CS340 - The Registrar's Problem

Audrey Yang, Jacqueline McCarty, Juno Bartsch

September 30, 2022

1 Description

Initially, all classes $c = |C|$, time slots $t = |T|$, rooms $r = |R|$, professors $p = |P| = 0.5|C|$, and students $s = |S|$ are available. Assume that each class c_i has a professor p_i to teach it, where each professor can teach at most two courses. Also assume that $c > r$, but $c \leq rt$.

Organize classes in descending student preference, (i.e., the class that the most students have on their preference list is ranked first). Organize the rooms in decreasing size order. Let a "time conflict" be defined as when a room or professor is scheduled for two or more classes during one time slot.

Take the largest room that has yet to have all its time slots filled. Take the earliest time slot available for that room. Pick the most preferred unassigned class c_i.

Attempt to pair c_i and its professor p_i. If p_i cannot be paired with c_i (i.e., the professor is already teaching another class in this time slot), set this course aside. Check the next most preferred class c_{i+1} and repeat this process until an available course that can have a professor assigned to it is found, and assign that class-professor pair to this room and time slot. Add all classes that were set aside back into the priority queue of classes. In essence, this will effectively skip all classes that have a time conflict for this time slot.

Once a class has been assigned a professor, time slot, and room, fill the course with the students who prefer that class and do not have a time conflict for the chosen time slot until there is either no capacity in the room or all students who selected that course have been assigned. For every student who is assigned, increase the global student count by one. The global student count measures how many preferred classes the students end up enrolled in, and the overall optimality of the algorithm can be determined by dividing the global student count by 4s.

Continue the above process with each room, time, and class until all rooms have had all of their time slots assigned a class and a professor (or until there are no classes left), and the class has as many students as possible assigned to it. Output the assigned course lists as well as the student preference count.

2 Pseudocode

```
Function classSchedule(set of time slots T, set of room sizes R, set of classes C, set of professors P, set of student preference lists S)
    For any class, let preferredStudents[class] be the set of students with class on their preference lists
    Let classRankPriorityQ be a set of all classes organized by how many students have the class on their preference lists in descending order
    For any class, let classProfessor[class] be the professor capable of teaching class
    For all rooms, let maxRoomSizeQ be the set of all rooms organized by descending room size
    globalStudentCount = 0
    for each professor p_i in P do
        Let p_i.schedule be all the time slots for which p_i is scheduled for a class
        Set p_i.schedule = []
    end
    for each student s_i in S do
        Let s_i.schedule be the time slots for which s_i is scheduled for a class (i.e., attending a class)
        Set s_i.schedule = []
    end
    for each room in maxRoomSize do
        for each time slot ts in T do
            if classRankPriorityQ is empty then
                return globalStudentCount
            end
            class = classRankPriorityQ.extractMax
            while classProfessor[class] has a conflict for ts do
                append class to holdClass
                class = classRankPriorityQ.extractMax
            end
            Re-add all items in holdClass to classRankPriorityQ
            professor = classProfessor[class]
            add ts to professor.schedule (i.e., professor cannot teach another class at ts, it will be a conflict)
            Let students be a set of students enrolled in class
            while size(students) < room.size and preferredStudents is not empty do
                x = preferredStudents[class].extractFront
                if x does not have a time conflict at ts then
                    append x to students
                    increase globalStudentCount by one
                end
            end
            studentPreferenceCount = globalStudentCount/4s
            output class, ts, room, professor, students, studentPreferenceCount
        end
    end
```

3 Time Analysis

Recall that $|T| = t$, $|C| = c$, $|P| = p$, $|R| = r$, and $|S| = s$. Also recall that $p = c/2$, t is a small constant, and $c \leq rt$.

Initializing the classRankPriorityQ will involve adding c classes to a priority queue, for a total complexity of $O(c \log(c))$. Initializing maxRoomSizeQ will involve adding r rooms to a priority queue, for a total complexity of $O(r \log(r))$.

Thus, the *for each professor* p_i loop and *for each student* s_i loop can each be accomplished in $O(p) = O(0.5c)$ and $O(s)$ time respectively.

Consider the nested loops *for each room* and *for each time slot*. The first loop iterates through all rooms, and the second iterations through all time slots. Because $c \leq rt$ and both loops will be broken once classRankPriorityQ is empty, the number of iterations for this nested loop set is not $O(rt)$, but is $O(c)$. Each iteration sees exactly one class removed from the classRankPriorityQ. Though classes are re-appended to the queue as time conflicts are encountered, the number of classes extracted from the queue is always one more than the number re-appended, which allows the classRankPriorityQ to steadily decrement in size with each passing iteration. Thus, the nested for loops terminates after $O(c)$ iterations.

