Goodwing Timetabler

How to Contribute - The code explained
by Hugo Bonnell

15 janvier 2025

Table des matières

1 The problem

1.1 Problem Description

The disruptions caused by the COVID-19 pandemic have introduced new constraints in managing several complex problems of daily life. Among these, work schedule planning, an NP-hard optimization problem, has been particularly affected. This type of problem appears in various sectors such as administration, transportation, production, healthcare, and education.

In this context, we focus on the problem of optimizing university timetables. Before the pandemic, creating a timetable involved respecting certain constraints, such as the availability of teachers, classrooms, and permissible time slots. Any solution meeting these constraints was considered optimal. However, new constraints have emerged with the pandemic, such as limiting the number of students physically present, distance learning, hybrid formats, making the planning process more complex.

The objective of this project is to propose solutions, either exact or approximate, for the problem of course scheduling by considering both classical constraints (availability of teachers and classrooms) and new constraints related to teaching methods (in-person, distance, or hybrid), as well as balancing the workload for students according to these modalities.

1.2 Problem Definition

- Our university has :
  - A given **r** number of rooms. Each room might be a **special room**.
    - **Special rooms** are rooms with specific furnitures. For instance lab furnitures are in **lab** rooms. **Lab** rooms are considered as a **special room**.
    - Rooms that aren't **special rooms** are refered too as **rooms**.
    - **Amphitheatres** are considered as **special rooms**.
  - A given **t** amount of teachers. Each teacher has one or more corresponding subjects he/she can teach.
  - **N** promotions (= year the student is on). Each promotion has :
    - **n** classes (= group of students). To each class we assign :
      - **s** subjects (= materials) the class has to attend.

- Each of the **s** subjects has :
  - A fixed number of hours **h** to be completed.
  - The **h** hours are divided in **online_hours** and **presential_hours**.
  - We need to make sure that at most 30% of the **h** hours are **online_hours**.

- To complete the **h** hours :
  - One course duration is 1h30.
  - A class can attend as many **courses** as required to attend the **h** hours.
  - Some **courses** need to take place in special rooms depending on the **course**'s format. (For example practical works need to be in labs)
  - Two different classes can't attend the same **course** in the same **room** on the same **timeslot**.
    - An exception is made for **courses** taking place in **amphitheatres**
  - One class can't have two **courses** in the same **timeslot**

- The **courses** can take place in defined **timeslots** each day.
  - **Timeslots** are defined as follows :
    - 08 :15 to 09 :45
    - 10 :00 to 11 :30
    - 11 :45 to 13 :15
    - 13 :30 to 15 :00
    - 15 :15 to 16 :45
    - 17 :00 to 18 :30
    - 18 :45 to 20 :15

  - No **course** can be scheduled on saturday and thursday afternoon (from 13 :30 to 20 :15). No **course** ca be scheduled on Sunday.
  - Students must have at least one free slot a day. Either 11 :45 to 13 :15 or 13 :30 to 15 :00. This is to ensure they can eat.
  - **Courses** can be either **online** or **presential**. Respecting the previous condition regarding total amount of online hours.
  - Certain **courses** may require specific time slots due to logisitcal reasons. (e.g. lab courses requiring 3h sessions, thus, two back-to-back **timeslots**)

- To take place, each **course** needs :
  - An available room. (= A room where there is no course taking place)
  - A corresponding **class**
  - A **teacher** that can teach the **course**'s subject.
    - The **teacher** must be available. He can't give two courses at the same time.
    - The **teacher** may have preferred teaching time or days. The **schedule** should try to accomodate these perferences as much as possible.

- Room and Resource Allocation :
  - Scheduling must allocate regular or special rooms (e.g., labs or amphitheaters) based on course requirements.
  - Ensure adequate use of room resources while preventing overbooking or misuse of special-purpose spaces.

2 Code Structure

Corresponding to V 0.0.1

2.1 The app itself

The whole app is stored inside of the **GoodwingTimetabler** folder. Which contains :

- app/
- csp/
- myTests/
- util/
- __main__.py

2.1.1 __main__.py

The **__main__.py** file is the file called when we're launching the app using :

```bash
# This bash script launches the app
cd C:/path_to_repo
python ./GoodwingTimetabler
```

The __main.py__ file contains the following :

```python
# Import the run_app and run_test functions from the 'app' module
from app import run_app, run_test

# Call the run_test function to start running the app (in its test state for now)
run_test()
```

2.1.2 The modules

All the other folders are modules and respect the module folder structure.

