**ABSTRACT**

Post Covid outbreak, the whole world is recovering from the loss it had incurred, especially education sector which is still suffering from the gross mismanagement. There are hundreds and thousands of students and a few teachers to teach them and this is not only the story of the author’s university but it is applicable to each and every university or college. Scheduling the lectures and labs manually for that many batches having that many subjects including electives would really be stressful situation for anyone. In this world which is on the way to bring the age of automation doing scheduling manually is a sin. Due to this, AI based schedulers are needed and author is presenting one of many ways to implement it. The whole world is working on the idea of trial and error so is the one. Author is using genetic algorithm to reach the target. This scheduler is making different chromosomes and then through the process of trial and error rejecting the ones with clashes to provide the best offspring or result. This process seems lengthy i.e. contains a very high time complexity but at the same time is giving accurate result as it is tapping on all the possible scenarios. Generating scheduler is a one-time process then people generally stores the result so gaining accuracy on the cost of time is a win-win situation for all.

**INTRODUCTION**

The method of automating the generation of a timetable that fits within the constraints of a scenario is known as AI-based Timetable Scheduling. According to GOI (Government of India) data, India had 8,997 AICTE-approved institutes as of February 2022. As of 2020-21, according to AISHE data, the country has 1,043 universities and 42,343 colleges. At many of these institutions, timetable scheduling is currently done manually. The amount of physical work required by these institutes to develop a timetable and adjust it based on faculty and resource availability can be time-consuming. The authors propose that this procedure be automated by creating an efficient mechanism for generating AI-based timetables. AI-based solutions are a relatively new technique to address problems and improve existing mechanisms in a wide range of economic sectors by improving existing solutions. The authors' proposed AI-based Timetable Scheduler seeks to apply Genetic Algorithm, an optimization algorithm used in AI-based solutions. The Genetic Algorithm is a metaheuristic method for simulating natural selection. This study seeks to create a schedule with the goal of overcoming all constraints and providing the best solution feasible. The proposed mechanism accepts manual inputs, in which the user enters information on instructors, rooms, sections, and subjects, or imports directly from a csv file, in which case manual input of sections is required. Then it generates a schedule that can be accessed by different classes, where Section, Room, and Instructor are the classes, and each class has its own schedule for its subclasses (for example, Instructors class will have a feature to view timetable of each instructor). These results can be further exported in csv file for ease of usage and later processing.

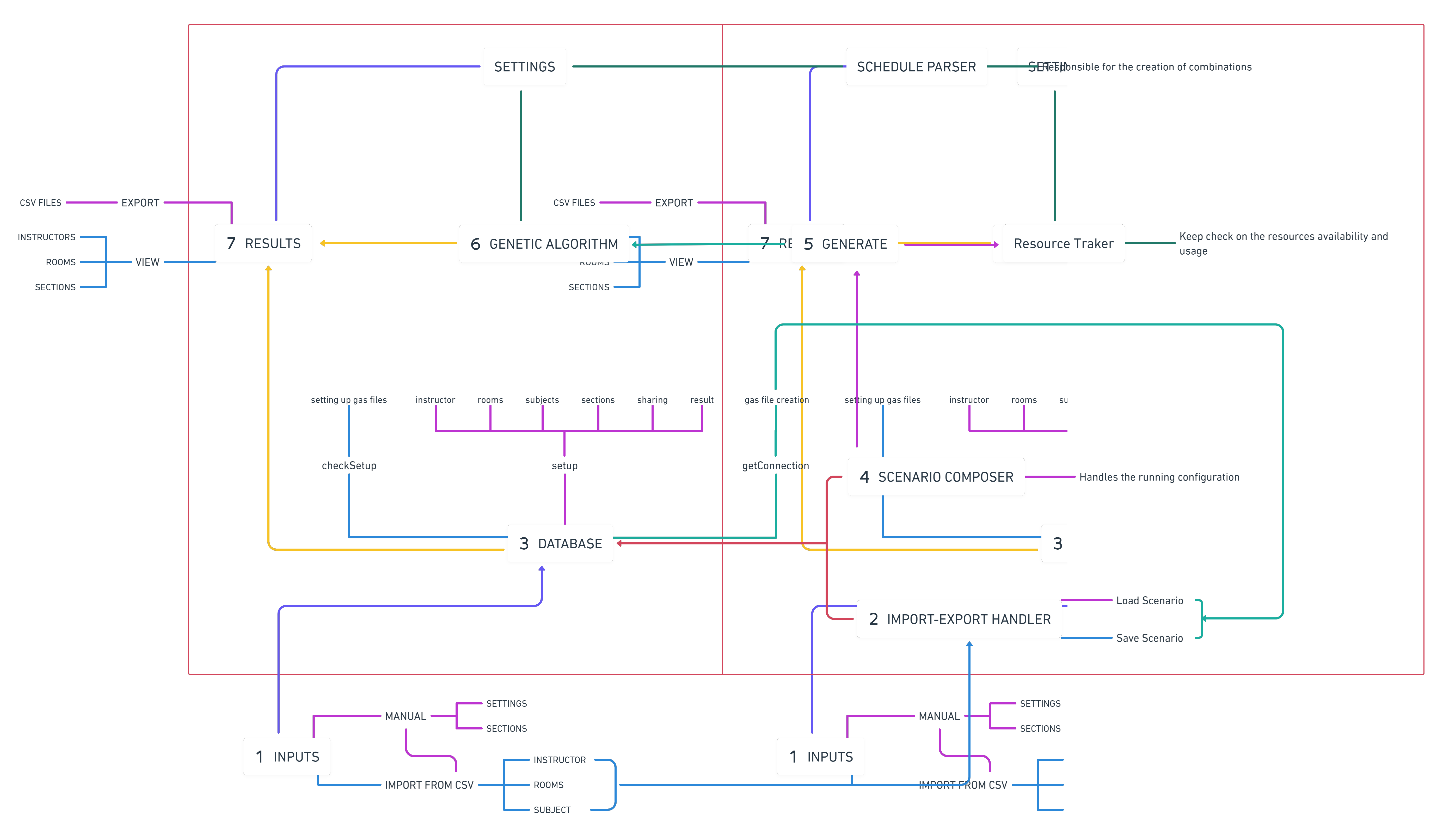
**UTILITY**

The large number of colleges and universities in India and the time-consuming manual timetable scheduling procedure, this system is beneficial where a strong framework is needed to analyse data to build an AI-based timetable scheduling. The Genetic Algorithm, a popular optimization tool in AI-based solutions, is employed by the authors to provide an effective method for generating AI-based timetables. The suggested technique generates a schedule that can be accessed by various classes and can take user inputs or import data from a csv file. For additional processing, the findings can be exported in csv format. The suggested mechanism has the potential to greatly cut down on the time and work needed to schedule timetables in educational institutions while also increasing the effectiveness of the procedure.

**SYSTEM ARCHITECTURE**

The system architecture of the proposed solution for the timetable problem is shown in Fig. X.x. The system's flow is clearly illustrated, and it includes descriptions of the Inputs, Import-Export Handler, Database, Scenario Composer, Generate, Genetic Algorithm, and Results segments.

The inputs segment enables data entry both directly from a csv file and manually, giving the user access to additional capabilities like applying custom settings and adding sections. Both imported and manually entered data are sent to the import-export handler, which loads the scenario—generally speaking, the settings for a previously saved model—or saves the scenario for the freshly entered data that was manually. Additionally, the system solicits data from or stores it in the database segment, which stores instructors, sections, subjects, sharing, rooms, and outcomes in accordance with the input method. The configuration to pass the data through the system to build the Timetable is then handled by the Scenario Composer segment. The two functions of the Generate segment are to use the data and another segment, a Genetic Algorithm, to produce a plausible timeline. Another is that it sends data to Resource Tracker, which monitors the resources' availability and consumption. The system displays the final answer in the Result segment, which can be viewed from different perspectives, such as from the perspective of the instructor, based on room allotment, or according to a timetable divided into different portions. Also, it may export the results as a CSV file to store them all.



**RELATED WORK**

The University Timetabling Problem (UTP) is a challenging combinatorial optimization problem that involves scheduling courses, classrooms, and instructors to maximize resource utilization and minimize conflicts. Due to its complexity, researchers have explored various optimization techniques, including Genetic Algorithm (GA), to solve the UTP efficiently. This document presents a review of relevant studies on GA and its application for solving the UTP.

[[1]](#r1) shows the study of GA and its application in solving optimization problems, including the UTP. They demonstrated the effectiveness of GA in generating optimal solutions and reducing computational time. Another study [[2]](#r2) implemented GA in an academic scheduling system and showed that GA outperforms traditional approaches in terms of quality of solutions and efficiency.

In 2003, [[3]](#r3) demonstrated the use of GA to solve the graph coloring problem and the effectiveness of GA in solving combinatorial problems. After this a comprehensive review of GA concepts and application in various optimization problems, including scheduling problems was researched [[4]](#r4). Scheduling problems and genetic algorithms together we further explored when [[5]](#r5) experimented with GA for timetabling and showed that GA is a robust and efficient solution in solving the UTP. Applications of GA were explored when [[6]](#r6) (2006) conducted a study on improving GA for optimization problems and demonstrated the effectiveness of the proposed approach in reducing computational time and increasing the quality of solutions.

In recent years, many researchers have used GA to solve the UTP. [[7]](#r7) formulated the UTP as an integer programming problem and compared the results with other optimization techniques, including GA. [[8]](#r8) (2016) proposed a University Timetable Generator using Tabu Search and compared the results with GA. [[9]](#r9) used GA to solve the UTP and demonstrated the effectiveness of the approach in generating optimal solutions. [[10]](#r10) proposed an automatic timetable generation system using GA and demonstrated the effectiveness of the approach in reducing the computational time and generating high-quality solutions. [[11]](#r11) (2021) proposed a dynamic chromosome approach using GA to solve the UTP, and their results showed that the approach generates high-quality solutions. [[12]](#r12) (2019) proposed a GA-based approach for solving timetable scheduling problems and demonstrated the effectiveness of the approach in generating optimal solutions.

[[13]](#r13) reviewed the state-of-the-art techniques for automated university timetabling and discussed the potential of GA in solving the UTP. [[14]](#r14) provided a comprehensive review of university timetabling and discussed the applications of various optimization techniques, including GA.

In summary, GA is a popular optimization technique for solving the UTP. The studies reviewed in this document demonstrate the effectiveness of GA in generating optimal solutions, reducing computational time, and improving resource utilization. While GA is not the only optimization technique used to solve the UTP, it has shown promising results and potential for future research.

