Algorithm Fall 2022 Final project
FUV Course scheduling

Group 4

December 23, 2022

1. Problem

Input

- A graph where
  - Each node is a course, and the corresponding maximum number of students.
  - Each edge between two nodes means these two courses can not be scheduled at the same time.
- A list of classrooms and the capacity (i.e. maximum number of students) of each classroom.
- A list of possible class time slots (e.g., 8am to 9:30am Mon/Wed,...).
- A list of time slots (i.e. 10am - 11am Monday) that all faculty members should be available (e.g., for weekly meeting).

Output

A schedule if it is possible, otherwise return "No!" and suggest some constraints that should be reduced.

Constrains

- There is at most one course teaching in a classroom at the same time.
- A satisfied timeslot for a course is a 1.5 hours that the corresponds to a particular classroom's capacity and can be selected as a course time.
- In a week, a course have two timeslot, which are two consecutive timeslots in a day or two timeslots (at the same timestamp) in two day apart. Two timeslots (at the same timestamp) in two day apart is more prioritized.
- A lecturer cannot teach two courses at the same time.
- No course begins after 4.30 p.m.

2. Algorithm

With csv file of n courses with course code, lecturer, capacity, and csv file of m classrooms with classroom name, available time, our code is decoded as follows:

- *Course_list*: List of all courses code with its capacity. This list is sorted ascendingly by capacity.
- *Conflicting_dict*: {faculties'sname: [timeslots]}.
- *Classroom_schedule*: List of timeslots of 1.5 hours with classroom name, available time (day, timestamp). This list is generated ascendingly by capacity and time.

- *Back_ up_ Classroom_ schedule*: List of backup timeslots of 1.5 hours with classroom name, available time (day, timestamp). This list is generated ascendingly by capacity and time.
- *Classroom_ schedule* only includes slots end before 4.30 p.m and *Back_ up_ Classroom_ schedule* includes the slots start after 4.45 p.m. This assumption can be set up differently by user.

Our procedure is designed in two steps: generate a function that will find suitable timeslots for each course which will satisfy our conditions. Then, in the main function, run the previous function for all courses to get the final result. If the main algorithm print **False**, suggestions for backup timeslots (night classes) or adding classroooms will be printed. Else, the result where satisfy all constrains will be generated.

Combining the encoding and procedure above, this is the pseudocode for scheduling problem, which includes an extra function and the main algorithm.

```
   // Function to find a suitable time slot for a given course. Return its
      time slot (2 sessions, classroom/day/time) if possible, otherwise,
      return empty
1  Function arrangeSchedule(classroom_ schedule, course, Conflicting_ dict[faculty]):
      Office hour: List of 6 range of working hours per day
      Office time: List of 5 working days from Mon to Fri
      for classroom in Classroom_ schedule do
         if the course's capacity $\geqslant$ the class's capacity then
            continue
         else
            Available_ slot = Time slots of classroom - Conflicting_ dict [faculty]
            for  time in Office time do
               if  there is only one day in Available_ slot contains this time then
                  continue
               else
                  timeslot1 = the slot of earliest day that contains this time
                  timeslot2 = the slot of next day that contains this time so that two
                  days are at least 1 day apart
                  schedule = timeslot1[day1,time], timeslot2[day2,time]
                  return schedule
               end
            end
            for day in Office time do
               if  there are 2 consecutive timeslots on this day  then
                  schedule = timeslot[day,time], timeslot[day,time+1]
                  return schedule
               end
            end
         end
      end
      return []
2  end
```

