])

parentASections 🡨 {key value for key, value in filter(lambda item len(item[1]) > 1, parentASections.items())}

# Calculate the shareables of each section

for section, values in parentASections.items()

# Amount of section subjects to share

sectionCarve 🡨 round(len(values) / 3)

# Middlemost element with bias to left

startingPoint 🡨 int(len(values) / 2) - (sectionCarve - 1)

subjects 🡨 [id for id in values.keys()]

for index in range(startingPoint, startingPoint + sectionCarve)

IF section not in parentAShareables['sections']

parentAShareables['sections'][section] 🡨 {}

parentAShareables['sections'][section][subjects[index]] <- values[subjects[index]]

parentBShareables 🡨 {

'sharings' {},

'sections' {}

}

# Add remaining sharings from parent B

for id, sharing in parentB.data['sharings'].items()

IF id not in parentAShareables['sharings'].keys()

parentBShareables['sharings'][id] 🡨 sharing

# Create list of parent B sections

parentBSections 🡨 {}

for section, value in copy.deepcopy(parentB.data['sections']).items()

parentBSections[section] 🡨 value['details']

for sharing in self.data['sharings'].values()

for section in sharing[1]

parentBSections[section].pop(sharing[0])

# Create list of subjects that are not in parent A shareables

for section in parentBSections

parentBShareables['sections'][section] 🡨 {}

for id, subject in parentBSections[section].items()

IF not (id in list(parentAShareables['sections'][section].keys()))

parentBShareables['sections'][section][id] 🡨 subject

# List of unplaced sharings with or without data

unplacedSharings 🡨 {}

# Insert parent A sharings into chromosome

sharings 🡨 self.data['sharings']

for id, sharing in parentAShareables['sharings'].items()

IF not len(sharing)

unplacedSharings[id] 🡨 []

continue

offspring.insertSchedule([sharing[0], sharings[id][1], sharings[id][0], sharing[1], \*sharing[25], id])

# Add parent B subjects in random manner

parentBSharings 🡨 list(parentBShareables['sharings'].keys())

np.random.shuffle(parentBSharings)

for id in parentBSharings

sharing 🡨 parentBShareables['sharings'][id]

IF not len(sharing)

unplacedSharings[id] 🡨 []

continue

IF offspring.insertSchedule([sharing[0], sharings[id][1], sharings[id][0], sharing[1], \*sharing[25], id])

unplacedSharings[id] 🡨 sharing

# List of unplaced subjects with or without data

unplacedSectionSubjects 🡨 {}

# Insert parent A section subjects into chromosome

for section, subjects in parentAShareables['sections'].items()

IF section not in unplacedSectionSubjects.keys()

unplacedSectionSubjects[section] 🡨 {}

for subject, details in subjects.items()

IF not len(details)

unplacedSectionSubjects[section][subject] 🡨 []

continue

IF offspring.insertSchedule([details[0], [section], subject, details[1], \*details[25]])

unplacedSectionSubjects[section][subject] 🡨 details

# Insert parent B section subjects into chromosome

for section, subjects in parentBShareables['sections'].items()

IF section not in unplacedSectionSubjects.keys()

unplacedSectionSubjects[section] 🡨 {}

for subject, details in subjects.items()

IF not len(details)

unplacedSectionSubjects[section][subject] 🡨 []

continue

IF offspring.insertSchedule([details[0], [section], subject, details[1], \*details[25]])

unplacedSectionSubjects[section][subject] 🡨 details

# Attempt to insert unplaced sharings

for sharing in copy.deepcopy(unplacedSharings).keys()

IF self.generateSubjectPlacement(sharings[sharing][1], sharings[sharing][0], sharing)

unplacedSharings.pop(sharing)

# Attempt to insert unplaced section subjects

for section, subjects in copy.deepcopy(unplacedSectionSubjects).items()

for subject, detail in subjects.items()

IF self.generateSubjectPlacement([section], subject)

unplacedSectionSubjects[section].pop(subject)

return offspring

Function 🡨 mutation(self)

sharings 🡨 self.data['sharings']

sections 🡨 self.data['sections']

mutationCandidates 🡨 {

'sections' {},

'sharings' [key for key in sharings.keys()]

}

# Prepare clean list of subject placement with consideration for sharing

for section, data in copy.deepcopy(sections).items()

mutationCandidates['sections'][section] 🡨 data[2]

for sharing in sharings.values()

for section in sharing[1]

mutationCandidates['sections'][section].remove(sharing[0])

