CS340 - The Registrar's Problem

Ary Wilson, Sidd Phatak, Mihir Patel

August 27, 2025

1 Description

The algorithm assigns rooms to classes based on the number of people who want to take each class. It then assigns professors and classes to time slots if the professor is available at that time and the class' room is currently unassigned to a time. Finally, it assigns students to classes if doing so doesn't create a conflict.

2 Pseudocode

```
Function schedule
    Let the popularity rating of a class be the number of times each class appears
      on any student's preference list
    Sort on C by popularity, descending
    Sort on R by room size, descending
    let $idx$ be 0
    foreach room $i$ in $R$ from largest to smallest room size do
        for $t$ iterations, with $t$ as the length of $T$ do
            if $idx$ is less than the number of classes then
                Assign class in $C$ at index $idx$ to room $i$
                increment $idx$ by 1
            end
        end
    end
    foreach professor $p$ in $P$ do
        foreach class $c$ that $p$ teaches do
            foreach timeslot $t$ in professor $p$'s list of times do
                if class $c$'s room $r$ is still unassigned to a time then
                    schedule room $r$ to time $t$, and class $c$ to time $t$
                    remove $t$ from professor $p$'s list of times
                    break out of this loop
                end
            end
        end
    end
    foreach student $s$ in $S$ do
        foreach class $c$ in the preference list of $s$ do
            if class has available spots AND class time is not yet in student object
              then
                add class time to student object, and add student to class object
            end
        end
    end
```

3 Time Analysis

3.1 Variable and Data Structure Definitions

Given:
C: a list of classes, $|C| = c$
a class object holds

1. a class id number
2. a teacher
3. a room id number
4. a timeslot
5. a list of enrolled students
6. a popularity rating

R: a list of classroom sizes holds the capacity of each room, $|R| = r$
Room objects hold:

1. room name/id
2. room capacity
3. the list of classes in that room

T: list of all time slots $|T| = t$
P: a list of all professors and the names of the classes they teach, $|P| = p$
Professor objects hold:

1. the professor's id
2. a list of the classes they can teach
3. a list of the times they are teaching

S: a list of students $|S| = s$
Student objects hold:

1. a student id number
2. an array of class preferences of size 4, each element is a link to a class object
3. an array of timeslots the student is currently enrolled in

3.2 Analysis

Each of the items listed below show the different time complexities of different parts of this algorithm. If we can find all of these different complexities and add them together at the end, we have found the time complexity of the function.

1. Counting the number of times a class appears on the preference list will take $4s$ iterations, if s is the number of students, as each students will put 4 classes on their preference lists (**O(4s)**). Sorting the classes in C will take **O(clogc)**, because sorting has a well established complexity. Similarly, sorting R by room size will take **O(rlogr)**. Traversing through the room in R will have r iterations, and for each we do t class assignments, giving us a complexity of **O(rt)**.
2. Looping through all the professors takes p iterations, there are two classes per professor, and looping through each time slot is t iterations. Furthermore, to remove an element from the professor's list of times (which has length t) will take $O(t)$. Thus the total complexity of the nested for loops is $O(pt^2)$.
3. Finally, looping through each student in S and looking at their preference lists will have complexity $O(4s)$.

Thus, the total complexity of this algorithm will be

$$O(4s + clogc + rlogr + rt + pt^2 + 4s)$$

We know that at the Bi-Co, $p = c/2$, so $pt^2 = \frac{1}{2}ct^2$ (and in big O, we can ignore the constant). Even when we have a small number of time slots, say $t = 8$, for $logr = t = 8$, this would imply that $r = 2^8$. Because log reduces so quickly, $t > logr$ and $t^2 > t > logc$ (meaning $rt > rlogr$ and $ct^2 > clogc$). Thus we have:

$$O(8s + rt + ct^2)$$

Since at the Bi-Co, $s \approx c/2$, $8s \approx 16c$, which is linear, and thus dominated by ct^2:

$$O(rt + ct^2)$$

In the likely case that there are fewer rooms than total classes, (r < c), the complexity of this function would simply be

$$O(ct^2)$$

For this time analysis to be correct, the following must be performed in constant time:

1. Assigning a certain class to a room
2. Checking if a certain class is assigned to a time
3. Assigning a class and room to a time
4. Checking if a class has available spots
5. Getting/setting a class time to a student object
6. Add a student to a class object

3.3 Data Structures Implementation

For (1), we have defined our class objects as containing a room. All we need to do to match a class to a room is store the room id at this place within the class object (and assigning an attribute is constant time).

For (2), we can create a 2D array, with one dimension as rooms and the other as times. If the value stored at [r][t]=0, then there is no class in room r at time t, otherwise we store the class id of the class meeting there and then. Classes store room ids, so to see if a class or a room has a time assignment is a constant time check because we know the two indices we want to access in the 2D array. Then for (3), to assign a class or room to a time slot is just to change the value stored at the index in the 2D array to the class id, also O(1).

For (4), to check if a class has an available spot, all we need to do is compare its room size to enrolled students. Classes store a room object, which stores its capacity (accessing this attribute is O(1)). Classes also keep a list of currently enrolled students, and calculating the length of this list is constant time as well. Thus all that is needed to see if a class still has available spots is to check if room capacity is greater than number of enrolled students. As long as it is, the class has available spots.

For (5), to get and set a class or classtime to a student object is also O(1). Student objects will store a list of the timeslots of classes they are currently enrolled in. To check if a classtime is in a student object requires at most 4 checks, because a student can only be enrolled in 4 classes. To add a classtime to a student is just to append the time to the list attribute of the student, which is still constant time.

For (6), to add a student to a class object, all we need to do is append the student to the list of enrolled students stored inside the class object (which is O(1)).

Therefore, the above time analysis is valid.

4 Proof of Correctness

Proof of Termination. schedule() clearly terminates because all of the loops inside of it terminate at some point. All these loops traverse different lists (whether its the list of classes, a student's preference list, the list of rooms, etc.) without backtracking at all, and all these lists have a finite number of elements. Thus this algorithm can only run for a finite number of iterations. □

Proof of Validity.

1. no teacher time conflict
2. no room time conflict
3. all schedulable classes are scheduled
4. no classes are scheduled more than once
5. no enrolled student has a schedule conflict

1. Our algorithm only assigns a teacher to a time in their current availability list, removing that time slot once this is done. As teachers only teach two classes, when assigning the second class, none of the time slots in the professor's time list will conflict with the first class assigned. Therefore, a teacher time conflict cannot exist.
2. There is no room conflict by construction of the algorithm. Since classes are assigned to rooms before times are assigned, and times are assigned by room. The algorithm uses a 2D array to keep track of room-time assignments, so if we want to assign a class and its corresponding room to a time, we will only do so if there is not already a class in the same room and time. Therefore there can be no room time conflicts.
3. Suppose all classes are not scheduled. That implies that there exists some class c that is not assigned to a room or a time. However, in the first for loop we fully iterate through C (by going through R and T), assigning every class to a room. Furthermore, while iterating through the professors, we cover every class in C (as every class is taught by a professor). Because classes can't get unscheduled from time slots, and we schedule every class when it is unassigned, eventually we will end up with all classes scheduled.
4. Claim: For all i $\leq$ c, c_i is not assigned multiple times (to multiple rooms). Suppose c_i is assigned to multiple rooms. However, every time a class is assigned, we increment our *idx* counting variable, and the next time around we will thus access the next class (and then assign it, etc.). It is therefore not possible for a class to be assigned to two or more rooms, because after a class is assigned to one room, we move on and never backtrack.