Algorithm 1: Course scheduling algorithm

```
Input  : Course_list, Conflicting_dict, Back_up_Classroom_schedule,
         Classroom_schedule
Output: True and a schedule, or False and suggestion.
1  course_schedule = [] // List of schedule corresponding with Course_list
2  Arrange = True
3  for each course in Course_list do
4      schedule = arrangeSchedule(Back_up_Classroom_schedule, course,
         Conflicting_dict[faculty])
5      if schedule is empty list then
6          Arrange = False
7          schedule = arrangeSchedule(Back_up_Classroom_schedule, course,
             Conflicting_dict[faculty])
8          if schedule is empty list then
9              Print: "We do not have enough classroom"
10             continue
11         else
12             Print: "We should add two time slots" and schedule
13         end
14         Update course_schedule, Conflicting_dict with schedule
15     end
16 end
17 Format course_schedule
18 return Arrange, course_schedule
```

The time complexity of this algorithm is $O(mn)$, where m, n be the number of courses, and classrooms, respectively. This is significantly faster and more applicable than the naive algorithm with $\Omega(n!)$ we presented in class.

3. Implementation

All of our code can be found in the attached file. To see the result, the user can run at least one of the two function below in ***generateTestCase.py*** to generate the test case and ***arrange.py*** to run the main algorithm.

3.1. Generate test case

Go to ***generateTestCase.py*** and run the function below where n is number of courses, m is number of classrooms, **classroomPath** is classroom's input path of your desire and **coursePath** is course's input path of your desire

```

genTestCase(n, m, classroomPath, coursePath)
```

3.2. Run the scheduling function

In ***arrange.py***, print the function below where **classroomPath** is the input classroom path, **coursePath** is the input course path, **resultPath** is the output path.

```
print(arrangeCourse(classroomPath, coursePath, resultPath))
```

After running, if our screen prints **True**, it only generates the result in the CSV output path. If it prints **False**, it will show the suggestion (adding timeslots or adding classroom) and generate the result in the output path.

3.3. Example

In this example, we generate test case (input) for 1000 courses and 100 classsroom in **generateTestCase.py**. Without this function, we can still use our generated input and FUV's input in ***Testcase/Input*** for the main function in 3.2

```

genTestCase(1000, 100, 'classroom.csv', 'course.csv')
```

After input generation, we run this input in the main function in the ***arrange.py*** and will receive the output schedule from the path of file **result.csv**.

```

print(arrangeCourse('classroom.csv', 'course.csv', 'result.csv'))
```

Without new input generation, we can use the previous test case in the ***Testcase/Input*** file without new test case generation.

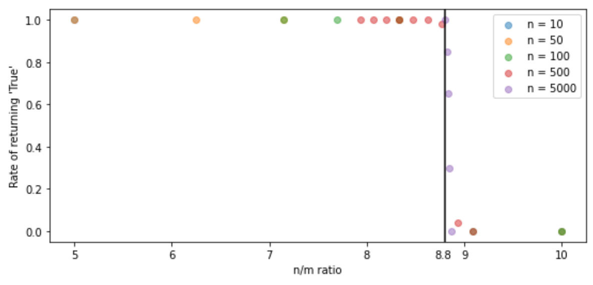
```

print(arrangeCourse('/.../algorithm_final_project/Test case/Input/
    classroom_case14n5000m566', '/.../algorithm_final_project/Test case/Input/
    course_case0n5000m556', '/.../algorithm_final_project/Test case/Output/
    case14n5000m566'))
```

4. Algorithm evaluation

This part will briefly describe the exact time of test case running and the suggestion for course scheduling problems in reality.

After running 10000 test cases using normal device, the algorithm only takes at most 0.01 s for small and medium test cases to generate the result of ***arrange.py*** function. For bigger cases with $n = 5000$, the running time is still smaller than 50 s. This analysis guarantee that this algorithm is applicable for larger cases in a bigger school or more constrains.

.



Moreover, the experiment with 10000 test cases suggests that for all pair (n, m) such that $\frac{n}{m} \leqslant 8.8$, it is confident that our algorithm will return a correct result with available timeslots (return True, and the corresponding schedule). Otherwise, if the rate $\frac{n}{m}$ is bigger than 8.8, the main function will return **False**, and the schedule is likely to include timeslots at night (back-up slots) or not have enough classes for all courses.