CS340 - Scheduling Project

David Dinh and William Lawrence

November 15, 2021

1 Project

1.1 Abstract

For our initial algorithm we simply chose a greedy popularity algorithm. The most popular courses were assigned to the biggest rooms then we assigned students to the classes within their preference lists. Alongside our initial greedy-popularity based algorithm we also created and tested 5 extensions for the algorithm during which we found that for all extensions – except for extension 3 where classes are assigned to rooms based on their subject and extension 5 where students choose classes in a first come first serve model – there is an increase in average optimality percentage. Extension 3 showed a drastic decrease in average optimality and our recommendation based on this would be that lectures should be assigned to any possible room regardless of the subject of classes. Only labs where specific equipment is required should have building constraints.

Looking at Extensions 1 and 2 we saw an increase in the average optimality throughout the semesters. As a result of Extension 1, our recommendation is to spread classes within the same subject across timeslots to help lessen student conflicts. For Extension 2, our recommendation for the registrar is to spread classes of the same levels across timeslots. For Extension 4 (Increasing Room Capacity), our recommendation for the registrar is to increase the amount of seats available for each room. For Extension 5 (First Come First Serve) although the resulting optimality scores decreased throughout the different semesters, the optimality score was not meant to be indicative of the success of the algorithm but instead just shows the percentage of students who gained their desired course. The metric of success is then instead not based on the total preference value but on individual preference. The results garnered from extension 5 do not lead to a conclusive recommendation due to how success or optimality is measured.

1.2 Description

Given a set of students, classes, professors, timeslots, and rooms, we strive to create the most optimal schedule given these constraints. We define an optimal schedule as one where each student gets as many preferred classes as possible. Knowing that the maximum number of classes taught is the number of rooms multiplied by the number of timeslots, we consider putting the most popular classes into the biggest rooms first so that we allow a substantial amount of students to take their preferred classes.

However, we might run into a variety of conflicts. One possible conflict that we might consider is putting a class into a timeslot wherein the professor is already teaching another class at that same timeslot. We then consider the next most popular class instead. Another conflict is that the number of students who want to take the class considered may exceed the room size. In this case, we simply assign the class to the room and timeslot, and enroll the number of students equal to the room capacity. The last conflict we consider is student conflict, where a large number of students who want to take the course are already scheduled for another course at that particular timeslot. In this case, we simply remove the students who have the timeslot conflict from the class. Working around these conflicts, we are able to create a schedule that is viable and based on the popularity of classes.

1.3 Pseudocode

class object holds the class ID, students to be enrolled in the course, professor assigned to the class, and room and timeslot that the class is assigned to in the finalized schedule

```
Function classToAssign(room, timeslot, classPref, schedule, pTimeslots, sTimeslots):
    for each class in classPref do
        if pTimeslots[professor of class] already has timeslot continue onto next class
        assign room and timeslot schedule[class]
        assign timeslot to pTimeslots[professor of class]
        for each student in class do
            check if sTimeslots[student] already has timeslot and if student was already
              assigned to timeslot, pop student from class
        end
        if class size exceeds room size then
            remove the excess students at the end of class so that class size is equal to room
              size
        end
        for each student in class do
            add timeslot to sTimeslots[student]
        end
        pop class from classPref
        return
    end
    return
```

```
1  Function scheduler(R, S, P, T, C):
2      add each class ID $\in C$ and corresponding professor $\in P$ to schedule

3      let classPref be set of classes which have sets of students who prefer to take that
         particular course
4      for each student $\in S$ do
5          for each of student's preferred classes called prefClass do
6              add student to classPref[prefClass]
7          end
8      end
9      sort classPref by highest to lowest number of students in each class set
10     sort $R$ by highest to lowest size

11     let pTimeslots be an array of professors and timeslots they are currently assigned to
12     let sTimeslots be an array of students and timeslots they are currently assigned to

13     for each room $\in R$ do
14         for each timeslot $\in T$ do
15             classToAssign(room, timeslot, classPref, schedule, pTimeslots, sTimeslots, P)
16         end
17     end
18     return schedule
```

1.4 Time Analysis

Given inputs R, S, P, T and C, let $r = |R|$, $s = |S|$, $p = |P|$, $t = |T|$, $c = |C|$. We will run the function *scheduler* first, which will initialize schedule, classPref, pTimeslots, and sTimeslots. In the data structures section, we see that constructing schedule takes O(c), constructing classPref takes O(c + s), constructing pTimeslots takes O(p) and constructing sTimeslots takes O(s). Then, we see that we also sort classPref which takes O(clogc) and sort R which takes O(rlogr). Then, we have to consider how many times the function classToAssign is called on line 15 of function scheduler. We see that classToAssign is called O(r*t) times because we call classToAssign for each room and timeslot. Now, we must analyze the complexity of the function classToAssign.