5. Suppose an enrolled student has a schedule conflict. This means that a student was enrolled in a class at a certain timeslot t_0 when the student is already enrolled in a different class at t_0. However, the conditional in the last for loop of *schedule*() explicitly states not to pair a student with a class if they are already enrolled in a class at the same time. Therefore, a student cannot have a scheduling conflict.

□

5 Additional Constraints

We chose to implement 5 additional constraints into our algorithm:

1. Classes must be taught in their assigned department rooms. In reality, a class is not assigned to any of the available rooms. Typically, Math classes will always be in a set of Math classrooms (the Math building), and English classes will be in English classrooms (the English building).
2. Minimize overlap between classes in the same department. It is not practical to schedule all Math classes at the same time(s), because then it becomes difficult for a Math major to take multiple required Math classes in the same semester. Instead there should be minimal overlap between same department classes so that majors can take as many classes for their major as necessary.
3. Students cannot take classes in adjacent time slots at different universities. It is difficult for a student to take a class at Haverford and Bryn Mawr if they start/end at the same time, as transportation between campuses takes a certain amount of a time.
4. The t most popular classes are taught over Zoom, so that room capacity does not prohibit students from enrolling in the class.
5. Classes with the same course title cannot be scheduled at the same time if possible. For classes with multiple sections (such as elementary language or lower-level Math classes), we want to prevent those sections from being scheduled at the same time. This would allow a student to still take a desired class even if one section creates a time conflict.

For (1) and (2), we will use the fact that every class has a small list of rooms that it can be assigned to (its department building). Instead of how we initially assign classes to rooms, we can now do the following: This will go through each class and find the biggest room that it is allowed to be taught in, as long as there are not already t classes in that room. This achieves both (1) and (2), because it prevents classes from being placed in rooms that they are not supposed to be in, and it also puts the t biggest classes (assuming for a department) all in that department's biggest room. This would space out the classes for

```
foreach class c in C from most to least popular do
    foreach room i in R from largest to smallest room size do
        if room i is one of c's possible rooms AND room i has fewer than t classes then
            Assign class c to room i
        end
    end
end
```

that department/major, minimizing the overlap between those classes. In terms of running time, we are looping through each class, and in worst case through all of R per iteration. Furthermore, checking if room i is in c's possible rooms is R iterations at worst, because it is possible that c can be taught in all r rooms. Thus we have a complexity of $O(cr^2)$, which would make the entire algorithm that complexity as well.

For (3), we can use the fact that student objects store the classes they are currently enrolled in, as well as the times those classes are taught. When we want to assign a student to a class, we also need to check if any of the classes currently enrolled in (max 3) have a time slot that is adjacent to the current time slot we are trying to enroll in. If they do, and the class in those adjacent time slots is at a different university than the class we are trying to enroll in, then we do not enroll in that class (and we do enroll if at the same university). The course catalog we are given stores course names, which include the first letter of university it is taught at (e.g. B340), and we can loop through the list of classes once and store within the objects which university it is taught at. We would modify the code as:

```
foreach student s in S do
    foreach class c in the preference list of s do
        if class has available spots AND class time is not yet in student object
          AND previous,next class time are not at a different college then
            add class time to student object, and add student to class object
        end
    end
end
```

This modification would not change the complexity of this step, as we have to check max 3 other classes, what times they are taught, and what universities they are taught at (both constant time checks because they are attributes stored within the class object).

For (4), in pre-processing we can add one room to R, with size of s (total students in the school). Once we sort R, this room will be at the front of R, as it is the biggest room. If we want to comply with the department constraint, we should add this room to the possible rooms of every class as well (O(c)). Thus when we are assigning classes to rooms, the t biggest classes will all be put in this "room" (which is really just a zoom meeting). As a result, when assigning students to classes, more students will get into these classes, as we will never fail the conditional that class does not have available spots. Doing all of this will be constant time complexity (O(c) if we comply with department constraint), as we are only adding one element to a list, and not changing any of our algorithm other than that pre-processing.