**DESCRIPTION AND ALGORITHMS**

**INSTRUCTOR**

The code starts by importing the PyQt5 library, which is used to create a desktop GUI application in python.

The code consists of two classes, Instructor and Tree.

The instructor class is used to create a new dialog window that allows the user to create or edit the information of an instructor. The information will include name, hours, and schedule of the instructor. The \_\_init\_\_ method is used to create a dialog box and it also sets the UI elements. When the user will click on the “Finish” button, the finish method will be called. This will perform some validation on the input and then insert or update the information in the database.

The tree class is used to display a list of all the instructors in a QTreeView widget, with columns for each place of information. This class consists of a toggleAvailibilty() method which will be called when the user changes the availability of the instructor. The class also has a display() method that is used to get the data from the database and full the QTreeView with data.

Both the classes use a database module which is imported as db to interact with the database. It retrieves or updates the data on the database using the getConnection() method which is defined in the imported database module, so if any update is made it will be directly made to the “instructors” table in the gas.db database.

**ALGORITHM : INSTRUCTOR**

Import 🡪 QtWidgets, QtGui 🡪 PyQt5

Import 🡪 Database as db, Timetable 🡪 components

import 🡪 Instructor as Parent 🡪 py\_ui

import 🡪 json

Class 🡨 Instructor

Function 🡨 \_init\_ (self, id)

self.id 🡨 id

self.dialog 🡨 dialog 🡨 QtWidgets.QDialog()

self.parent 🡨 parent 🡨 Parent.Ui\_Dialog()

parent 🡨 setupUi (dialog)

IF id

self.fillForm()

ELSE

# Create a new instance of timetable

self.table 🡨 Timetable.Timetable(parent.tableSchedule)

parent.btnFinish.clicked.connect(self.finish)

parent.btnCancel.clicked.connect(self.dialog.close)

dialog.exec\_()

def function 🡨 fillForm(self)

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('SELECT name, hours, schedule FROM instructors WHERE id = ?', [self.id])

result 🡨 cursor.fetchone()

conn.close()

self.parent.lineEditName.setText(str(result[0]))

self.parent.lineEditHours.setText(str(result[1]))

# Generate timetable from custom schedule

self.table 🡨 Timetable.Timetable(self.parent.tableSchedule, json.loads(result[2]))

def function 🡨 finish(self)

# Verification of input

IF not self.parent.lineEditName.text():

return 🡨 False

name 🡨 self.parent.lineEditName.text()

TRY

hours 🡨 int(self.parent.lineEditHours.text())

if hours <= 0 or hours > 100 or hours % .5 != 0:

return 🡨 False

EXCEPT

return 🡨 False

data 🡨 [name, hours, json.dumps(self.table.getData()), self.id]

IF not self.id

data.pop()

self.insertInstructor(data)

self.dialog.close()

def function 🡨 insertInstructor(data):

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

IF len(data) > 3

cursor.execute('UPDATE instructors SET name = ?, hours = ?, schedule = ? WHERE id=?', data)

ELSE

cursor.execute('INSERT INTO instructors (name, hours, schedule) VALUES (?, ?, ?)', data)

conn.commit()

conn.close()

return True

class 🡨 Tree

def function 🡨 \_\_init\_\_(self, tree)

self.tree 🡨 tree

self.model 🡨 model 🡨 QtGui.QStandardItemModel()

model.setHorizontalHeaderLabels(['ID', 'Available', 'Name', 'Hours', 'Operation'])

tree.setModel(model)

tree.setColumnHidden(0, True)

model.itemChanged.connect(lambda item: self.toggleAvailability(item))

self.display()

def function 🡨 toggleAvailability(self, item)

# Get ID of toggled instructor

id 🡨 self.model.data(self.model.index(item.row(), 0))

newValue 🡨 1 if item.checkState() == 2 else 0

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('UPDATE instructors SET active = ? WHERE id = ?', [newValue, id])

conn.commit()

conn.close()

def function 🡨 display(self)

# Clear model

self.model.removeRows(0, self.model.rowCount())

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('SELECT id, active, hours, name FROM instructors')

result 🡨 cursor.fetchall()

conn.close()

for instr in result:

# ID Item

id 🡨QtGui.QStandardItem(str(instr[0]))

id.setEditable(False)

# Availability Item

availability 🡨 QtGui.QStandardItem()

availability.setCheckable(True)

availability.setCheckState(2 if instr[1] == 1 else 0)

availability.setEditable(False)

# Hours Item

hours 🡨 QtGui.QStandardItem(str(instr[2]))

hours.setEditable(False)

# Name Item

name 🡨 QtGui.QStandardItem(instr[3])

name.setEditable(False)

# Edit Item / Container for operation buttons

edit 🡨 QtGui.QStandardItem()

edit.setEditable(False)

# Append items to model

self.model.appendRow([id, availability, name, hours, edit])

# Create a widget group for edit and delete buttons

frameEdit 🡨 QtWidgets.QFrame()

btnEdit 🡨 QtWidgets.QPushButton('Edit', frameEdit)

btnEdit.clicked.connect(lambda state, id=instr[0]: self.edit(id))

btnDelete 🡨 QtWidgets.QPushButton('Delete', frameEdit)

btnDelete.clicked.connect(lambda state, id=instr[0]: self.delete(id))

frameLayout 🡨 QtWidgets.QHBoxLayout(frameEdit)

frameLayout.setContentsMargins(0, 0, 0, 0)

frameLayout.addWidget(btnEdit)

frameLayout.addWidget(btnDelete)

# Append the widget group to edit item

self.tree.setIndexWidget(edit.index(), frameEdit)

def function 🡨 edit(self, id)

Instructor(id)

self.display()

def function 🡨 delete(self, id)

# Show confirm model

confirm 🡨 QtWidgets.QMessageBox()

confirm.setIcon(QtWidgets.QMessageBox.Warning)

confirm.setText('Are you sure you want to delete this entry?')

confirm.setWindowTitle('Confirm Delete')

confirm.setStandardButtons(QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No)

result 🡨 confirm.exec\_()

# 16384 == Confirm

if result ==16384:

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('DELETE FROM instructors WHERE id = ?', [id])

conn.commit()

conn.close()

self.display()

**ROOM**

The Room class has an id property which is about when it is created and can only be changed if the user enters it manually.The code then creates a replacement instance of dialog, parent, and tableSchedule from components. Next, this code connects btnFinish with self. finish (), btnCancel with self. dialog. close (), and lineEditName with self `startupUi' method in order that they are all called when their respective buttons are clicked on the custom dialog.The Room class has two methods, fillForm and finish. The finish method checks if the user has filled out their name and sort before calling insertRoom on itself, which inserts a replacement room into its database.

Next, we create a function called toggleAvailability () which can allow us to change the availability state of an item in our database based on its row number. The code has two functions: 1) display () - This function displays the model that was created within the constructor of this class. 2) toggleAvailability () - This function changes the supply state of an item in the model.

**ALGORITHM : ROOM**

From PyQt5 🡨 import 🡪 QtWidgets, QtGui

from components 🡨 import 🡪 Database as db, Timetable

from py\_ui 🡨 import 🡪 Room as Parent

import json

Class 🡨 Room:

Function 🡨 \_\_init\_\_(self, id):

self.id 🡨 id

# New instance of dialog

self.dialog 🡨 dialog 🡨 QtWidgets.QDialog()

# Initialize custom dialog

self.parent 🡨 parent 🡨 Parent.Ui\_Dialog()

# Add parent to custom dialog

parent.setupUi(dialog)

# Connect timetable widget with custom timetable model

IF id:

self.fillForm()

ELSE:

self.table = Timetable.Timetable(parent.tableSchedule)

parent.btnFinish.clicked.connect(self.finish)

parent.btnCancel.clicked.connect(self.dialog.close)

dialog.exec\_()

Function 🡨 fillForm(self):

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('SELECT name, schedule, type FROM rooms WHERE id = ?', [self.id])

result 🡨 cursor.fetchone()

conn.close()

self.parent.lineEditName.setText(str(result[0]))

self.table 🡨 Timetable.Timetable(self.parent.tableSchedule, json.loads(result[1]))

IF result[2] EQUALS 'lec':

self.parent.radioLec.setChecked(True)

ELSE:

self.parent.radioLab.setChecked(True)

Function 🡨 finish(self):

IF NOT self.parent.lineEditName.text():

THEN Return 🡨 False

name 🡨 self.parent.lineEditName.text()

type 🡨 'lec' if self.parent.radioLec.isChecked() else 'lab'

data 🡨 [name, json.dumps(self.table.getData()), type, self.id]

IF NOT self.id:

THEN data.pop()

self.insertRoom(data)

self.dialog.close()

@staticmethod

Function 🡨 insertRoom(data):

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

IF len(data) > 3:

cursor.execute('UPDATE rooms SET name = ?, schedule = ?, type = ? WHERE id = ?', data)

ELSE:

cursor.execute('INSERT INTO rooms (name, schedule, type) VALUES (?, ?, ?)', data)

conn.commit()

conn.close()

Class 🡨 Tree:

Function 🡨 \_\_init\_\_(self, tree):

self.tree 🡨 tree

self.model 🡨 model = QtGui.QStandardItemModel()

model.setHorizontalHeaderLabels(['ID', 'Available', 'Name', 'Operation'])

tree.setModel(model)

tree.setColumnHidden(0, True)

model.itemChanged.connect(lambda item: self.toggleAvailability(item))

self.display()

Function 🡨 toggleAvailability(self, item):

id 🡨 self.model.data(self.model.index(item.row(), 0))

newValue 🡨 1 IF item.checkState() EQUALS 2 ELSE 0

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('UPDATE rooms SET active = ? WHERE id = ?', [newValue, id])

conn.commit()

conn.close()

Function 🡨 display(self):

self.model.removeRows(0, self.model.rowCount())

conn = db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('SELECT id, active, name FROM rooms')

result 🡨 cursor.fetchall()

conn.close()

FOR entry IN result:

id 🡨 QtGui.QStandardItem(str(entry[0]))

id.setEditable(False)

availability 🡨 QtGui.QStandardItem()

availability.setCheckable(True)

availability.setCheckState(2 if entry[1] == 1 else 0)

availability.setEditable(False)

name 🡨 QtGui.QStandardItem(entry[2])

name.setEditable(False)

edit 🡨 QtGui.QStandardItem()

edit.setEditable(False)

self.model.appendRow([id, availability, name, edit])

frameEdit 🡨 QtWidgets.QFrame()

btnEdit 🡨 QtWidgets.QPushButton('Edit', frameEdit)

btnEdit.clicked.connect(lambda state, id=entry[0]: self.edit(id))

btnDelete = QtWidgets.QPushButton('Delete', frameEdit)

btnDelete.clicked.connect(lambda state, id=entry[0]: self.delete(id))

frameLayout = QtWidgets.QHBoxLayout(frameEdit)

frameLayout.setContentsMargins(0, 0, 0, 0)

frameLayout.addWidget(btnEdit)

frameLayout.addWidget(btnDelete)

self.tree.setIndexWidget(edit.index(), frameEdit)

END FOR LOOP

Function 🡨 edit(self, id):

Room(id)

self.display()

Function 🡨 delete(self, id):

confirm 🡨 QtWidgets.QMessageBox()

confirm.setIcon(QtWidgets.QMessageBox.Warning)

confirm.setText('Are you sure you want to delete this entry?')

confirm.setWindowTitle('Confirm Delete')

confirm.setStandardButtons(QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No)

result 🡨 confirm.exec\_()

IF result EQUALS 16384:

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('DELETE FROM rooms WHERE id = ?', [id])

conn.commit()

conn.close()

self.display()

**SECTION**

This script is using the PyQt5 library to create a graphical user interface (GUI) for a program. The script imports several classes and modules, including the QtWidgets and QtGui modules from the PyQt5 library, the Share, Database (renamed as db), and Timetable classes from the containers and components modules, and the Section class from the py\_ui module.