For classToAssign we must analyze how many classes we consider before finalizing the class into schedule. We see that the only time a class cannot be finalized is when a professor assigned to the class considered was already teaching a class at that timeslot. Therefore, we see that we will go through at most p classes before either deciding to either finalize the class at the timeslot and room or not assign any class to the considered timeslot and room. Thus, we consider at most $O(p)$ classes per line 2 in each call of classToAssign. Afterwards, we assign the timeslot and room values to the class in schedule, which takes $O(1)$. We also assign the timeslot to the professor in pTimeslots which takes $O(1)$.

For the class currently being considered, we have to iterate through all the students who prefer that particular class and check for student conflict. A class can have at most s students who prefer the course and each student can have at most 3 timeslots before enrolling in their last course. In the data structures section, we see that we can access sTimeslots in $O(1)$ but popping an element from class takes time proportional to the amount of elements in class. We see that there can be at most s students who prefer each class, so each pop takes at most $O(s)$. Therefore the for loop on line 6 takes $O(3 * s * s) = O(s^2)$.

We then must consider the if statement on line 9, where the class size might exceed the room size. Well, we see that we might have to remove $O(s)$ amount of students before achieving a class size that is equal to room size. By list slicing, we are able to remove the students in $O(s)$ which is proportional to the amount of students in class.

We must also update each student's timeslots, *sTimeslots*, to reflect that they were assigned to the timeslot associated with the class. We can update at most s students and appending a timeslot to a list takes $O(1)$. Therefore, the for loop on line 12 takes $O(s)$.

Lastly, we pop the class we finalized from classPref which takes time proportional to the number of classes in classPref, which means that line 15 takes $O(c)$. Thus, we see that the time complexity of classToAssign function is proportional to the amount of classes we consider, time it takes to check for student conflict and pop students, time it takes to check for room size conflict and slice the student list, time it takes to assign timeslots to students, and popping the class from classPref. Thus we see that the classToAssign function takes $O(p + s^2 + s + s + c) = O(s^2)$. We see that checking for student conflict dominates as it takes quadratic time relative to the number of students. Therefore, we see that each call to classToAssign takes $O(s^2)$.

Then, looking back at the entire algorithm, we see that classToAssign is called t*r times. So, the for loop on line 13 of function scheduler takes $O(t * r * s^2)$ before terminating. Since we see that constructing the data structures and sorting classPref and R takes at most $O(clogc)$ or $O(rlogr)$, we see that the calls to classToAssign dominates as we have a quadratic complexity relative to the student times the number of rooms and timeslots, which grows faster than logarithmic functions. Thus, we see that the algorithm takes $O(t * r * s^2)$.

1.5 Data Structures

Inputs R, S, P, T, C are all arrays of size r, s, p, t, and c respectively.

We represent schedule by an array of class objects to be finalized, wherein the ID of the classes corresponds to the indices of the array. This takes $O(c)$ to build because there are c courses to be considered.

We then represent classPref as an array of 2-element arrays. The outer array simply is a container for all the classes. The first element in the 2-element array corresponds to each class ID in C. The second element in the 2-element array will correspond to a list of students that have that class on their preference list. Before sorting, the indices of the container array will match up with the first elements of the 2-element array, which means the indices of the container initially correspond to the class IDs. We see that the container array takes $O(c)$ to construct, and each 2-element array takes constant time for the first element, while the second element takes time proportional to the number of students who prefer that particular class. To construct the list, we iterate through all students and their preference lists in S, and put students in the corresponding second element of the array that corresponds to the class they prefer. Thus, we see that for every 2-element array, the number of total students across all classes will be *4s* because each student only has 4 preferred classes. Therefore, we see that classPref takes $O(c + c + 4s) = O(c + s)$ because the container array takes $O(c)$, the first element of the 2-element array takes $O(c)$ in total, and the total time for all linked list of students takes $O(4s)$. We will also keep track of indices, so we can easily find the size of each linked list and number of preferences each class has in constant time. So, we can sort classPref by highest to lowest preference value (denoted by the size of each linked list) of each class. The sort takes $O(clogc)$ because we can find the preference values of each class by taking the size of each linked list in constant time. We then see that classPref takes, overall, $O(c + s) + O(clogc) = O(clogc)$.