For (5), we implement this constraint via dynamic programming – by checking all possible combinations of adding classes to the schedule with these three constraints:

1. classes in the same timeslot are not taught by the same professor
2. classes in the same timeslot are not in the same room
3. no two classes in the same timeslot have the same course title, as much as possible

For each class, the function attempts to add that class to all valid timeslots in the schedule that satisfy the three conditions above. Once that class is added to that timeslot, the function recursively calls itself on the classes minus the class that was just added and the updated schedule with the class that was just added into the schedule. It continues to do this until there are no more classes to add, at which point the schedule is either filled OR there are no valid timeslots to put a particular class in.

If the former happens, the function simply returns the schedule. If the latter happens, we know the schedule generated so far will not result in a valid schedule, and thus an empty schedule is returned. If an empty schedule is returned from a recursive call within the function, the function will simply move on to the next valid timeslot for the given class. We repeat the process until there are no valid timeslots left, OR a recursive call in a valid timeslot returns a valid schedule.

The time analysis for this function would be $T(c, r, t) = crt + ctT(c-1, r, t)$. This time analysis makes sense because each recursive call (done ct times) is done on an input size only one class smaller than the previous recursive call and the work done outside of the recursion is crt. Since each recursive call takes $(c-1)rt + (c-1)T(c-2, r, t)$ iterations as well, it is clear that continuous back substitution will result in a time complexity that looks something like $T(c, r, t) = crt + kc^2t^2r$ which is in $O(c^2t^2r)$.

6 Experimental Analysis

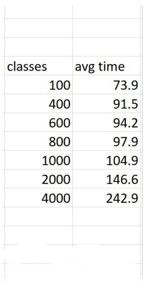
Our experimental analysis was conducted on 17 differently sized inputs. For each size, we generated 10 random inputs and ran our algorithm on each. The below table shows the input sizes, the average time it took to run each trial, and the average fit over each trial. Each of these unique data sets is identical to one other data set except for the value of one variable.

The experimental time analysis demonstrates that our theoretical analysis is slightly wrong. If you were to view the graphs for Time vs Classes and Time vs Students, it is clear that the relationship is linear between time and Classes as well as between time and Students. This makes sense because the number of students in each test case is strictly equal to two times the number of classes. Thus, if one is linear in relation to time, the other one will be too.

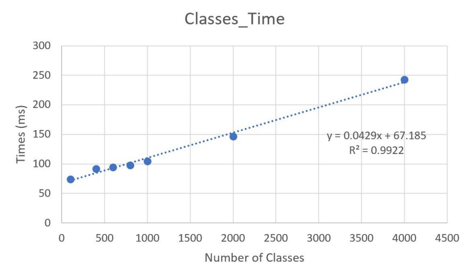
The graph of Time vs Rooms has a somewhat low R^2 value and a very low slope. The graph of Time vs Timeslots has a very low R^2 value regardless of the regression scheme. These two observations suggest that changing the number of rooms and timeslots does not significantly affect the running time of the algorithm, the time only changes in these graphs when we hold the value of room/timeslot constant and modify other variables. This makes sense, as neither t or r make it into our big O time analysis. In fact, our time analysis shows that the time complexity of this function in relation to c is clogc because of an initial sort of C based off of class popularity. However, it is possible to do this sort in linear time because class popularity is an integer and so Counting Sort, a 7 Figure 1: Table 1 linear sorting algorithm, can be implemented. Thus, instead of this sort taking clogc time it takes O(s) (which grows at the same rate at O(c)), because the maximum value for a class's popularity is s (if every student wants to take that class), and so a linear time sort will depend on the value of the maximum number. Thus O(s) is the dominant term in the final analysis (assuming our rlogr will also get sorted linearly). This linear relation can not only be seen in the class and student graphs but also in the table of total algorithm running times.