The Section class is defined, which is used to create a dialog window. Inside the class, there are several methods including the \_init\_, setupSubjects, finish, fillForm .

The \_init\_ method is called when an instance of the class is created. It takes a single argument, id, which is used to determine if the dialog is being used to create a new section or to edit an existing one. It initializes several instance variables including shareId, removeShareId, dialog, parent,table, tree, model. Dialog is the Qdialog window, parent is an instance of the Ui\_Dialog class defined in the py\_ui.Section module, and table is an instance of the Timetable.Timetable class that is used to create the schedule table. It also call the fillForm() and setupSubjects method

The setupSubjects method is used to set up the subjects tree view in the dialog window. It creates an instance of a QstandardItemModel and sets the horizontal header labels for the tree view. It then hides the first and fifth columns of the tree view. It then populates the tree view with data from the database by executing a SELECT statement to get all the subjects and their details, and then adding them to the tree view as items.

The finish method is called when the user clicks the ‘Finish’ button in the dialog window and it executes the necessary operations to process the changes.

**ALGORITHM : SECTION**

from PyQt5 🡨 import 🡪 QtWidgets, QtGui

from containers 🡨 import 🡪 Share

from components 🡨 import 🡪 Database as db, Timetable

from py\_ui 🡨 import 🡪 Section as Parent

import 🡪 json

Class 🡨 Section:

Function 🡨 \_\_init\_\_(self, id):

self.id 🡨 id

# Array of share IDs to be finalized

self.shareId 🡨 []

# Array of share IDs to be removed

self.removeShareId 🡨 []

self.dialog = dialog 🡨 QtWidgets.QDialog()

# From the qt\_ui generated UI

self.parent = parent 🡨 Parent.Ui\_Dialog()

parent.setupUi(dialog)

IF id:

THEN self.fillForm() ELSE: self.table 🡨 Timetable.Timetable(parent.tableSchedule)

# Create new instance of timetable

self.setupSubjects()

parent.btnFinish.clicked.connect(self.finish)

parent.btnCancel.clicked.connect(self.dialog.close)

dialog.exec\_()

Function 🡨 setupSubjects(self):

# Setup subjects tree view

self.tree 🡨 tree 🡨 self.parent.treeSubjects

self.model 🡨 model 🡨 QtGui.QStandardItemModel()

model.setHorizontalHeaderLabels(['ID', 'Available', 'Shared', 'Subject Code', 'Subject Name', 'Share ID'])

tree.setModel(model)

tree.setColumnHidden(0, True)

tree.setColumnHidden(5, True)

# Populate tree with values

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

# Get subjects for listing

cursor.execute('SELECT id, name, code FROM subjects')

subjects 🡨 cursor.fetchall()

# Subjects that the current section have

currentSubjects 🡨 []

# Subjects that are shared to the current section

# [(sharing\_id, subject\_id, sections [str of list - load using json.loads])]

sharedSubjects 🡨 []

if self.id:

cursor.execute('SELECT subjects FROM sections WHERE id = ?', [self.id])

# Convert result into list of int

currentSubjects 🡨 list(map(lambda id: int(id), json.loads(cursor.fetchall()[0][0])))

# Get active sharing by subject

FOR id IN currentSubjects:

cursor.execute('SELECT id, subjectId, sections FROM sharings WHERE subjectId = ? AND final = 1', [id])

sharedSubjects.append(cursor.fetchone())

END FOR LOOP

sharedSubjects 🡨 [sharing for sharing in sharedSubjects if sharing]

# Get section names

# {id: name}

sectionNames = []

cursor.execute('SELECT id, name FROM sections WHERE active = 1')

sectionNames 🡨 dict(cursor.fetchall())

conn.close()

FOR subject IN subjects:

subjectId 🡨 QtGui.QStandardItem(str(subject[0]))

subjectId.setEditable(False)

availability 🡨 QtGui.QStandardItem()

availability.setCheckable(True)

availability.setEditable(False)

availability.setCheckState(2 if subject[0] in currentSubjects else 0)

shared 🡨 QtGui.QStandardItem('')

shared.setCheckable(True)

shared.setEditable(False)

shareId 🡨 QtGui.QStandardItem()

shareId.setEditable(False)

FOR sharing IN sharedSubjects:

IF sharing[1] != subject[0]:

THEN continue

sectionList 🡨 list(map(lambda id: int(id), json.loads(sharing[2])))

IF self.id NOT IN sectionList:

THEN continue

sectionList.remove(self.id)

sectionList 🡨 ', '.join(list(map(lambda id: sectionNames[id], sectionList)))

shared.setText(sectionList)

shared.setCheckState(2)

shareId.setText(str(sharing[0]))

FOR LOOP END

code 🡨 QtGui.QStandardItem(subject[2])

code.setEditable(False)

name 🡨 QtGui.QStandardItem(subject[1])

name.setEditable(False)

model.appendRow([subjectId, availability, shared, code, name, shareId])

FOR LOOP END

model.itemChanged.connect(lambda item: self.toggleSharing(item))

Function 🡨 toggleSharing(self, item):

IF item.column() EQUALS 2:

subjectId 🡨 self.model.data(self.model.index(item.row(), 0))

shareToggle 🡨 self.model.item(item.row(), 2).checkState()

IF shareToggle EQUALS 2 AND NOT self.model.item(item.row(), 2).text():

shareData 🡨 Share.Share(subjectId, self.id).getShareData()

IF NOT shareData[0]:

self.model.item(item.row(), 2).setCheckState(0)

Return 🡨 False

shareId 🡨 shareData[0]

self.shareId.append(shareId)

self.model.item(item.row(), 5).setText(str(shareId))

self.model.item(item.row(), 2).setText(shareData[1])

self.model.item(item.row(), 1).setCheckState(2)

ELSE IF shareToggle EQUALS 0 AND self.model.item(item.row(), 2).text():

IF int(self.model.item(item.row(), 5).text()) in self.shareId:

self.shareId.remove(int(self.model.item(item.row(), 5).text()))

ELSE:

self.removeShareId.append(int(self.model.item(item.row(), 5).text()))

self.model.item(item.row(), 5).setText('')

self.model.item(item.row(), 2).setText('')

ELSE IF item.column() EQUALS 1:

IF self.model.item(item.row(), 1).checkState() EQUALS 0 AND self.model.item(item.row(), 5).text():

THEN self.model.item(item.row(), 2).setCheckState(0)

Function 🡨 fillForm(self):

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('SELECT name, schedule, stay FROM sections WHERE id = ?', [self.id])

result 🡨 cursor.fetchone()

conn.close()

self.parent.lineEditName.setText(str(result[0]))

self.parent.checkStay.setChecked(result[2])

self.table 🡨 Timetable.Timetable(self.parent.tableSchedule, json.loads(result[1]))

Function 🡨 finish(self):

IF NOT self.parent.lineEditName.text():

THEN Return 🡨 False

name 🡨 self.parent.lineEditName.text()

stay 🡨 1 IF self.parent.checkStay.isChecked() ELSE 0

schedule 🡨 json.dumps(self.table.getData())

subjects 🡨 []

FOR row in range(self.model.rowCount()):

IF self.model.item(row, 1).checkState() EQUALS 2:

subjects.append(self.model.item(row, 0).text())

subjects 🡨 json.dumps(subjects)

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

IF self.removeShareId:

FOR id IN self.removeShareId:

cursor.execute('SELECT sections FROM sharings WHERE id = ?', [id])

result 🡨 list(map(int, json.loads(cursor.fetchone()[0])))

IF len(result) > 2:

result.remove(self.id)

cursor.execute('UPDATE sharings SET sections = ? WHERE id = ?', [json.dumps(result), id])

ELSE cursor.execute('UPDATE sharings SET final = 0 WHERE id = ?', [id])

FOR LOOP END

IF self.shareId:

FOR id IN self.shareId:

cursor.execute('UPDATE sharings SET final = 1 WHERE id = ?', [id])

IF self.id:

cursor.execute('UPDATE sections SET name = ?, schedule = ?, subjects = ?, stay = ? WHERE id = ?',

[name, schedule, subjects, stay, self.id])

ELSE:

cursor.execute('INSERT INTO sections (name, schedule, subjects, stay) VALUES (?, ?, ?, ?)',

[name, schedule, subjects, stay])

conn.commit()

conn.close()

self.dialog.close()

Class 🡨 Tree:

Function 🡨 \_\_init\_\_(self, tree):

self.tree 🡨 tree

self.model 🡨 model = QtGui.QStandardItemModel()

model.setHorizontalHeaderLabels(['ID', 'Available', 'Name', 'Stay in Room', 'Operation'])

tree.setModel(model)

tree.setColumnHidden(0, True)

model.itemChanged.connect(lambda item: self.toggleAvailability(item))

self.display()

Function 🡨 toggleAvailability(self, item):

id 🡨 self.model.data(self.model.index(item.row(), 0))

newValue 🡨 1 if item.checkState() == 2 else 0

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('UPDATE sections SET active = ? WHERE id = ?', [newValue, id])

conn.commit()

conn.close()

Function 🡨 display (self):

self.model.removeRows(0, self.model.rowCount())

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('SELECT id, active, name, stay FROM sections')

result 🡨 cursor.fetchall()

conn.close()