For input R, we sort the room by decreasing room *size*, which takes $O(rlogr)$.

For *pTimeslots*, we initialize an array of lists. The indices of the array corresponds to the professor and the lists at each professor corresponds to the timeslots that the professor is assigned to. Thus, the construction of pTimeslots takes $O(2 * p) = O(p)$ as each professor can have at most two timeslots. We can access and assign values for pTimeslots in constant time.

We represent *sTimeslots* as an array of lists. The indices of the array corresponds to the students and the lists at each student corresponds to the timeslots that the student is assigned to. Thus, the construction of sTimeslots takes $O(4 * s) = O(s)$ because each student can have at most 4 timeslots. We can access and assign values for sTimeslots in constant time.

1.6 Discussion

Proof of Termination: Since we know that the inputs R, S, P, T, C are finite, we are able to trace the termination of each loop. On line 4, since the number of students is finite and each student has 4 preferences, we see that the for loop on line 4 terminates after 4s iterations. Clearly, sorting and creating arrays are operations that terminate. Then, we see that the main function scheduler ends after we call classToAssign for each room and timeslot. We see that classToAssign is called r*t times and each call of classToAssign ends after we check for professor conflicts, student conflicts, room size conflict, and assign timeslots to students. In the time analysis, we saw that classToAssign does $O(s^2)$ work before terminating, and so, since the number of students is finite callToAssign also terminates. Therefore, we see that the for loop on line 13 exits and the algorithm terminates after $O(trs^2)$ work done in that for loop.

Proof of Validity (by contradiction): Assume that the schedule returned is not valid. This means that either the schedule 1) assigned a professor to teach two classes at the same timeslot, 2) that a student is taking two classes at the same timeslot, 3) the number of students exceed the class room size, 4) classes are scheduled more than once, or 5) not all schedulable classes are scheduled.

In case 1), assume that the schedule has a teacher assigned to two classes in the same timeslot.

In the pseudocode, line 3 of function *classToAssign* states that the algorithm does not consider a class to be assigned for a timeslot if the professor was already assigned to a class that was already finalized at that timeslot. Therefore, a professor can never teach twice at the same timeslot.

In case 2), assume that the schedule has a student that is taking two classes at the same timeslot. On line 6 and 7 of function *classToAssign*, we see that if a student considered in the finalized course is already taking a class at the same timeslot, then we pop the student from the class. Therefore, a student cannot take two classes at the same timeslot.

In case 3), assume that the schedule returns a class where the number of students enrolled exceeds the class room size. In line 9 of function *classToAssign*, we see that if the class size exceeds the room size, then we remove students from the class until we have the same number of students as the capacity of the room. Therefore, we cannot have more students enrolled in a class than the room size assigned to that class.

In case 4), assume that the algorithm scheduled a class c in more than one timeslot. However, the algorithm says that we never consider a class again once they have been finalized, therefore we cannot have a class assigned to multiple timeslots. This is evident on line 15 of function *classToAssign* where we pop the finalized class.

In case 5), assume that there exists an open timeslot and room but also a class c that remains but also fits in an open timeslot and room after the algorithm terminates. Since class c remains, then we know that class c was never popped from classPref. By line 2 of the pseudocode, we see that we consider the most popular classes until we get a valid class. Therefore, if c was a valid class for an available timeslot and room, then c would have been assigned that timeslot and room, and then popped from classPref, resulting in a contradiction. Thus, we cannot have a situation where there is an open timeslot, room, and a valid class that was not scheduled after the algorithm terminates.

Since we see that all cases end in contradiction, the algorithm returns a valid schedule.

1.6.1 Design Choices

We chose this algorithm because we believed it to be an optimal algorithm in garnering student preference values by looking at the most popular classes and assigning them to the biggest rooms. The greedy part of our algorithm is that we want to assign a class to a specific timeslot and room such that we get as many students enrolled in the class as possible. By sorting R by room sizes, we assign the most popular classes in the biggest rooms at different timeslots.