- __init__.py | This is where we turn the folder into a python module
- fileA.py
- fileB.py
- ...
- Others necessary .py files. The useful "exported" functions are specified in init.py

main.py calls the app.run_test() function. This function itself calls the generateScheduleUsingCSP() function inside of myTests module, itself creating a university object, creating and solving its corresponding CSP and outputting the schedules as pdf files.

3 The model

You can find all defined objects in csp/objects.py

3.1 Base classes

Timeslot

The timeslot takes datetime arguments :

- A day (datetime.date fomat, we're excluding the hour information)
- A starting time (datetime.time format, excluding the day information)
- An ending time (datetime.time format)

This lets us track the different available timeslots each day, to know where we can add courses later on. (Courses will be linked to an id corresponding to a timeslot position in the timeslots list).

```
    """
    Object representing a timeslot.\n
    Parameters:\n
    - day : datetime.date | Date of the timeslot
    - start : datetime.time | Start date and time of the timeslot
    - end: datetime.time | End date and time of the timeslot
    """
    def __init__(self, day: dt.date, start: dt.time, end: dt.time
       ):
        self.day = day
        self.start:dt.time = start
        self.end:dt.time = end

    def __str__(self):
        return f"Timeslot date: {self.day} , starts at: {self.
           start} , ends at: {self.end}"
```

Person

Base class used to define students (not necessary nor used at the moment) and teachers (used !).

```python
class Person:
"""
Object for any person (teacher, student, etc...)\n
Parameters:\n
- first_name: str | First name of the person
- last_name: str | Last name of the person
"""
def __init__(self, first_name: str, last_name: str):
    self.first_name = first_name
    self.last_name = last_name

def __str__(self):
    return f"{self.first_name} {self.last_name}"
```

Subject

Class used to define the different university subjects, with its :

- name and id (id = 0 by default, else considered as a string)
- number of hours required to complete the subject (used to compute the necessary courses for this subject)
- a color, to add colors to the final timetable pdf (hexadecimal str reference to a color)

```python
class Subject:
"""
Object representing a school subject.\n
Parameters:\n
- name : str | The name of the subject
- id : str | The id of the subject (if applicable, '0' by
  default)
- hours : float | The number of hours to complete the course
  (9.0 by default)
"""
def __init__(self, name: str, id: str = "0", hours: float =
   9.0, color: str = "29FF65"):
    self.name = name
    self.id = id
    self.hours = hours
    self.color = color

def __str__(self):
    return f"{self.name} with id: {self.id}"
```

3.2 People

Teacher

Inheriting from the Person class, we can define teachers by adding a list of a subject a Person is able to teach to make a teacher !

```
class Teacher(Person):
"""
Object representing a teacher.\n
Parameters:\n
- subjects : [Subject] | List of the subjects the teacher is
   able to teach
"""
def __init__(self, first_name: str, last_name: str, subjects:
    List[Subject] = []):
    super().__init__(first_name, last_name)
    self.subjects = subjects

def __str__(self):
    result = super().__str__() + " can teach: ["
    for subject in self.subjects:
        result += subject.name + " "
    result += "]"
    return result
```

Student

By giving a student id to a Person we can create a student, then a list of students can be added to a group but this feature has no use so far.

```
class Student(Person):
"""
Object representing a student.\n
Parameters:\n
- student_id : str | Badge id of the student (Default : '0')
"""
def __init__(self, first_name: str, last_name: str,
   student_id: str = "0"):
    super().__init__(first_name, last_name)
    self.student_id = student_id

def __str__(self):
    return f"{super().__str__} is a student with id: {self.
       student_id}"
```

3.3 Facilities

Room

As the 'University' is considered a complex class (built using an aggregation of various others), the Room object is the only one in the 'Facilities' section. A list of rooms is used to define a university, giving the list of usable places to take courses.