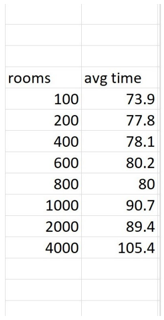
In order to calculate the score, all that was done was to divide the given "Student preferences value" from is valid by 4s, the total number of classes on all student's preference lists. The worst-case scenario for this algorithm is if all of the students have the same preference list of four classes and if all of those four classes occur at the same time. If this is the case, then each student will get at most one of the classes of their preference list, while some students may miss out on one of those four classes altogether if all of the classes get full. Thus, the score would be less than or equal to 25%. However, these kinds of cases are rare and not likely to come up when the algorithm is actually being used so we did not account for these kinds of edge cases when designing the algorithm - some constraints are not realistic in the real world. Experimentally, the minimum bound of the algorithm seems to be around 0.74, though we did not stress test it to see how much lower it could go for reasonable constraints. All in all, the average score of our randomly generated data was .837 which suggests that on average 83.7% classes on the preference list of the students were assigned to them.

6.1 Time Complexity

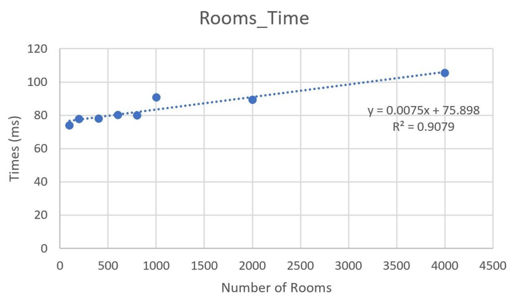


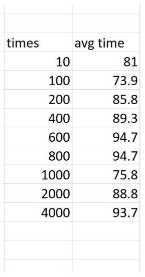
classes	avg time
100	73.9
400	91.5
600	94.2
800	97.9
1000	104.9
2000	146.6
4000	242.9



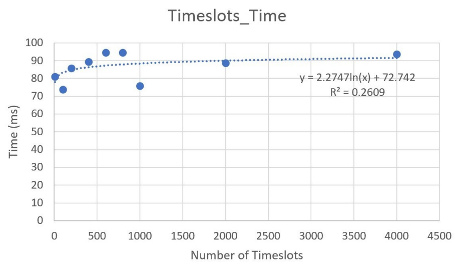


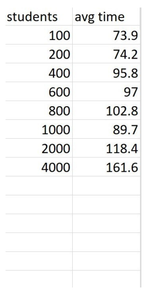
rooms	avg time
100	73.9
200	77.8
400	78.1
600	80.2
800	80
1000	90.7
2000	89.4
4000	105.4



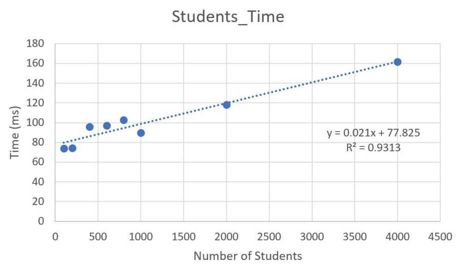


times	avg time
10	81
100	73.9
200	85.8
400	89.3
600	94.7
800	94.7
1000	75.8
2000	88.8
4000	93.7





students	avg time
100	73.9
200	74.2
400	95.8
600	97
800	102.8
1000	89.7
2000	118.4
4000	161.6



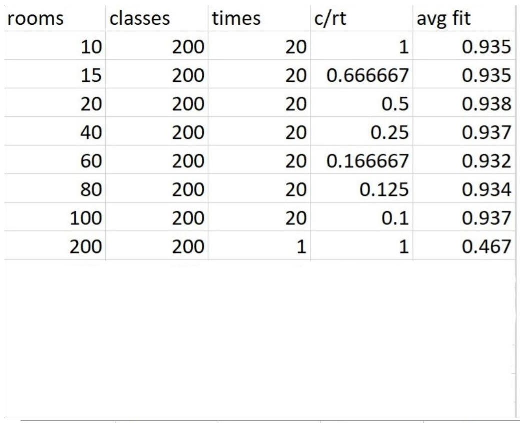
6.2 Stress Testing

Stress testing was done by generating 10 different inputs of the same dimension. In the first graph, we generated 7 unique sets of inputs where the number of rooms decreased until c/rt = 1. In the second graph, we generated 12 unique sets of inputs where the number of timeslots decreased until c/rt = 1. The average fits are displayed against c/rt for each table.