FOR instr IN result:

id 🡨 QtGui.QStandardItem(str(instr[0]))

id.setEditable(False)

availability 🡨 QtGui.QStandardItem()

availability.setCheckable(True)

availability.setCheckState(2 IF instr[1] EQUALS 1 ELSE 0)

availability.setEditable(False)

name 🡨 QtGui.QStandardItem(instr[2])

stay 🡨 QtGui.QStandardItem('TRUE' if instr[3] == 1 else 'FALSE')

stay.setEditable(False)

name.setEditable(False)

edit 🡨 QtGui.QStandardItem()

edit.setEditable(False)

self.model.appendRow([id, availability, name, stay, edit])

frameEdit 🡨 QtWidgets.QFrame()

btnEdit 🡨 QtWidgets.QPushButton('Edit', frameEdit)

btnEdit.clicked.connect(lambda state, id=instr[0]: self.edit(id))

btnDelete 🡨 QtWidgets.QPushButton('Delete', frameEdit)

btnDelete.clicked.connect(lambda state, id=instr[0]: self.delete(id))

frameLayout 🡨 QtWidgets.QHBoxLayout(frameEdit)

frameLayout.setContentsMargins(0, 0, 0, 0)

frameLayout.addWidget(btnEdit)

frameLayout.addWidget(btnDelete)

self.tree.setIndexWidget(edit.index(), frameEdit)

Function 🡨 edit (self, id):

Section(id)

self.display()

Function 🡨 delete (self, id):

confirm 🡨 QtWidgets.QMessageBox()

confirm.setIcon(QtWidgets.QMessageBox.Warning)

confirm.setText('Are you sure you want to delete this entry?')

confirm.setWindowTitle('Confirm Delete')

confirm.setStandardButtons(QtWidgets.QMessageBox.Yes | QtWidgets.QMessageBox.No)

result 🡨 confirm.exec\_()

IF result EQUALS 16384:

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('DELETE FROM sections WHERE id = ?', [id])

conn.commit()

conn.close()

self.display()

**SETTINGS**

The code starts by declaring a category called Settings. The settings are stored within the file “settings.json”. so as to read from this file, it’s opened with open(‘settings.json’) as json\_file: self.settings = json.load(json\_file) The code then uses the load() method of the JSON library to read altogether of the settings from this file and store them into an instance variable called “self.” This instance variable is then used throughout the program for various purposes like setting values or retrieving values that have been set previously: def getSetting(self, key): return self.settings[key] The code will parse the settings.json file and return a dictionary of all settings. The code above will set the worth for each key in the Settings class to its corresponding value from the settings.json file.

**ALGORITHM : SETTINGS**

Import 🡪 json

Class 🡨 Settings

Function 🡨 \_\_init\_\_ (self):

OPEN json\_file 🡨 (‘Settings.json’)

WITHIN json\_file:

Self.settings 🡨 json.load (json\_file)

END json\_file

Function getSetting(self,key):

Return 🡨 self.settings[key]

END CLASS

Function 🡨 getSetting(key):

OPEN json\_file 🡨 (‘Settings.json’)

WITHIN json\_file:

settings 🡨 json.load (json\_file)

END json\_file

Return 🡨 settings [key]

Function 🡨 getSettings():

OPEN json\_file 🡨 (‘Settings.json’)

WITHIN json\_file:

settings 🡨 json.load (json\_file)

END json\_file

Return 🡨 settings

Function 🡨 setSettings(key, value, secondKey=False):

Settings 🡨 **getSettings()**

IF (secondKey):

settings[key][secondKey] 🡨 value

ELSE:

settings[key] 🡨 value

END ELSE

OPEN json\_file in Write mode🡨 (‘Settings.json’)

WITHIN json\_file:

json\_file.write(json.dumps(settings))

END json\_file

**IMPORT EXPORT HANDLER**

The code starts by importing the required libraries. It then creates a function called getCSVFile which takes in two parameters: type and ilename. The primary parameter is the name of the file that will be created, and therefore the second parameter is a string representing where to save it on your computer. If you pass in None because the value for ilename, this suggests that you want to create a new CSV file with no filename yet assigned to it. the subsequent line of code creates an instance of QfileDialog which will return an open File Name dialog box if there are any files selected from your computer’s hard drive or other storage devices (such as USB drives). If not, this line returns False meaning that we cannot create our CSV file at this point because there are no files available on our computer’s hard drive or other storage devices (such as USB drives). Next comes another function called getOpenFileName which accepts three arguments: None, ‘’, and ‘CSV File (\*.csv)’. This function simply exposes a window where users can select one or more files from their computers’ hard drive or other storage devices (such as USB drives) so they can be imported into Python using csv library functions like reader().  
The code may be a function that returns True if the user has selected an existing file. The code may be a function that saves the content of the CSV file to disk.

**ALGORITHM : IMPORT EXPORT HANDLER**

Import 🡪 PyQt5 🡪 QtCore

Class 🡨 TableModel (QtCore.QAbstractableModel)

Function 🡨 \_\_init\_\_ (self,header,data):

Super() 🡨 \_init\_()

self.data 🡨 data

self.header 🡨 header

Function 🡨 data (self,index,role):

IF (index not valid)

THEN Return 🡨 QtCore.QVariant()

ELSE IF (role not equals QtCore.Qt.DisplayRole)

THEN return QtCore.QVariant()

END ELSE IF

Return 🡨 self.data[index.rows][index.column]

Function 🡨 rowCount (self,parent=None,\*args,\*\*kwargs):

Return 🡨 length of self.data

Function 🡨 columnCount (self, parent=None,\*args,\*\*kwargs):

Return 🡨 length of self.data[0]

Function 🡨 headerData (self, p\_int, Qt\_Orientation, role=None):

IF (Qt\_Orientation equals QtCore.Qt.Horizontal AND role equals QtCore.Q.DisplayRole):

THEN Return 🡨 QtCore.QVariant (self.header[0][p\_int]

END IF

Return 🡨 QtCore.QVariant()

Function 🡨 setData (self, index, value, role=None):

IF (index not valid)

THEN Return 🡨 False

END IF

self.data[index.row()][index.column()] 🡨 value

self.dataChanged.emit(index, index)

Return 🡨 True

**SCENARIO COMPOSER**

The primary method is used to create an instance of the ScenarioComposer class, which can be used later on in this program.The second method gets all instructors from the database who are currently active (i. e. , have hours scheduled). The code starts by creating an instance of the ScenarioComposer class named self. This line tells us that we’d like to use our connection object whenever we want to make queries against our database through self’s cursor variable later on in this program when we execute SQL statements like “SELECT \* FROM instructors” or “SELECT \* FROM schedules”. The primary method of this class, getInstructors (), retrieves all instructors within the database and returns them as a dictionary. It starts by getting an inventory of instructors, which is completed with self.jsonToList(). Then it gets an inventory of rooms, which is completed with self. ListToDictionary () then finally it gets a dictionary that has keys for each subject and values for each room’s schedule (which was created with self. JsonToList ()). The code returns an inventory of dictionaries with keys being the names of the rooms and values being their respective ids. The code above are going to be executed by the following: self. Cursor. Execute (‘SELECT id, name, hours, code, description, instructors, divisible, type FROM subjects’) this may execute the SQL statement “SELECT id, name, hours, code, description from subjects” which returns an inventory of dictionaries with each section’s information. The code then takes this data and creates two lists: one with ids and one with names. The code then takes these two lists and converts them to integers by using stringToInt () function. The code may be a function that fetches the sections data from the database and returns them in a dictionary. The code may be a function that takes each list of dictionaries and converts them into integers by using stringToInt () method.

**ALGORITHM : SCENARIO COMPOSER**

Class 🡨 ScenarioComposer

Function 🡨 \_\_init\_\_(self):

self.conn = **Database.getConnection()**

self.cursor 🡨 self.conn.cursor()

Function 🡨 getInstructors(self):

Self.cursor.execute(‘SELECT id, name, hours, schedule FROM instructors WHERE active = 1')

instructors 🡨 self.listToDictionary(self.cursor.fetchall())

instructors 🡨 self.jsonToList(instructors,2)

Return 🡨 instructors

Function 🡨 getRooms(self):

self.cursor.execute('SELECT id, name, type, schedule FROM rooms WHERE active = 1')

rooms 🡨 self.listToDictionary(self.cursor.fetchall())

rooms 🡨 self.jsonToList(rooms,2)

Return 🡨 rooms

Function 🡨 getSubjects(self):

self.cursor.execute('SELECT id, name, hours, code, description, instructors, divisible, type FROM subjects')

subjects 🡨 self.listToDictionary(self.cursor.fetchall())

subjects 🡨 self.jsonToList(subjects, 4)

subjects 🡨 self.stringToInt(subjects, 4)

Return 🡨 subjects

Function 🡨 getSections(self):

self.cursor.execute('SELECT id, name, schedule, subjects, stay FROM sections WHERE active = 1')

sections 🡨 self.listToDictionary(self.cursor.fetchall())

sections 🡨 self.jsonToList(sections, 1)

sections 🡨 self.jsonToList(sections, 2)

sections 🡨 self.stringToInt(sections, 2)

Return 🡨 sections

Function 🡨 getSharings(self):

self.cursor.execute('SELECT id, subjectId, sections FROM sharings WHERE final = 1')

sharings 🡨 self.listToDictionary(self.cursor.fetchall())

sharings 🡨 self.jsonToList(sharings, 1)

sharings 🡨 self.stringToInt(sharings, 1)

Return 🡨 sharings

Function 🡨 listToDictionary(self, toDict):

Return 🡨 {entry[0]: list(entry[1:]) for entry in toDict}

Function 🡨 jsonToList(self, dictionary, index):

FOR key, value in dictionary.items():

dictionary[key][index] 🡨 json.loads(value[index])

END FOR Loop

Return 🡨 dictionary

Function 🡨 stringToInt(self, dictionary, index):

FOR key, value in dictionary.items():

dictionary[key][index] 🡨 list(map(int, value[index]))

END FOR Loop

Return 🡨 dictionary

Function 🡨 closeConnection(self):

self.conn.commit()

self.conn.close()

Function 🡨 getScenarioData(self):

data = {

'instructors': self.getInstructors(),

'sharings': self.getSharings(),

'sections': self.getSections(),

'subjects': self.getSubjects(),

'rooms': self.getRooms()

}

**self.closeConnection()**

Return 🡨 data

**DATABASE**

A few functions for configuring and utilising the “gas.db” SQLite database are defined in this code.

The first function checkSetup() connects to the database and queries the sqlite\_master table to check if a table names “instructors” exists or not.