To define a room, we give it a name, a type ('default' by default, could be 'amphitheater' for instance) and an id if needed (0 by default, can stay this way as we're refering to the name of the room when plotting the timetable).

```python
class Room:
"""
Object representing a room, used by the university.\n
Parameters:\n
- name : str | Name of the room.
- type : str | Type of the room if it's a special room.
   Otherwise "default"
"""
def __init__(self, name: str, type: str = "default", id: int
   = 0):
    self.name = name
    self.type = type
    self.id = id

def __str__(self):
    return f"Room {self.name} has type: {self.type}"
```

3.4 Scholarship

Group

Class used to define students groups inside of promotions (TDA, TDB ...)

```python
class Group:
"""
Object representing a class (in the 'group of students'
   meaning)\n
Parameters:\n
- name : str | The name of the group
- students : [Student] | List of the students in the class,
   None by default
"""
def __init__(self, name: str, students: List[Student] = None)
   :
    self.name = name
    self.students = students

def __str__(self):
    return f"{self.name}"
```

Promotion

Class used to represent the different promotions, group of previously defined groups. (A1, A2...)

```
    class Promotion:
    """
    Object representing a promotion (multiple groups on the same
       level)\n
    Parameters:\n
    - name : str | Name of the promotion
    - groups : [Group] | List of the groups on this promotion
    - subjects : [Subject] | List of the subjects the promotion
       has to attend
    """
    def __init__(self, name: str, groups: List[Group], subjects:
       List[Subject]):
        self.name = name
        self.groups = groups
        self.subjects = subjects

    def __str__(self):
        groupCount = len(self.groups)
        subjectsCount = len(self.subjects)
        return f"Promotion {self.name} has {groupCount} groups
           and must attend {subjectsCount} subjects."
```

3.5 Complex end objects

When combining previously defined bricks, we can make end objects used in our CSP.

Course

The course class is defined as a comment below. Note that it can be turned into a YAML entry for the pdf timetale outputed when the CSP is solved.

```python
class Course:
"""
Object representing the courses.\n
Prameters:\n
- timeslot : Timeslot | The Timeslot the course takes place on
- group : Group | The group attending the course
- subject : Subject | The subject the course is about
- teacher : Teacher | The teacher giving the course
- room : Room | The room where the course takes place
"""
def __init__(self, timeslot: Timeslot, group: Group, subject: Subject, teacher: Teacher, room: Room):
    self.timeslot = timeslot
    self.group = group
    self.subject = subject
    self.teacher = teacher
    self.room = room

def __str__(self):
    return f"Course: {self.subject.name} | Group: {self.group.name} | Teacher: {self.teacher.first_name} {self.teacher.last_name} | Timeslot: {self.timeslot} | Room: {self.room.name} ({self.room.type})"

def to_yaml_entry(self):
    day = self.timeslot.day.strftime('%a')
    time = f"{self.timeslot.start.strftime('%H:%M')} - {self.timeslot.end.strftime('%H:%M')}"
    entry = {
        "name": f"{self.subject.name}\n{self.teacher.first_name} {self.teacher.last_name}\n({self.room.name})",
        "days": day,
        "time": time,
        "color": self.subject.color,
    }
    return entry
```

University

The University class is also defined as a comment in the code below. Note that the reference_date related stuff is **only** used to compute the duration of a course.

```python
class University:
    """
    Object representing a university.\n
    Parameters:\n
    - name : str | University's name
    - rooms : [Room] | List of rooms in our university.
    - teachers: [Teacher] | List of teachers in the university.
    - promotions: [Promotion] | List of promotions in the
       university.
    - start_date: datetime.date | First day of school.
    - days: int | number of days that the semester lasts.
    - time_ranges: [Timeslot] | Allowed Timeslots
    """
    def __init__(self, name: str, rooms: List[Room], teachers:
       List[Teacher], promotions: List[Promotion], start_date: dt.
       date, days: int, time_ranges: List[tuple]):
        self.name = name
        self.rooms = rooms
        self.teachers = teachers
        self.promotions = promotions
        self.timeslots: List[Timeslot] = generate_timeslots(
           start_date, days, time_ranges)

        start_time = self.timeslots[0].start
        end_time = self.timeslots[0].end
        # Combine with a reference date
        reference_date = date.today()
        start_datetime = dt.datetime.combine(reference_date,
           start_time)
        end_datetime = dt.datetime.combine(reference_date,
           end_time)
        # Calculate the difference in hours
        duration = (end_datetime - start_datetime).total_seconds
           () / 3600.0

        self.timeslot_duration = duration

    def __str__(self):
        return f"{self.name} has {len(self.rooms)} rooms, {len(
           self.teachers)} teachers and {len(self.promotions)}
           promotions."
```