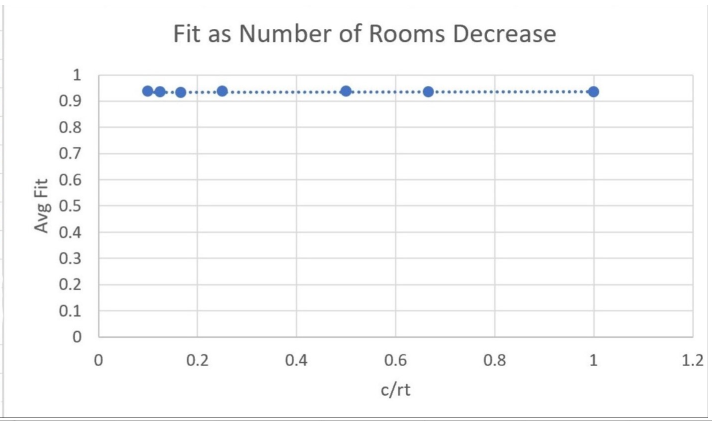
From the graphs, we see when c approaches rt by decreasing the number of rooms, the fit remains above .9, however, when c approaches rt by decreasing the number of time slots, the fit drastically decreases (with a fit of .4 - .5, when t = 1 or 2). This makes sense because rooms are assigned based on popularity, and times are, besides ensuring a valid schedule, assigned randomly. With fewer time slots this leads to a lot more conflict because there is no design minimizing student conflict in regards to timeslots.

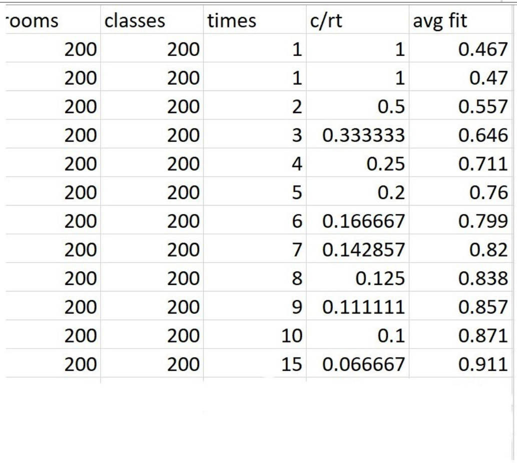
When looking at individual schedules we find students do not get classes because of an internal schedule conflict. That is, a student fails to add a class to their schedule because

the class's preassigned time conflicts with the times of the classes the student already is signed up for. Students are randomly assigned time slots without any concern for when their other classes take place. With only a few time slots (2-4), a student is going to run into 2 conflicts around half the time, for an average fit of 50%.
This problem may be mitigated by introducing a conflict graph to assign times, however, the basic algorithm shouldn't have much of a problem as long as t > 4.