IF not len(mutationCandidates['sharings'])

mutationCandidates.pop('sharings')

for section in copy.deepcopy(mutationCandidates['sections'])

IF not len(mutationCandidates['sections'][section])

mutationCandidates['sections'].pop(section)

# Randomly select chromosomes to mutate

for index, chromosome in enumerate(copy.deepcopy(self.chromosomes))

IF np.random.randint(100) > (self.mutationRate \* 100) - 1

continue

self.statusSignal.emit('Mutating Chromosome #{}'.format(index + 1))

self.tempChromosome 🡨 Chromosome(self.data)

# Select a gene to mutate

mutating 🡨 np.random.choice(list(mutationCandidates.keys()))

IF mutating == 'sections'

section 🡨 np.random.choice(list(mutationCandidates['sections'].keys()))

mutating 🡨 ['sections', section, np.random.choice(mutationCandidates['sections'][section])]

ELSE

mutating 🡨 ['sharing', np.random.choice(mutationCandidates['sharings'])]

# Replicate chromosome except the mutating gene

for sharing in mutationCandidates['sharings'] IF 'sharings' in mutationCandidates ELSE []

IF mutating[0] == 'sharing' and sharing == mutating[1]

continue

details 🡨 chromosome.data['sharings'][sharing]

IF len(details)

self.tempChromosome.insertSchedule(

[details[0], sharings[sharing][1], sharings[sharing][0], details[1], \*details[25], sharing])

for section, subjects in mutationCandidates['sections'].items()

for subject in subjects

IF mutating[0] == 'sections' and mutating[1] == section and mutating[2] == subject

continue

details 🡨 chromosome.data['sections'][section]['details'][subject]

IF len(details)

self.tempChromosome.insertSchedule([details[0], [section], subject, details[1], \*details[25]])

# Generate mutation

IF mutating[0] == 'sharing'

self.generateSubjectPlacement(sharings[mutating[1]][1], sharings[mutating[1]][0], mutating[1])

ELSE

self.generateSubjectPlacement([mutating[1]], mutating[2])

self.chromosomes[index] 🡨 copy.deepcopy(self.tempChromosome)

Function 🡨 run(self)

self.statusSignal.emit('Initializing')

self.initialization()

generation 🡨 0

runThread 🡨 True

while (runThread)

IF self.running

generation += 1

self.statusSignal.emit('Preparing Evaluation')

self.evaluate()

self.detailsSignal.emit(

[generation, len(self.chromosomes), int(self.mutationRate \* 100), round(self.averageFitness, 2),

round(self.pastAverageFitness, 2), self.highestFitness, self.lowestFitness])

IF self.highestFitness >= self.settings['maximum\_fitness']

self.statusSignal.emit('Reached the Highest Fitness')

self.operationSignal.emit(1)

self.running 🡨 runThread 🡨 False

break

IF self.settings['maximum\_generations'] < generation - 1

self.statusSignal.emit('Hit Maximum Generations')

self.operationSignal.emit(1)

self.running 🡨 runThread 🡨 False

break

self.statusSignal.emit('Tweaking Environment')

self.adapt()

self.statusSignal.emit('Preparing Selection')

self.selection()

self.statusSignal.emit('Preparing Crossover')

self.crossover()

self.statusSignal.emit('Preparing Mutation')

self.mutation()

Class 🡨 Chromosome

# data = {

# sections && sharings {

# id {

# details {

# subject [roomId,

# instructorId,

# [day / s],

# startingTS,

# length

# ]

# },

# schedule [days]

# }

# },

# instructors && rooms {

# id [

# [days] // Timeslots

# [1, None, 1, None, 1, False] // Example

# // None = Vacant, False = Unavailable

# ]

# },

# unplaced {

# 'sharings' [], // List of unplaced sharings

# 'sections' {

# id [] // Section ID and unplaced subjects

# }

# }

# }

Function 🡨 \_\_init\_\_(self, data)

self.fitness 🡨 0

self.fitnessDetails 🡨 []

self.data 🡨 {

'sections' {},

'sharings' {},

'instructors' {},

'rooms' {},

'unplaced' {

'sharings' [],

'sections' {}

}

}

self.rawData 🡨 data

self.settings 🡨 Settings.getSettings()

self.buildChromosome()