4 OR Tools

4.1 What's OR Tools

OR-Tools is an open-source optimization library developed by Google that provides robust tools for solving complex optimization problems. It supports a variety of problem-solving techniques, including linear programming, mixed-integer programming, routing problems, and constraint satisfaction problems (CSPs). Written in C++ with Python bindings, it is highly efficient and versatile for real-world applications.

Applications to CSP

Constraint Satisfaction Problems (CSPs) involve finding solutions that satisfy a set of constraints. OR-Tools provides a powerful CP-SAT solver designed specifically for this purpose. It allows users to model variables, define constraints, and specify optimization objectives.

OR-Tools' flexibility and ease of use make it a go-to choice for researchers, developers, and businesses tackling optimization challenges.

4.2 OR Tools basics - Solve a simple CSP

OR-Tools is a powerful optimization library designed to model and solve optimization problems efficiently. Its core functionality revolves around defining variables, constraints, and objectives to represent a given problem mathematically. OR-Tools is particularly effective for solving Constraint Satisfaction Problems (CSPs), where the goal is to assign values to variables such that a set of constraints is satisfied.

Key Components in OR-Tools

- **Variables :** Represent the decision points in the problem, with domains specifying the possible values they can take.
- **Constraints :** Define the rules or conditions that the solution must satisfy.
- **Solvers :** The engine that processes variables and constraints to find feasible or optimal solutions.
- **Objectives :** *(Optional)* Define criteria to optimize (e.g., minimize or maximize).

Solving a simple CSP

Let's solve a simple CSP where we assign values to three variables x, y, z such that :

- $x + y + z = 10$
- $x < y < z$
- $x, y, z \in 1, 2, 3, 4, 5, 6, 7, 8, 9$

Steps to solve the CSP

Import OR-Tools → Define the Model → Define Variables → Add Constraints → Solve the Problem

Python code example :

```python
from ortools.sat.python import cp_model

# Create the model
model = cp_model.CpModel()

# Define variables
x = model.NewIntVar(1, 9, 'x')  # x is in [1, 9]
y = model.NewIntVar(1, 9, 'y')  # y is in [1, 9]
z = model.NewIntVar(1, 9, 'z')  # z is in [1, 9]

# Add constraints
model.Add(x + y + z == 10)  # Sum constraint
model.Add(x < y)           # x is less than y
model.Add(y < z)           # y is less than z

# Create a solver
solver = cp_model.CpSolver()

# Solve the model
status = solver.Solve(model)

# Print the solution if it exists
if status == cp_model.FEASIBLE:
    print(f"x = {solver.Value(x)}")
    print(f"y = {solver.Value(y)}")
    print(f"z = {solver.Value(z)}")
else:
    print("No solution found.")
```

A possible output could be : $x = 2, y = 3, z = 5$

5 CSP Implementation

To implement flawlessly OR-Tools, I created a CSP class (inside of csp/csp.py). Feel free to explore this file to know more about all its functions. The CSP class defined this way :

```python
    def __init__(self, university: University):
        self.university = university
        self.model = cp_model.CpModel()
        self.variables = {}  # Dictionary to store variables for each course
        self.generated_courses: List[Course] = []  # List of all generated courses
        self.solver = cp_model.CpSolver()

        self.createVariables()
        self.createConstraints()
        self.solveCSP()
```