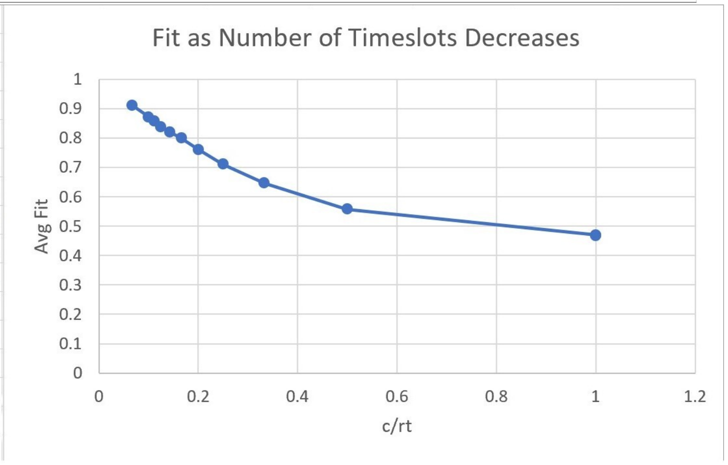


rooms	classes	times	c/rt	avg fit
10	200	20	1	0.935
15	200	20	0.666667	0.935
20	200	20	0.5	0.938
40	200	20	0.25	0.937
60	200	20	0.166667	0.932
80	200	20	0.125	0.934
100	200	20	0.1	0.937
200	200	1	1	0.467





rooms	classes	times	c/rt	avg fit
200	200	1	1	0.467
200	200	1	1	0.47
200	200	2	0.5	0.557
200	200	3	0.333333	0.646
200	200	4	0.25	0.711
200	200	5	0.2	0.76
200	200	6	0.166667	0.799
200	200	7	0.142857	0.82
200	200	8	0.125	0.838
200	200	9	0.111111	0.857
200	200	10	0.1	0.871
200	200	15	0.066667	0.911



7 Recommendations

We determined an implementable recommendation for the registrar – allowing popular classes to be taught on Zoom. Obviously some quality of teaching may be lost over Zoom, but this allows classes that are very desired/popular to have as many students as possible (giving more students their ideal schedule).

Furthermore, when we implemented the assigned department room constraint (1), the preference values are higher when including the building constraint than they are without it. This means it's better to assign the most popular course to the largest room within each department than it is to assign the most popular classes to the largest room in the entire college. Looking at the schedules produced, both algorithms scheduled the same number of classes, but there were more students in each class. This means that time conflicts limit schedules more than room capacity. However, discrete room assignments allow less conflict, because if a student takes one class in a department they are more likely to take another class in that same department. Assigning rooms this way decreases time conflict because classes assigned to the same room will need to be at different times, and it spreads out the classes in the same department evenly over the different time slots.

Therefore we also recommend that classrooms are assigned based on their department (or more generally, that classes with high conflict are scheduled at different times).

8 Discussion

The number of variables made the problem difficult to approach in terms of keeping everything organized. It also made designing the algorithm difficult. We had trouble deciding in what order to assign students, rooms, and times to classes. Assigning rooms and times before students was an appealing option because we could easily check a student's schedule before adding them to a class. We decided to first determine the schedule and then assign students to avoid backtracking. Without knowing the size of the room or the time of the class, a student could be added to a class and later need to be removed.

Because student preference lists are not weighted, we decided to assign rooms based on the number of times a class appeared on a preference list, which allowed us to get closer to optimality and check if a schedule is valid before adding a student. Originally to assign time slots, we wanted to group classes that shared the fewest potential students with a graph. We planned to implement this as a graph adjacency list and assign times based on the k-clustering alteration of Kruskal's algorithm. We changed our algorithm after encountering problems when implementing it as a conflict matrix instead. The union-find classes we imported didn't have everything needed to implement our algorithm. Additionally, the time preference lists were based on the ability to 'remove' an edge from a graph, which is more challenging to ensure with a matrix implementation.

However, there were also some fundamental problems with how we planned to use the conflict matrix. The original pseudocode unhelpfully assigned each class the same time preference list, which we didn't realize until after we implemented it. In future implementations, we may use the conflict matrix differently, but it was not used for this analysis.

Instead, we implemented the popularity-based assignment of rooms, and randomly assigned times. The simplified time assignments allowed us to prevent multiple classes from being assigned to rooms at the same time. It also decreased the time complexity. The assignment of rooms is greedy and based on the criteria determined in the preprocessing steps. The assignment of times and students is random, aside from ensuring schedule validity.