The next function is setup() which creates 6 tables with different columns if they don’t already exist. All the tables have a primary key which is “id” and an “active’ column which is a Boolean column with default value of 1, and a check constraint that the value must be either 0 or 1. The 6 tables that are made in this function include the “instructors” table, “rooms” table, “subjects” table, “sections” table, “sharing” table and “results’ table.

Last function of the code is a getConnection() function that returns the connection to the “gas.db” database so that the other parts of the code can interact with the database.

**ALGORITHM : DATABASE**

Import 🡪 sqlite3

Function 🡨 checkSetup():

conn 🡨 sqlite3.connect(‘Database fileName’)

cursor 🡨 conn.cursor()

cursor.execute("SELECT name FROM sqlite\_master WHERE type is 'table' AND  
 name is 'instructors'")  
   
 result 🡨 cursor.fetchone()

conn.close()

IF result is None

THEN Return 🡨 False

ELSE Return 🡨 True

Function 🡨 setup():

conn 🡨 sqlite3.connect(‘Database fileName’)

cursor 🡨 conn.cursor()

create\_instructions\_table 🡨 """ CREATE TABLE IF NOT EXISTS instructors (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

hours INTEGER NOT NULL,

schedule TEXT NOT NULL,

active BOOLEAN NOT NULL DEFAULT 1 CHECK (

active IN (0, 1))

); """

create\_rooms\_table 🡨 """CREATE TABLE IF NOT EXISTS rooms (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

type TEXT NOT NULL,

schedule TEXT NOT NULL,

active BOOLEAN NOT NULL DEFAULT 1 CHECK (

active IN (0, 1))

); """

create\_subjects\_table 🡨 """CREATE TABLE IF NOT EXISTS subjects (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

hours REAL NOT NULL,

code TEXT NOT NULL,

description TEXT NOT NULL,

instructors TEXT NOT NULL,

divisible BOOLEAN NOT NULL DEFAULT 1 CHECK (

divisible IN (0, 1)),

type TEXT NOT NULL

); """

create\_sections\_table 🡨 """CREATE TABLE IF NOT EXISTS sections (

id INTEGER PRIMARY KEY,

name TEXT NOT NULL,

schedule TEXT NOT NULL,

subjects TEXT NOT NULL,

active BOOLEAN NOT NULL DEFAULT 1 CHECK (

active IN (0, 1)),

stay BOOLEAN NOT NULL DEFAULT 0 CHECK (

active IN (0, 1))

);"""

create\_sharing\_table 🡨 """CREATE TABLE IF NOT EXISTS sharings (

id INTEGER PRIMARY KEY,

subjectId INTEGER NOT NULL,

sections TEXT NOT NULL,

final BOOLEAN NOT NULL DEFAULT 0 CHECK (

final IN (0, 1)

); """

create\_results\_table 🡨 """CREATE TABLE IF NOT EXISTS results (

id INTEGER PRIMARY KEY,

content BLOB NOT NULL,

timestamp DATETIME DEFAULT CURRENT\_TIMESTAMP

); """

cursor.execute(create\_instructors\_table)

cursor.execute(create\_rooms\_table)

cursor.execute(create\_subjects\_table)

cursor.execute(create\_sections\_table)

cursor.execute(create\_sharing\_table)

cursor.execute(create\_results\_table)

conn.commit()

conn.close()

Function 🡨 getConnction():

Return 🡨 sqlite3.connect(‘Database fileName’)

**RESOURCE TRACKER**

The psutil library is imported first in the code. The percentage of CPU use on a single core is given via the getCPUUsage() method. The getMemoryUsage() function returns the amount of memory used in megabytes and the percentage of total memory that is still available. Bytes are converted to megabytes using the byteToMegabyte() function, rounding up as appropriate. The code then used these routines to determine how much RAM is being used and what proportion of the entire RAM pool it represents (the MemoryPercentage). The entire amount of memory being used by the system will be returned by the function. The code will output the machine's memory utilisation rate.

**ALGORITHM : RESOURCE TRACKER**

Import 🡪 os

Import 🡪 psutil

Function 🡨 getCPUUsage():

Return 🡨 psutil.cpu\_percent(1)

Function 🡨 getMemoryUsage():

Return 🡨 [psutil.Process(os.getpid()).memory\_info()[0], psutil.virtual\_memory()[0]]

Function 🡨 getMemoryPercentage(memoryUsage):

Return 🡨 (memoryUsage[0] / memoryUsage[1]) \* 100) rounded upto 2 decimal places

Function 🡨 byteToMegabyte(byte):

Return 🡨 (byte / 1048576) rounded upto 2 decimal places

**SCHEDULE PARSER**

The code starts by importing the PyQt5 library, which is used to create a desktop GUI application in python.

The code defines two classes, ScheduleParser and ScheduleParserModel.

The ScheduleParser class is used to parse and display data in a table-like format. This class takes two arguments in its constructor, a table and data. The constructor initializes the table by setting its model to an instance of the ScheduleParserModel class, which is defined by the components.TableModel module. It also loads time slots from a json file, ‘timeslots.json’ and call the parseData method to populate the table with data.

The ScheduleParserModel class is a subclass of TableModel class, it defines the setData. The data method and is used to store the data that is displayed in the table. It takes two arguments in its constructor, the header and data of the table.

**ALGORITHM : SCHEDULE PARSER**

Import 🡪 PyQt5 🡪 QtCore, QtWidgets, QtGui

Import 🡪 json

Class 🡨 ScheduleParser

Function 🡨 \_\_init\_\_(self, table, data):

self.table 🡨 table

header 🡨 [['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']]

OPEN json\_file 🡨 (‘timeslots.json’)

WITHIN json\_file:

self.timeslots 🡨 timeslots 🡨 json.load(json\_file)['timeslots']

END json\_file

self.settings 🡨 settings 🡨 **getSettings()**

header 🡨 *append* 🡨 timeslots[settings['starting\_time']:settings['ending\_time'] + 1]

temporaryData 🡨 []

FOR i in range (0 and settings['ending\_time'] + 1 - settings['starting\_time']):

temporaryData 🡨 append 🡨 ['', '', '', '', '', '']

END FOR loop

self.model 🡨 ScheduleParserModel(header, temporaryData)

table.setModel(self.model)

table.setFocusPolicy(QtCore.Qt.NoFocus)

table.setSelectionMode(QtWidgets.QAbstractItemView.NoSelection)

table.verticalHeader().setSectionResizeMode(QtWidgets.QHeaderView.Fixed)

self.**parseData(data)**

Function 🡨 parseData(self, data):

view 🡨 self.table

model 🡨 self.model

FOR entry in data :

entry['color'] = **colorGenerator()**

FOR instance in entry['instances']:

index = model.index(instance[1], instance[0])

view.setSpan(instance[1], instance[0], instance[2] - instance[1], 1)

item 🡨 QtGui.QStandardItem(entry['text'])

item.setBackground(QtGui.QBrush(QtGui.QColor(\*entry['color'])))

item.setForeground(QtGui.QBrush(QtGui.QColor(\*Utilities.textColor  
 (entry['color']))))

model.setData(index, item)

Function 🡨 subjectGenerator(self):

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

END CLASS

Class 🡨 ScheduleParserModel(TableModel.TableModel)

Function 🡨 \_\_init\_\_(self, header, data):

super().\_\_init\_\_(header, data)

Function 🡨 setData(self, index, value, role=None):

IF (index not valid)

THEN Return 🡨 False

ELSE IF (role 🡨 None)

THEN self.data[index.row()][index.column()] 🡨 value

END ELSE IF

self.dataChanged.emit(index, index)

Return 🡨 True

Function 🡨 data(self, index, role):

IF (index not valid or not self.data[index.row()][index.column()])

THEN Return 🡨 QtCore.QVariant()

ELSE IF (role equals QtCore.Qt.TextAlignmentRole)

THEN Return 🡨 QtCore.Qt.AlignCenter

ELSE IF (role equals QtCore.Qt.BackgroundRole)

THEN Return 🡨 self.data[index.row()][index.column()].background()

ELSE IF (role equals QtCore.Qt.ForegroundRole)

THEN Return 🡨 self.data[index.row()][index.column()].foreground()

ELSE IF (role not equals QtCore.Qt.DisplayRole)

THEN Return 🡨 QtCore.QVariant()

END ELSE IF

Return 🡨 self.data[index.row()][index.column()].text()

**GENERATE**

Code gives an overall description of how a class is created. It starts with the declaration of the category. The \_\_init\_\_ method is where all the initialization is done. In this case, it'end with setting the self.totalresource to an empty dictionary with two keys: 'cpu' and 'memory'. Next, we declare the self.tick as 0 and set the self.data to be a dictionary with three keys: results, rooms, instructors, sections, sharings (which has no key). The topchromosomes list can hold all of our chromosomes at the end of each generation so that they can be sorted into their respective groups based on fitness value.

It is creating a replacement instance of the Activity class. This is done with the self = MyActivity() line. The next two lines create instances of the Label classes, one for CPU usage and one for memory usage. The first line in each label's setText method calls their parent's setText method, which then changes their text to "CPU Usage: Stopped" or "Memory Usage: Stopped", respectively. Finally, at the top of this code block, we calculate what proportion of resource is being used on our device by taking the mean value from all three resources. The code may be a part of the code which will be executed when the program is complete. The code may be a part of the code which will be executed when the program is complete. The code may be a part of the code which will be executed when the program is complete.