Inside the for loops, classRankPriorityQ.extractMax can be completed in $O(\log(c))$ time.

The first of the inner while loops – *while classProfessor[class] has a conflict* – can, at a maximum, iterate exactly $r - 1$ times. As there are only r possible rooms in which to teach, there can be at most r professors teaching at any one time. Whenever this loop is called, at least one of those rooms will be unassigned during this time slot. Thus, this loop will terminate after $O(r)$ iterations. Moreover, because classRankPriorityQ is a priority queue composed of all classes, extracting the most desired class will take $O(\log c)$ time.

Likewise, at maximum, only r classes can be held at any one time – one for each room. Therefore, $holdClass$ can be a maximum size of $r-1$, since the current room will be still unassigned when items are being appended to and removed from $holdClass$. Hence, there can be exactly $O(r)$ removals from $holdClass$ and appends to $classRankPriorityQ$. Since appending items and extractMax for a priority queue is $O(\log(r))$ each, adding and removing r items from $classRankPriorityQ$ is an $O(r \log(r))$ operation.

The second and final inner while loop – *while* $size(students) < room.size$ – in a worst case scenario, would have s students in preferredStudents, with all of them having time conflicts for *time*. In this scenario, this while loop would terminate after a maximum of $O(s)$ iterations.

Therefore, summing all of these complexities up, the time complexity of this algorithm is:

$$O(c \log(c) + r \log(r) + s + 0.5c + c(r \log(c) + r \log(r) + s)$$
$$\Rightarrow O(c \log(c) + r \log(r) + s + 0.5c + rc \log(c) + rc \log(r) + cs)$$
$$\approx O((r+1)c \log(c) + (c+1)r \log(r) + cs)$$
$$\approx O(rc \log(c) + rc \log(r) + cs)$$

However, since we have assumed that $r < c$, we can assume that $\log(r) < \log(c)$, and by extension, $rc \log(r) < rc \log(c)$. Thus:

$$complexity \approx O(rc\log(c) + cs)$$

In order for this algorithm to complete in $O(rc\log(c)+rc\log(r)+cs)$ time, the following operations must be completed in $O(1)$:

1. retrieving professor from classProfessor[class]
2. determining whether professor has a conflict for a time slot
3. adding and removing items to $holdClass$
4. adding a time to professor.schedule
5. extracting items from preferredStudents[class]
6. appending x to students

3.0.1 Data Structures

All (class, professor) pairings can be stored in an array $classProfessor$, where $classProfessor[class] = professor$. This means that (1) can be accomplished in $O(1)$ time.

Each professor's schedule can be stored in an array of linked lists, where each professor's schedule is indexed by their professor ID number, and each linked list consists of all times for which the professor is teaching a class (i.e., $schedule[professor_i] = [time_a, time_b]$). Since each professor can teach a maximum of two classes, each linked list can be of maximum length 2. Thus, running schedule[professor].contains(time) would involve comparing $time$ against only two other items in the list, accomplishing (2) in $O(1)$. Likewise, because finding the correct professor's schedule in an array is $O(1)$ and adding any item to a linked list is also $O(1)$, accomplishing (4) is an $O(1)$ operation.

$holdClass$ can be a first-in-first-out (FIFO) queue. This would accomplish (3) in $O(1)$ time.

$preferredStudents$ can be stored as an array of FIFO queues, where each $class$ is indexed by a class ID number to a location in the array, and each $class$'s corresponding FIFO queue is made up of all students who put down $class$ on their preference sheets. Finding the correct FIFO queue for $class$ would be an $O(1)$ operation. Likewise extracting items from the FIFO queue would be an $O(1)$ operation, accomplishing (5) in $O(1)$ time. Similarly, $preferredStudents$ could be initialized by iterating through each student preference list, and then appending the student to the correct $preferredStudents[class]$ FIFO queue for each class on each student's preference list. Since each preference list has a maximum of only 4 classes, this can be done in $O(4s) \approx O(s) < O(rc\log(c)+cs)$ time.

The final list of students enrolled in each class can be held in a FIFO queue. This would allow for (6) in $O(1)$ time.

4 Proof of Correctness

Proof. To prove that this algorithm works, four parts need to be shown.