We encountered many complications when designing this algorithm. For example, our algorithm is purely popularity based, where we simply assessed the popularity of a class without considering student conflicts. We did not account for the fact that there could be a lot of students who conflicted at a particular timeslot, which may explain the lower optimality value. A current complication we have and may still need to deal with is the fact that there may be a class that is not scheduled, but we still have a timeslot and room open. This only occurs when the class will conflict with another class in the same timeslot wherein the professor is already assigned to teaching another class. Could we schedule the classes in a different way to make room for that remaining class?

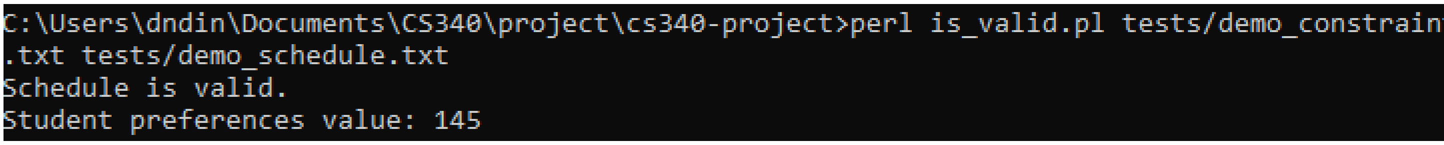
1.7 Implementation

Implementing our algorithm, we decided to do a purely popularity based greedy algorithm. Essentially, we tallied the number of students who preferred each class and sorted each class by descending popularity. We then put the most popular classes in the biggest rooms and an arbitrary timeslot such that a professor does not already teach at that timeslot. To also avoid student conflict, we removed a student from a class if they were already assigned at that particular timeslot. As a result, we return a finalized schedule that is valid and based on popularity.

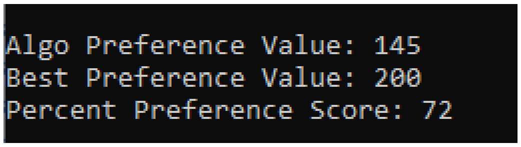
We created a class called classes to hold the information for our finalized classes. We then change the classes defined in schedule to reflect the assignments before eventually returning that finalized schedule.

1.7.1 Testing Results

We ran our algorithm on the demo constraints and student preferences and got this as a result for our created schedule:



```
C:\Users\dndin\Documents\CS340\project\cs340-project>perl is_valid.pl tests/demo_constrain
.txt tests/demo_schedule.txt
Schedule is valid.
Student preferences value: 145
```



```
Algo Preference Value: 145
Best Preference Value: 200
Percent Preference Score: 72
```

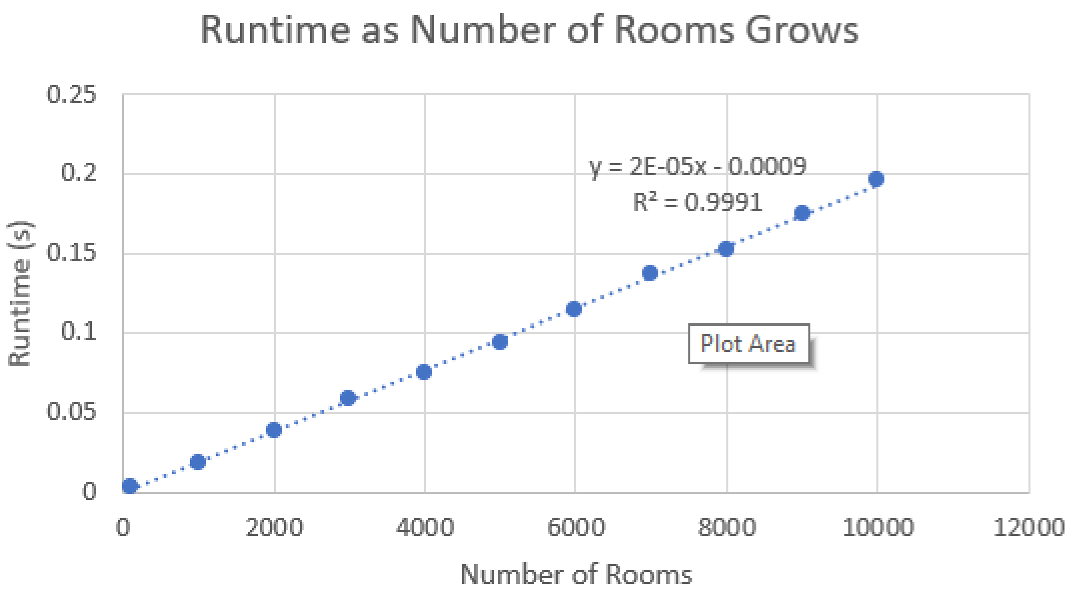
We then created 10 different constraints and student preferences that had 10 rooms, 36 classes (18 teachers), 5 timeslots, and 100 students. We then ran the algorithm on each of the random inputs and got an average percent preference score of 75 percent on randomly generated data. From our results, we saw that many students had a timeslot conflict where they could not take multiple courses they preferred because the courses were assigned in the same timeslot. Since there were only 5 timeslots, we had many of these student conflicts.