**ALGORITHM : GENERATE**

From PyQt5 🡨 Import 🡪 QtCore, QtWidgets

From components 🡨 Import 🡪 Database as db, ResourceTracker, ScheduleParser, ScenarioComposer, GeneticAlgorithm

From py\_ui 🡨 Import 🡪 Generate as Parent

From sqlite3 🡨 Import 🡪 Binary

From numpy 🡨 Import 🡪 mean

Import 🡪 pickle

Import 🡪 copy

Class 🡨 Generate

Function 🡨 \_\_init\_\_(self):

self.totalResource 🡨 {

'cpu': [],

'memory': []

}

self.tick 🡨 0

self.data 🡨 {

'results': [],

'rooms': [],

'instructors': [],

'sections': [],

'sharings': [],

'subjects': []

}

self.topChromosomes 🡨 []

self.meta 🡨 []

self.preview 🡨 True

self.sectionKeys 🡨 []

composer 🡨 ScenarioComposer.ScenarioComposer()

composer 🡨 composer.getScenarioData()

self.data.update(composer)

self.dialog 🡨 dialog 🡨 QtWidgets.QDialog(parent=None)

# Initialize custom dialog

self.parent 🡨 parent 🡨 Parent.Ui\_Dialog()

# Add parent to custom dialog

parent.setupUi(dialog)

dialog.setWindowFlags(QtCore.Qt.Window | QtCore.Qt.WindowTitleHint | QtCore.Qt.CustomizeWindowHint)

self.time 🡨 QtCore.QTime(0, 0)

self.timer 🡨 QtCore.QTimer()

self.timer.timeout.connect(self.updateTime)

self.timer.start(1000)

self.running 🡨 True

self.table 🡨 parent.tableSchedule

parent.btnPause.clicked.connect(self.togglePause)

parent.btnStop.clicked.connect(self.stopOperation)

parent.chkPreview.clicked.connect(self.togglePreview)

parent.cmbSection.clear()

FOR section, details IN self.data['sections'].items():

self.sectionKeys.append(section)

parent.cmbSection.addItem(details[0])

END FOR loop

parent.cmbSection.currentIndexChanged.connect(self.changePreview)

self.startWorkers()

dialog.exec\_()

Function 🡨 togglePreview(self, state):

self.preview 🡨 not state

Function 🡨 togglePause(self):

self.toggleState()

self.parent.btnPause.setText('Pause Generation' if self.running else 'Resume Generation')

Function 🡨 toggleState(self, state=None):

self.running 🡨 (not self.running) if state is None else state

self.resourceWorker.running 🡨 self.running

self.geneticAlgorithm.running 🡨 self.running

Function 🡨 startWorkers(self):

self.resourceWorker 🡨 ResourceTrackerWorker()

self.resourceWorker.signal.connect(self.updateResource)

self.resourceWorker.start()

self.geneticAlgorithm = GeneticAlgorithm.GeneticAlgorithm(self.data)

self.geneticAlgorithm.statusSignal.connect(self.updateStatus)

self.geneticAlgorithm.detailsSignal.connect(self.updateDetails)

self.geneticAlgorithm.dataSignal.connect(self.updateView)

self.geneticAlgorithm.operationSignal.connect(self.updateOperation)

self.geneticAlgorithm.start()

Function 🡨 updateStatus(self, status):

self.parent.lblStatus.setText('Status: {}'.format(status))

Function 🡨 updateDetails(self, details):

self.parent.boxGen.setTitle('Generation #{}'.format(details[0]))

self.parent.lblPopulation.setText('Population: {}'.format(details[1]))

self.parent.lblMutation.setText('Mutation Rate: {}%'.format(details[2]))

self.parent.lblFitness.setText('Average Fitness: {}%'.format(details[3]))

self.parent.lblPreviousFitness.setText('Previous Average Fitness: {}%'.format(details[4]))

self.parent.lblHighestFitness.setText('Highest Fitness: {}%'.format(details[5]))

self.parent.lblLowestFitness.setText('Lowest Fitness: {}%'.format(details[6]))

Function 🡨 updateView(self, chromosomes):

chromosomes.reverse()

self.topChromosomes 🡨 copy.deepcopy(chromosomes)

self.changePreview(self.parent.cmbSection.currentIndex())

Function 🡨 changePreview(self, index):

data 🡨 []

IF NOT len(self.topChromosomes) OR NOT self.preview:

Return 🡨 False

sections 🡨 self.topChromosomes[0][0].data['sections']

rawData 🡨 self.data

subjects 🡨 sections[self.sectionKeys[index]]['details']

FOR subject, details IN subjects.items():

IF NOT len(details):

Continue

END IF

instructor 🡨 '' if not details[1] else rawData['instructors'][details[1]][0]

data.append({'color': None, 'text': '{} \n {} \n

{}'.format(rawData['subjects'][subject][0], rawData['rooms'][details[0]][0], instructor),'instances': [[day, details[3], details[3] + details[4]] for day in details[2]]})

END FOR LOOP

self.loadTable(data)

Function 🡨 loadTable(self, data=[]):

self.table.reset()

self.table.clearSpans()

ScheduleParser.ScheduleParser(self.table, data)

Function 🡨 updateOperation(self, type):

IF type == 1:

self.stopOperation()

Function 🡨 updateTime(self):

self.time 🡨 self.time.addSecs(1)

self.parent.lblTime.setText('Elapsed Time: {}'.format(self.time.toString('hh:mm:ss')))

Function 🡨 stopOperation(self):

self.toggleState(False)

self.resourceWorker.terminate()

self.resourceWorker.runThread 🡨 False

self.geneticAlgorithm.terminate()

self.timer.stop()

IF len(self.topChromosomes):

self.parent.btnStop.setText('View Result')

self.parent.btnStop.clicked.disconnect(self.stopOperation)

self.parent.btnStop.clicked.connect(self.dialog.close)

self.parent.lblCPU.setText('CPU Usage: Stopped')

self.parent.lblMemory.setText('Memory Usage: Stopped')

self.parent.lblStatus.setText('Status: Stopped')

self.totalResource['cpu'] = mean(self.totalResource['cpu'])

self.totalResource['memory'] = mean(self.totalResource['memory'])

self.meta 🡨 [[chromosome[1], chromosome[0].fitnessDetails] FOR chromosome IN self.topChromosomes]

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('INSERT INTO results (content) VALUES (?)', [Binary(pickle.dumps({'data': [chromosome[0].data for chromosome in self.topChromosomes], 'meta': self.meta, 'time': self.time.toString('hh:mm:ss'), 'resource': self.totalResource, 'rawData': self.data}, pickle.HIGHEST\_PROTOCOL))])

conn.commit()

conn.close()

END IF

ELSE:

self.dialog.close()

Function 🡨 updateResource(self, resource):

self.tick 🡨 self.tick + 1

IF self.tick == 3:

self.tick 🡨 0

ELSE:

self.totalResource['cpu'].append(resource[0])

self.totalResource['memory'].append(resource[1][1])

self.parent.lblCPU.setText('CPU Usage: {}%'.format(resource[0]))

self.parent.lblMemory.setText('Memory Usage: {}% - {} MB'.format(resource[1][0], resource[1][1]))

Class 🡨 ResourceTrackerWorker(QtCore.QThread):

signal 🡨 QtCore.pyqtSignal(object)

running 🡨 True

runThread 🡨 True

Function 🡨 \_\_init\_\_(self):

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

Function 🡨 \_\_del\_\_(self):

self.wait()

Function 🡨 run(self):

WHILE (self.runThread):

self.sleep(1)

IF self.running IS True:

cpu 🡨 ResourceTracker.getCPUUsage()

memory 🡨 ResourceTracker.getMemoryUsage()

memory 🡨 [ResourceTracker.getMemoryPercentage(memory), ResourceTracker.byteToMegabyte(memory[0])]

self.signal.emit([cpu, memory])

END IF

END WHILE LOOP

Return 🡨 True

**RESULT VIEWER**

This code is used to create a program that can analyse data from a database. It starts by importing the necessary modules and then creates a class called ResultViewer. It then sets up the UI of the custom dialog and assigns the tableResult property. Finally, it runs the analysis steps and gets the settings for the analysis. It then creates a function called getLastResult which gets the results from the database and displays them in a table. Finally, when the user clicks the 'Run' button, the program will execute all of the queries in sequence.

**ALGORITHM : RESULT VIEWER**

from PyQt5 🡨 import 🡪 QtWidgets

from components 🡨 import 🡪 Settings, Database as db, ScheduleParser

from py\_ui 🡨 import 🡪 Result as Parent

import 🡪 pickle

import 🡪 json

import 🡪 csv

import 🡪 copy

Class 🡨 ResultViewer

Function 🡨 \_\_init\_\_(self):

self.dialog 🡨 dialog 🡨 QtWidgets.QDialog()

# Initialize custom dialog

self.parent 🡨 parent 🡨 Parent.Ui\_Dialog()

# Add parent to custom dialog

parent.setupUi(dialog)

self.table 🡨 self.parent.tableResult

self.run 🡨 True

self.settings 🡨 Settings.getSettings()

self.result 🡨 { 'data': [] }

self.getLastResult()

IF self.run:

self.parseResultDetails()

self.connectWidgets()

self.updateTable(0)

dialog.exec\_()

END IF

END Function

Function 🡨 getLastResult(self):

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

cursor.execute('SELECT content FROM results WHERE id = (SELECT MAX(id) FROM results)')

result 🡨 cursor.fetchone()

conn.close()

IF result:

THEN self.result = pickle.loads(result[0])

ELSE:

messageBox 🡨 QtWidgets.QMessageBox()

messageBox.setWindowTitle('No Data')

messageBox.setIcon(QtWidgets.QMessageBox.Information)

messageBox.setText('You haven\'t generated a solution yet!')

messageBox.setStandardButtons(QtWidgets.QMessageBox.Ok)

messageBox.exec\_()

self.run 🡨 False

END ELSE

END Function

Function 🡨 parseResultDetails(self):

IF NOT len(self.result['data']):

THEN Return 🡨 False

END IF

result 🡨 self.result

self.rawData 🡨 copy.deepcopy(result['rawData'])

self.parent.lblTime.setText('Generation Time: {}'.format(result['time']))

self.parent.lblCPU.setText('Average CPU Usage: {}%'.format(round(result['resource']['cpu']), 2))

self.parent.lblMemory.setText('Average Mem Usage: {} MB'.format(round(result['resource']['memory']), 2))

self.updateEntries(0)

self.updateDetails(0)

Function 🡨 connectWidgets(self):

self.parent.cmbChromosome.currentIndexChanged.connect(self.updateDetails)

self.parent.cmbCategory.currentIndexChanged.connect(self.updateEntries)

self.parent.cmbEntry.currentIndexChanged.connect(self.updateTable)

self.parent.btnExport.clicked.connect(self.export)

Function 🡨 updateDetails(self, index):

parent 🡨 self.parent

meta 🡨 self.result['meta'][index]

parent.lblFit.setText('Total Fitness: {}%'.format(meta[0]))

parent.lblSbj.setText('Subject Placement: {}%'.format(meta[1][0]))

parent.lblSecRest.setText('Section Rest: {}%'.format(meta[1][2]))

parent.lblSecIdle.setText('Section Idle Time: {}%'.format(meta[1][4]))

parent.lblInstrRest.setText('Instructor Rest: {}%'.format(meta[1][3]))

parent.lblInstrLoad.setText('Instructor Load: {}%'.format(meta[1][6]))

parent.lblLunch.setText('Lunch Break: {}%'.format(meta[1][1]))

parent.lblMeet.setText('Meeting Pattern: {}%'.format(meta[1][5]))

parent.cmbCategory.setCurrentIndex(0)

parent.cmbEntry.setCurrentIndex(0)

self.updateEntries(0)

self.updateTable(0)