1. proof of termination
2. proof that all classes are assigned
3. proof that no professor teaches more than one class for a single time slot
4. proof that no room is scheduled for more than one class for a single time slot
5. proof that on completion of the algorithm, any unscheduled courses cannot be scheduled

4.0.1 Proof of Termination:

Pseudocode and time analysis sections show that this algorithm will terminate in $O(rc\log(c) + cs)$ time. The first for loop (*for each professor*) will terminate in $p = 0.5c$ iterations, and the second for loop (*for each student*) in s iterations. Furthermore, the nested for loops must terminate after c iterations of the inner loop, as was shown in the time analysis section. The *while classProfessor[class] has a conflict* loop must terminate after at most $O(r)$ iterations, since there can be at most r professors teaching at any one time (one for each room). Finally, the *while size(students) < room.size and preferredStudents is not empty* loop must terminate in $O(s)$ iterations, as each iteration decreases the size of preferredStudents by one and the maximum size of preferredStudents is s.

4.0.2 Proof that no professor teaches more than one class for a single time slot:

Suppose some professor $prof$ teaches both class c_a and class c_b for some time slot $time_i$. This would imply that $prof$ had $time_i$ listed in $prof$.schedule, and then was assigned to c_b for time slot $time_i$ regardless. However, this is a contradiction, because as is stated in the pseudocode, if a professor has a time conflict, the algorithm will search through the classRankPriorityQ until it finds the next available most preferred class with an available professor. At this point, all classes stashed in $holdClass$ will be re-added to classRankPriorityQ.

Furthermore, once the algorithm reaches the next time slot $time_{i+1}$, c_b will once again be the most preferred class in classRankPriorityQ. Recall now that each professor is capable of teaching two classes. Because $prof$ is already teaching c_a during $time_i$, and c_b has yet to be assigned a time slot at this point in the algorithm, there is no other possible class that could conflict with c_b at $time_{i+1}$.

Thus, no professor can teach more than one class for a particular time slot.

4.0.3 Proof that no room is scheduled for more than one class for a single time slot:

Let ts_r represent "a time slot for a room r." Pseudocode shows that for each room r, each time slot ts is iterated through, and for each ts_r, a class is assigned to it. Time slots never backtrack, and only one class is ever assigned during one iteration. Thus, no two classes can be scheduled for a single room for a single time slot.

Both of these amount to a contradiction, so no room has two classes assigned to any single time slot.

4.0.4 Proof that on completion of the algorithm, any unscheduled courses cannot be scheduled (by contradiction)

Suppose that on completion of the algorithm, an arbitrary course c has been unscheduled and it could have been scheduled with an arbitrary room r and time slot t. For this case to have occurred, the algorithm must have not scheduled this course in that available slot. However, the algorithm works by adding an unscheduled class to the classRankPriorityQ and, once extracted, checking it against every succeeding room/time-slot combination until the algorithm finds a class/room/time-slot combination that does not have a conflict. This means that there is either a conflict for the course not to be added or a time slot was skipped, which is impossible by construction of the greedy algorithm. Both cases result in contradiction.

□

5 Discussion

Why did you choose this algorithm?

This algorithm was designed to maximize student preferences while also running in a reasonable time frame. Our greedy approach assigns classes to the largest possible rooms in order of student preferences to ensure that as many students as possible are able to take the classes they prefer. Iterating by most preferred courses also made it simple when assigning room sizes, as the biggest rooms would go to the courses that were preferred by the most students.

What complications did you encounter while creating it?

We struggled with professor conflicts, which diminish the success of the greedy approach. Specifically, we wanted to ensure that there is no back-tracking. We created holdClass to ensure that when professor conflicts arise, the courses are put into the next-optimal place, if such a slot is available.

What characteristics of the problem made it hard to create an algorithm for?

We had originally considered a 'snaking' method to ensure that classes preferred by less students were matched with those preferred more. We wanted the algorithm to be similar to games in ranked sports where 1st place plays 16th place, 2nd with 15th, and so on. This would ensure that student time conflicts were minimized. Take a hypothetical case where there are 75 students, all of whom want an intro bio lecture, 15 of whom also want to take art history, and 25 want to take computer science. If bio and art history were scheduled in the same block, there would be 15 time conflicts, but if CS was used in place of art history, there would be an additional 10 conflicts. It is unclear whether this approach would be worth the time to implement.

What algorithmic category or categories does your algorithm fall into?

This is a greedy algorithm optimized by student preferences and it does not back-track.

Which algorithms that we have studied is your algorithm similar to?

Our algorithm is similar to other greedy algorithms such as the box-stacking problem. We attempt to optimize between multiple factors (matching high student preferences with larger sized classrooms) in order to achieve the highest fit.