Variables definition

```python
    def createVariables(self):
        overall_course_idx = 0
        for promo in self.university.promotions:
            for group in promo.groups:
                self.variables[group.name] = {}
                for subject in promo.subjects:
                    self.variables[group.name][subject.name] = {}

                    # Calculate number of timeslots needed for the subject
                    required_hours = subject.hours
                    timeslot_duration = self.university.timeslot_duration  # Assume in hours
                    num_courses = int(required_hours // timeslot_duration)

                    for idx_course in range(num_courses):
                        overall_course_idx += 1
                        # Timeslot variable
                        timeslot_var = self.model.new_int_var(0, len(self.university.timeslots) - 1, f"course_{overall_course_idx}_timeslot")

                        # Room variable
                        room_var = self.model.new_int_var(0, len(self.university.rooms) - 1, f"course_{
```

```
                        overall_course_idx}_room")

                    # Teacher variable
                    teacher_var = self.model.new_int_var(0,
                        len(self.university.teachers) -1, f"
                        course_{overall_course_idx}_teacher")

                    # Create the variable
                    self.variables[group.name][subject.name][
                        overall_course_idx] = {'subject':
                        subject.name, 'group': group.name, '
                        timeslot': timeslot_var, 'room':
                        room_var, 'teacher': teacher_var}
```

How are the variables created ?

- For every group in the university we add an entry to the variables dictionary with the group name as a key.
- To this key is assigned another 'sub'-dictionnary with different keys. The keys of the sub dictionary are the group's subjects' names.
- For each subject (inside of each group), we compte the number of hours the group has to take, and get the timeslot duration computed in the university object. We then compute how many courses are required to complete this subject.
- We then create 3 variables : timeslot, room and teacher. All of these variables correspond to the index of the timeslot, room and teacher defining the course. This lets us treat timeslots, rooms and teachers as integers, easing our problem.
- All the constraints are defined regarding these ids, and later we can translate all the id back to actual timeslots, rooms and teachers.

6 How to contribute ?

To contribute, you may add constraints to the CSP for the resulting schedule to be more relevant.

How are constraints defined ?

So far, constraints are defined by functions, all called in a CreateConstraints() function :

```
def createConstraints(self):
self.noRoomOverlap()
self.noMultipleCoursesOnTimeslotForGroup()
```

You may want to follow this model of constraint definition, adaptating it to your needs :

noRoomOverlap()

```
def noRoomOverlap(self):
    courses = []
    for _, group in self.variables.items():
        # Group is a Tuple with : (group name, corresponding
          items)
        for _, subject in group.items():
            # Subject is a Tuple with : (subject name,
              corresponding items)
            for course_key, course in subject.items():
                courses.append(course)

    for i in range(len(courses)):
        for j in range(i + 1, len(courses)):
            # Add constraint: if rooms are the same,
              timeslots must be different

            # Create a Boolean variable representing the
              condition
            same_timeslot = self.model.NewBoolVar(f'
              same_timeslot_{i}_{j}')

            # Add the condition to define the Boolean
              variable
            self.model.Add(courses[i]['timeslot'] == courses[
              j]['timeslot']).OnlyEnforceIf(same_timeslot)
            self.model.Add(courses[i]['timeslot'] != courses[
              j]['timeslot']).OnlyEnforceIf(same_timeslot.Not
              ())

```

```
                    # Add the constraint for room overlap, enforced
                        only if the timeslots are the same
                    self.model.Add(courses[i]['room'] != courses[j]['
                        room']).OnlyEnforceIf(same_timeslot)
```

noMultipleCoursesOnTimeslotForGroup()

```python
def noMultipleCoursesOnTimeslotForGroup(self):
    courses = []
    for _, group in self.variables.items():
        # Group is a Tuple with : (group name, corresponding
            items)
        for _, subject in group.items():
            # Subject is a Tuple with : (subject name,
                corresponding items)
            for course_key, course in subject.items():
                courses.append(course)

    for i in range(len(courses)):
        for j in range(i + 1, len(courses)):
            # Check if the groups are the same
            if courses[i]['group'] == courses[j]['group']:
                # Add constraint: courses in the same group
                    cannot share the same timeslot
                self.model.Add(courses[i]['timeslot'] !=
                    courses[j]['timeslot'])
```

Sidenote : Contributing to this project might be hard at first, feel free to test, print stuff, and don't get turned down by the apparent difficulty. Take your time to read and understand the whole code!

- Hugo