Function 🡨 updateEntries(self, index):

IF index == 0:

THEN key <- 'sections'

ELSE IF index == 1:

THEN key 🡨 'rooms'

ELSE key 🡨 'instructors'

self.entryKeys = []

self.changingKeys = True

self.parent.cmbEntry.clear()

FOR id, entry in self.rawData[key].items():

self.entryKeys.append(id)

self.parent.cmbEntry.addItem(entry[0])

END FOR loop

self.changingKeys 🡨 False

self.updateTable(self.parent.cmbEntry.currentIndex())

Function 🡨 updateTable(self, index):

IF self.changingKeys:

THEN Return 🡨 False

chromosome 🡨 self.result['data'][self.parent.cmbChromosome.currentIndex()]

category 🡨 self.parent.cmbCategory.currentIndex()

# {secId: {'details': {sbjId: [roomId, instructorId, [day/s], startingTS, length]}}}

sections 🡨 chromosome['sections']

rawData 🡨 self.rawData

data 🡨 []

# Section

IF category == 0:

subjects 🡨 sections[self.entryKeys[index]]['details']

FOR subject, details IN subjects.items():

IF NOT len(details):

THEN continue

instructor 🡨 '' IF NOT details[1] ELSE rawData['instructors'][details[1]][0]

data.append({'color': None, 'text': '{} \n {} \n {}'.format(rawData['subjects'][subject][2], rawData['rooms'][details[0]][0], instructor), 'instances': [[day, details[3], details[3] + details[4]] for day in details[2]]})

END FOR loop

END IF

# Room

ELSE IF category == 1:

FOR section, details in sections.items():

FOR subject, subjectDetail in details['details'].items():

IF not len(subjectDetail):

THEN continue

IF subjectDetail[0] != self.entryKeys[index]:

THEN continue

instructor = '' IF not subjectDetail[1] ELSE rawData['instructors'][subjectDetail[1]][0]

data.append({'color': None, 'text': '{} \n {} \n {}'.format(rawData['subjects'][subject][2], rawData['sections'][section][0], instructor),

'instances': [[day, subjectDetail[3], subjectDetail[3] + subjectDetail[4]] FOR day in

subjectDetail[2]]})

END both FOR loop

END ELSE IF

# Instructor

ELSE:

FOR section, details in sections.items():

FOR subject, subjectDetail in details['details'].items():

IF not len(subjectDetail):

THEN continue

IF subjectDetail[1] != self.entryKeys[index]:

THEN continue

data.append({'color': None, 'text': '{} \n {} \n {}'.format(rawData['subjects'][subject][2], rawData['rooms'][subjectDetail[0]][0], rawData['sections'][section][0]), 'instances': [[day, subjectDetail[3], subjectDetail[3] + subjectDetail[4]] FOR day in subjectDetail[2]]})

END FOR loop

END ELSE

self.loadTable(data)

END Function

Function 🡨 loadTable(self, data=[]):

self.table.reset()

self.table.clearSpans()

ScheduleParser.ScheduleParser(self.table, data)

Function 🡨 export(self):

directory 🡨 QtWidgets.QFileDialog().getExistingDirectory(None, 'Select Directory for Export')

IF NOT directory:

THEN Return 🡨 False

WITH open('timeslots.json') as json\_file:

timeslots 🡨 json.load(json\_file)['timeslots']

CLOSE json\_file

fieldnames 🡨 ['Time', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']

rawData 🡨 self.rawData

chromosome 🡨 self.result['data'][self.parent.cmbChromosome.currentIndex()]

# Create schedule for sections

WITH open('{}/sections\_schedule.csv'.format(directory), 'w', newline='') as file:

writer 🡨 csv.writer(file, dialect='excel')

FOR section, subjects in chromosome['sections'].items():

writer.writerow([self.rawData['sections'][section][0]])

writer.writerow(fieldnames)

schedule 🡨 [['' FOR j IN range(6)] FOR i IN

range(self.settings['ending\_time'] - self.settings['starting\_time'] + 1)]

FOR subject, details IN subjects['details'].items():

IF NOT len(details):

THEN continue

instructor 🡨 '' IF NOT details[1] else rawData['instructors'][details[1]][0]

FOR timeslot IN range(details[3], details[3] + details[4]):

FOR day IN details[2]:

schedule[timeslot][day] 🡨 '{} - {} - {}'.format(rawData['subjects'][subject][2],

rawData['rooms'][details[0]][0], instructor)

END FOR loops

END FOR loop

FOR timeslot IN range(self.settings['starting\_time'], self.settings['ending\_time'] + 1):

writer.writerow([timeslots[timeslot], \*schedule[timeslot - self.settings['starting\_time']]])

END FOR loop

writer.writerow([''])

#Create schedule FOR instructors

WITH open('{}/instructors\_schedule.csv'.format(directory), 'w', newline='') as file:

writer 🡨 csv.writer(file, dialect='excel')

FOR instructor IN rawData['instructors'].keys():

writer.writerow([rawData['instructors'][instructor][0]])

writer.writerow(fieldnames)

schedule = [['' FOR j IN range(6)] FOR i IN

range(self.settings['ending\_time'] - self.settings['starting\_time'] + 1)]

FOR section, subjects IN chromosome['sections'].items():

FOR subject, details IN subjects['details'].items():

IF NOT len(details) or details[1] != instructor:

continue

FOR timeslot IN range(details[3], details[3] + details[4]):

FOR day IN details[2]:

schedule[timeslot][day] = '{} - {} - {}'.format(rawData['subjects'][subject][2], rawData['rooms'][details[0]][0],

rawData['sections'][section][0])

FOR timeslot IN range(self.settings['starting\_time'], self.settings['ending\_time'] + 1):

writer.writerow([timeslots[timeslot], \*schedule[timeslot - self.settings['starting\_time']]])

writer.writerow([''])

**UTILITIES**

A function in code generates a random color. If you call the with arguments (just type colorGenerator() in Python), it will generate three colors, red, green, and blue, each evenly spaced between 0 and 255.

**ALGORITHM : UTILITIES**

Import 🡨 random

From numpy 🡨 import 🡪 random as np

Function 🡨 colorGenerator():

RETURN [random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)]

Function 🡨 textColor(rgb):

RETURN [0, 0, 0] IF (((rgb[0] \* 299) + (rgb[1] \* 587) + (rgb[2] \* 114)) / 1000) > 123 ELSE [255, 255, 255]

IF \_\_name\_\_ == '\_\_main\_\_':

FOR i IN RANGE(3):

settings = []

settings.append(np.randint(50, 200))

settings.append(np.randint(settings[0], 200))

settings.append(np.randint(50, 150))

settings.append(np.randint(1500, 4500))

settings.append(round(np.random() / 5, 2))

settings.append(np.randint(90, 100))

settings.append(np.randint(0, 10))

settings.append(np.randint(50, 75))

print 🡨 settings

**TABLE MODEL**

The rowCount and columnCount methods return what percentage rows and columns there are in total respectively. These methods also take parent arguments which permit you to specify where they should start counting from (the first row/column). The code may be a class that will be used to store data in a table. The next line is self.header = header which sets up our header for us so we do not need to worry about setting up any headers ourselves when creating tables with this model object. The last line during this section is to set data in row and column.

**ALGORITHM : TABLE MODEL**

Import 🡪 PyQt5 🡪 QtCore

Class 🡨 TableModel (QtCore.QAbstractableModel)

Function 🡨 \_\_init\_\_ (self,header,data):

Super() 🡨 \_init\_()

self.data 🡨 data

self.header 🡨 header

Function 🡨 data (self,index,role):

IF (index not valid)

THEN Return 🡨 QtCore.QVariant()

ELSE IF (role not equals QtCore.Qt.DisplayRole)

THEN return QtCore.QVariant()

END ELSE IF

Return 🡨 self.data[index.rows][index.column]

Function 🡨 rowCount (self,parent=None,\*args,\*\*kwargs):

Return 🡨 length of self.data

Function 🡨 columnCount (self, parent=None,\*args,\*\*kwargs):

Return 🡨 length of self.data[0]

Function 🡨 headerData (self, p\_int, Qt\_Orientation, role=None):

IF (Qt\_Orientation equals QtCore.Qt.Horizontal AND role equals QtCore.Q.DisplayRole):

THEN Return 🡨 QtCore.QVariant (self.header[0][p\_int]

END IF

Return 🡨 QtCore.QVariant()

Function 🡨 setData (self, index, value, role=None):

IF (index not valid)

THEN Return 🡨 False

END IF

self.data[index.row()][index.column()] 🡨 value

self.dataChanged.emit(index, index)

Return 🡨 True

**TIMETABLE**

This code defines two classes: Timetable and TimetableModel.

The Timetable class creates a timetable object that is displayed in a QTableView widget (passed as an argument to the constructor). It creates an instance of TimetableModel, which it assigns to the table view's model, and sets some properties of the table view, such as the size of its cells. Additionally, it connects some signals (e.g. clicked) to slots (e.g. toggleCells) so that certain actions are taken when the table view is interacted with. The toggleCells function toggles the availability of the selected cells and change the UI color to appropriate color.

The TimetableModel class is a subclass of TableModel.TableModel and provides color support for availability status. It overrides the data() method to return a background color depending on the cell's availability status.

The generateRawTable function create a 2D array, with each cell representing a time slot in the timetable and with default value of 'Available'.