Function 🡨 buildChromosome(self)

rawData 🡨 self.rawData

# {id {details [subject []], schedule [days]}}

sections 🡨 rawData['sections']

for section in sections

self.data['sections'][section] 🡨 {'details' {}, 'schedule' []}

self.data['sections'][section]['details'] 🡨 {key [] for key in sections[section][2]}

sectionTimetable 🡨 []

for timeslotRow in sections[section][1]

sectionTimetable.append([None IF day == 'Available' ELSE False for day in timeslotRow])

self.data['sections'][section]['schedule'] 🡨 sectionTimetable

self.data['unplaced']['sections'][section] 🡨 []

# {id [subjectId [details]]}

sharings 🡨 rawData['sharings']

for sharing in sharings

self.data['sharings'][sharing] 🡨 []

# {id [days]}

instructors 🡨 rawData['instructors']

for instructor in instructors

instructorTimetable 🡨 []

for timeslotRow in instructors[instructor][2]

instructorTimetable.append([None IF day == 'Available' ELSE False for day in timeslotRow])

self.data['instructors'][instructor] 🡨 instructorTimetable

# {id [days]}

rooms 🡨 rawData['rooms']

for room in rooms

roomTimetable 🡨 []

for timeslotRow in rooms[room][2]

roomTimetable.append([None IF day == 'Available' ELSE False for day in timeslotRow])

self.data['rooms'][room] 🡨 roomTimetable

# [roomId, [sectionId], subjectId, instructorID, [day/s], startingTS, length(, sharingId)]

Function 🡨 insertSchedule(self, schedule)

# Validate schedule details

isValid 🡨 self.validateSchedule(copy.deepcopy(schedule))

IF isValid is not True

return isValid

data 🡨 self.data

# [roomId, instructorId, [day/s], startingTS, length]

subjectDetails 🡨 [schedule[0], schedule[3], schedule[4], schedule[5], schedule[6]]

# Check IF schedule is for sharing

IF len(schedule) > 7

data['sharings'][schedule[-1]] 🡨 subjectDetails

# Insert details into section data

for section in schedule[1]

data['sections'][section]['details'][schedule[2]] 🡨 subjectDetails

# Update instructor and room timetable

for timeslot in range(schedule[5], schedule[5] + schedule[6])

for day in schedule[4]

IF schedule[3]

data['instructors'][schedule[3]][timeslot][day] 🡨 schedule[1]

data['rooms'][schedule[0]][timeslot][day] 🡨 schedule[1]

# False signIFies no error in insertion

return False

Function 🡨 validateSchedule(self, schedule)

IF not self.isRoomTimeslotAvailable(schedule)

return 1

IF not self.isInstructorTimeslotAvailable(schedule)

return 2

IF not self.isSectionTimeslotAvailable(schedule)

return 3

return True

Function 🡨 isRoomTimeslotAvailable(self, schedule)

room 🡨 self.data['rooms'][schedule[0]]

for timeslotRow in range(schedule[5], schedule[5] + schedule[6])

for day in schedule[4]

IF room[timeslotRow][day] is not None

return False

return True

Function 🡨 isSectionTimeslotAvailable(self, schedule)

rooms 🡨 self.data['rooms']

sections 🡨 self.data['sections']

# Check for each room IF on the given subject range, the section has Class <-

for room in rooms

for timeslotRow in range(schedule[5], schedule[5] + schedule[6])

for day in schedule[4]

roomDayTimeslot 🡨 rooms[room][timeslotRow][day]

# Check IF timeslot is blank

IF roomDayTimeslot is None

continue

# Check IF section is in timeslot

for section in schedule[1]

IF section in roomDayTimeslot

return False

# Check for section unavailable times

for section in schedule[1]

for timeslotRow in range(schedule[5], schedule[5] + schedule[6])

for day in schedule[4]

IF sections[section]['schedule'][timeslotRow][day] is not None

return False

return True

Function 🡨 isInstructorTimeslotAvailable(self, schedule)

# Pass IF no instructor is set

IF not schedule[3]

return True

instructor 🡨 self.data['instructors'][schedule[3]]

for timeslotRow in range(schedule[5], schedule[5] + schedule[6])

for day in schedule[4]

IF instructor[timeslotRow][day] is not None

return False

# Check IF instructor can still teach

maxLoad 🡨 self.rawData['instructors'][schedule[3]][1] \* 2

for timeslotRow in instructor

for day in timeslotRow

IF day

maxLoad -= 1

IF maxLoad < 0

return False

return True