**ALGORITHM : TIMETABLE**

From PyQt5 🡨 import 🡪 QtCore, QtWidgrts, QtGui

From components 🡨 import 🡪 Settings, TableModel

Import 🡪 json

#Used for displaying toggable timetable

Class 🡨 Timetable:

Function 🡨 \_\_init\_\_(self, table, data=False):

Self.table 🡨 table

Header 🡨 [[‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’]]

open 🡨 ‘timeslots.json’ as json\_file:

timeslots 🡨 json.load (json\_file) [‘timeslots’]

settings 🡨 Settings.getSettings()

header..append (timeslots [settings [‘starting\_time’] :settings [‘ending\_time’] + 1 ])

self.data 🡨 data

IF NOT data:

self.data 🡨 [ ]

FOR i IN range ( settings['ending\_time'] + 1 - settings['starting\_time']):

self.data.append (['Available', 'Available', 'Available', 'Available', 'Available', 'Available'])

self.model 🡨 TimetableModel(header, self.data)

table.setModel(self.model)

table.horizontalHeader().setSectionResizeMode(QtWidgets.QHeaderView.Fixed)

table.verticalHeader().setSectionResizeMode(QtWidgets.QHeaderView.Fixed)

table.clicked.connect(self.toggleCells)

table.horizontalHeader().sectionClicked.connect(self.toggleCells)

table.verticalHeader().sectionClicked.connect(self.toggleCells)

table.findChild(QtWidgets.QAbstractButton).clicked.connect(self.toggleCells)

# Toggles the availability and changes UI color to appropriate color

Function 🡨 toggleCells (self):

Indexes 🡨 self.table.selectionModel().selectedIndexes()

FOR i IN indexes:

value 🡨 'Available' if self.data[i.row()][i.column()] == 'Unavailable' ELSE 'Unavailable'

IF value == ‘Available’:

self.table.setStyleSheet('selection-background-color: rgb(46, 204, 113); selection-color: black;')

ELSE:

self.table.setStyleSheet('selection-background-color: rgb(231, 76, 60); selection-color: black;')

self.model.setData(i, value)

Function 🡨 getData (self):

Return self.data

# Timetable model that provides color support for availability status

Class 🡨 tometableModel (TableModel.TableModel):

Function 🡨 \_\_inti\_\_(slef, header, data):

Super().\_\_init\_\_(header, data)

Function 🡨 data (self, index, role):

IF NOT index.isValid():

RETURN QtCore.varient()

ELIF role == QtCore.Qt.BackgroundRole:

IF self.data[index.row()][index.column()] == 'Available':

RETURN QtGui.QBrush(QtGui.QColor(46, 204, 113))

ELSE:

RETURN QtGui.QBrush(QtGui.QColor(231, 76, 60))

ELIF role != QtCore.Qt.DisplayRole:

RETURN QtCore.QVariant()

RETURN self.data[index.row()][index.column()]

Function 🡨 generateRawTable():

Settings 🡨 Settings.getSettings()

Data 🡨 [ ]

FOR i IN RANGE (settings['ending\_time'] + 1 - settings['starting\_time']):

data.append(['Available', 'Available', 'Available', 'Available', 'Available', 'Available'])

RETURN data

**SHARE**

The The code starts by creating a cursor object. The cursor is then wont to execute the SQL statement "SELECT id, name, subjects FROM sections WHERE active = 1 AND id != ?". this may return all of the rows from the sections table where the active column has a value of 1 and that row's ID does not equal self.section\_id. Next, we use another SQL statement to urge all of the rows from our sharings table where subjectId equals self.id and final equals 1. The code ends with an execution that uses a get statement to get all of the rows in our sections table where subjectId equals self.id and final equals 0 (meaning it hasn't been deleted). The code attempts to list all the sections of a course with a lively status. If the section doesn't exist, then it'll create a new section and populate it with data from the database. If the section exists, then it'll execute a query to find out which courses share that particular subject. The code starts by creating an inventory of sections. It then creates a dictionary called sharings that maps the section ID to the JSON string representation of the sharing. The code then iterates through each section and adds it to its own row within the model, which is an object representing all data about this project. The finish() function checks if there are any selected indexes on the tree view and returns false if there are not any . If so, it gets a connection from db with conn and executes an INSERT statement into sharings for self's ID as subjectId and self's section\_id as value . Then it sets self's shareID adequate to lastrowid from cursor . Otherwise, it gets subjectID , text , and split values for every index on the tree view item (which represents one row) using getItemText() method on model .  
The code attempts to make a list of sections and the corresponding sharing information. the primary thing that happens in this code is that a list of all the sections are created. Then, for every section, an inventory of shares associated with it is created. Finally, for every share, an insert query is executed into the sharings table to feature the share's subject ID and section ID to the sharings table.

**ALGORITHM : SHARE**

from PyQt5 🡨 import 🡪 QtWidgets, QtGui

from components 🡨 import 🡪 Database as db

from py\_ui 🡨 import 🡪 Share as Parent

import 🡪 json

class 🡨 Share:

Function 🡨 \_\_init\_\_(self, subject\_id, section\_id):

self.id 🡨 int(subject\_id)

self.section\_id 🡨 int(section\_id)

self.shareId 🡨 False

self.shareMembersText 🡨 False

# New instance of dialog

self.dialog 🡨 dialog 🡨 QtWidgets.QDialog()

# Initialize custom dialog

self.parent 🡨 parent 🡨 Parent.Ui\_Dialog()

# Add parent to custom dialog

parent.setupUi(dialog)

self.setSharings()

parent.btnFinish.clicked.connect(self.finish)

parent.btnCancel.clicked.connect(self.dialog.close)

dialog.exec\_()

Function 🡨 getShareData(self):

Return 🡨 tuple([self.shareId, self.shareMembersText])

Function 🡨 setSharings(self):

self.tree 🡨 tree 🡨 self.parent.treeSections

self.model 🡨 model 🡨 QtGui.QStandardItemModel()

model.setHorizontalHeaderLabels(['ID', 'Sections', 'SectionID'])

tree.setModel(model)

tree.setColumnHidden(0, True)

tree.setColumnHidden(2, True)

model.itemChanged.connect(lambda item: self.toggleSharing(item))

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

# Get sections with mutual subjects

IF self.section\_id:

THEN cursor.execute('SELECT id, name, subjects FROM sections WHERE active = 1 AND id != ?', [self.section\_id])

ELSE:

cursor.execute('SELECT id, name, subjects FROM sections WHERE active = 1')

sections 🡨 cursor.fetchall()

cursor.execute('SELECT id, sections FROM sharings WHERE subjectId = ? AND final = 1', [self.id])

sharings 🡨 cursor.fetchall()

sharedSections 🡨 list(set([section for sectionGroup in list(

map(lambda sharing: list(map(lambda id: int(id), json.loads(sharing[1]))), sharings)) for section in

sectionGroup]))

sharings 🡨 dict(sharings)

conn.close()

sectionDict 🡨 {}

FOR section IN sections:

sectionDict[section[0]] 🡨 section[1]

IF self.id NOT IN list(map(lambda id: int(id), json.loads(section[2]))) or section[0] in sharedSections:

THEN continue

id 🡨 QtGui.QStandardItem()

id.setEditable(False)

sectionList 🡨 QtGui.QStandardItem(section[1])

sectionList.setEditable(False)

sectionID 🡨 QtGui.QStandardItem(str(section[0]))

sectionID.setEditable(False)

self.model.appendRow([id, sectionList, sectionID])

END FOR LOOP

FOR key, value in sharings.items():

sectionIDList 🡨 list(map(lambda id: int(id), json.loads(value)))

IF self.section\_id IN sectionIDList:

THEN continue

id 🡨 QtGui.QStandardItem(str(key))

sectionList = QtGui.QStandardItem(', '.join(list(map(lambda id: sectionDict[id], sectionIDList))))

sectionList.setEditable(False)

sectionID = QtGui.QStandardItem(','.join(map(str, sectionIDList)))

self.model.appendRow([id, sectionList, sectionID])

END FOR LOOP

Function 🡨 finish(self):

IF NOT self.tree.selectedIndexes():

Return 🡨 False

shareId 🡨 self.model.item(self.tree.selectedIndexes()[0].row()).text()

shareId 🡨 False if not shareId else shareId

conn 🡨 db.getConnection()

cursor 🡨 conn.cursor()

IF NOT shareId:

cursor.execute('INSERT INTO sharings (subjectId, sections) VALUES (?, ?)', [self.id, json.dumps(

[self.section\_id, self.model.item(self.tree.selectedIndexes()[0].row(), 2).text()])])

self.shareId = cursor.lastrowid

ELSE:

subjectID 🡨 self.model.item(self.tree.selectedIndexes()[0].row(), 2).text().split(',')

subjectID.append(self.section\_id)

cursor.execute('UPDATE sharings SET sections = ? WHERE id = ?', [json.dumps(subjectID), shareId])

self.shareId 🡨 shareId

conn.commit()

conn.close()

self.shareMembersText = self.model.item(self.tree.selectedIndexes()[0].row(), 1).text()

self.dialog.close()

**REFERENCES**

Genetic Algorithm

1. L.Haldurai, T.Madhubala, and R.Rajalakshmi, “A study on Genetic Algorithm and its Application” International Journal of Computer Sciences and Engineering, Volume-4, Issue-10, E-ISSN: 2347-2693
2. H.P. Hariyadi, T. Widiyaningtyas, et. al., Implementation of Genetic Algorithm to Academic Scheduling System, IEEE, Region 10 Conference, (2016).
3. J. W. Shen, Solving the graph coloring problem using genetic programming, "Genetic Algorithms and Genetic Programming at Stanford 2003, Ed. J. R. Koza, 187-196. Publisher: Stanford Bookstore.
4. K. F. Man, K. S. Tang, and S. Kwong, Genetic Algorithms: Concepts and Applications, IEEE Trans. Ind. Electron. 43 (5), 519 – 534, (1996).
5. L. Lalescu, C. Badica, Timetabling experiments using genetic algorithms. In Proceedings of the International 12th Turkish Symposium on Artificial Intelligence and Neural Networks (TAINN-2003), Canakkale, Turkey, (2003).
6. Yao Zhou, Study On Genetic Algorithm Improvement And Application, May 2006

Timetable + Genetic Algorithm

1. S. Daskalaki, T. Birbas, E. Housos, An integer programming formulation for a case study in university timetabling, Eur. J. Oper. Res., 153, 117-135 (2004).
2. T. Islam, Z. Shahriar, et. al., University Timetable Generator Using Tabu Search, Journal of Computer and Communications, 4,28-37, (2016).
3. T. Jain, N. Jamil, Genetic Algorithm Approach to Time Tabling Problem, European Journal of Business and Management, 7(4) 7-11, (2015).
4. D. Mitta, H. Doshi, M. Sunasra, R. Nagpur, Automatic Timetable Generation using Genetic Algorithm, Int. J. Adv. Res. Comput. Commun. Eng., 4 (2), 245-248, (2015).
5. Ghazi Alnowaini, Amjad Abdullah Aljomai, Genetic Algorithm For Solving University Course Timetabling Problem Using Dynamic Chromosomes, 2021 ICTSA, DOI: 10.1109/ICTSA52017.2021.9406539
6. Deeba Kannan, Kuntal Bajpayee, Samriddho Roy, Solving Timetable Scheduling Problems Using Genetic Algorithm, International Journal of Recent Technology and Engineering (IJRTE), ISSN: 2277-3878, Volume-7, Issue-5C, February 2019

Timetable

1. E. Burke, K. Jackson, J. H. Kingston, R. Weare, Automated university timetabling: The state of the art, The Computer Journal, 40 (9): 565-571, (1997).
2. S. Petrovic, E. K. Burke 2004. University Timetabling. In: Leung J. (ed.) Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Chapter 45. CRC Press.