1.7.2 Experimental Runtime

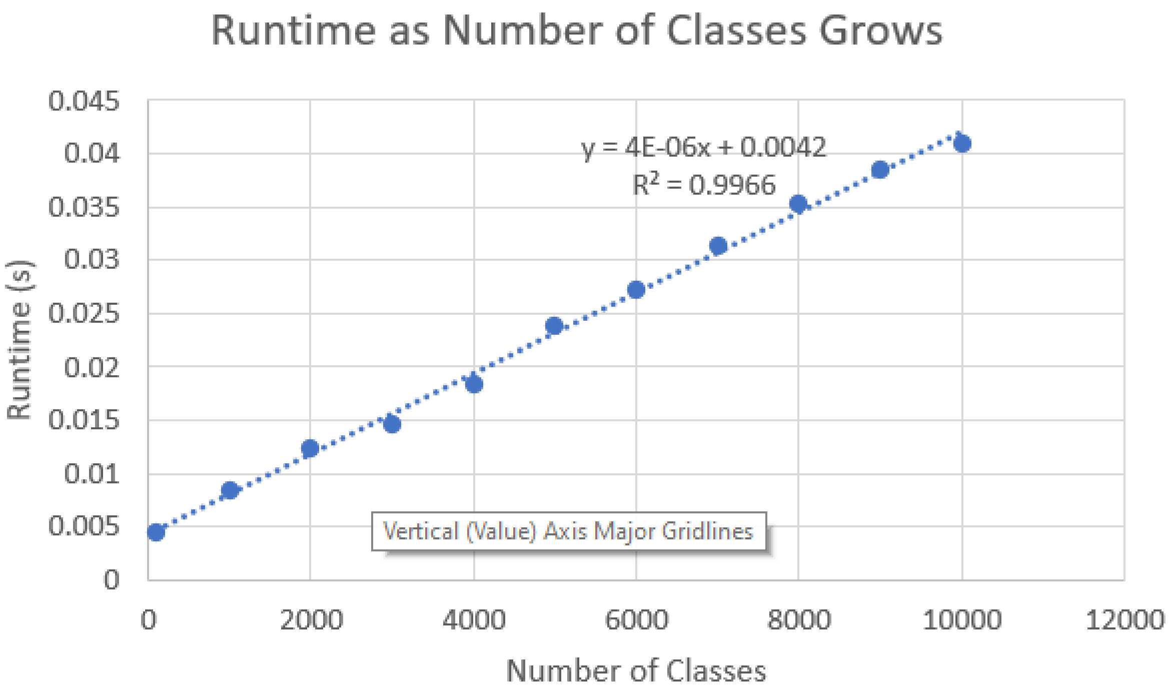
To observe our algorithm's runtime as the inputs grow, we chose an independent variable and increased its size while keeping all other variables constant. We chose to grow the inputs R, S, and C because they are able to actually grow in a real scenarios. We chose to increment each independent variable size by 1,000 until we reached 10,000 input size for that variable. We then recorded the run times of each and plotted the results on a graph to obtain a line of best fit for the relationship between input size of a particular variable and run time. We also held the other variables constant where each of their sizes were 100.

Runtime as the Number of Rooms Grow

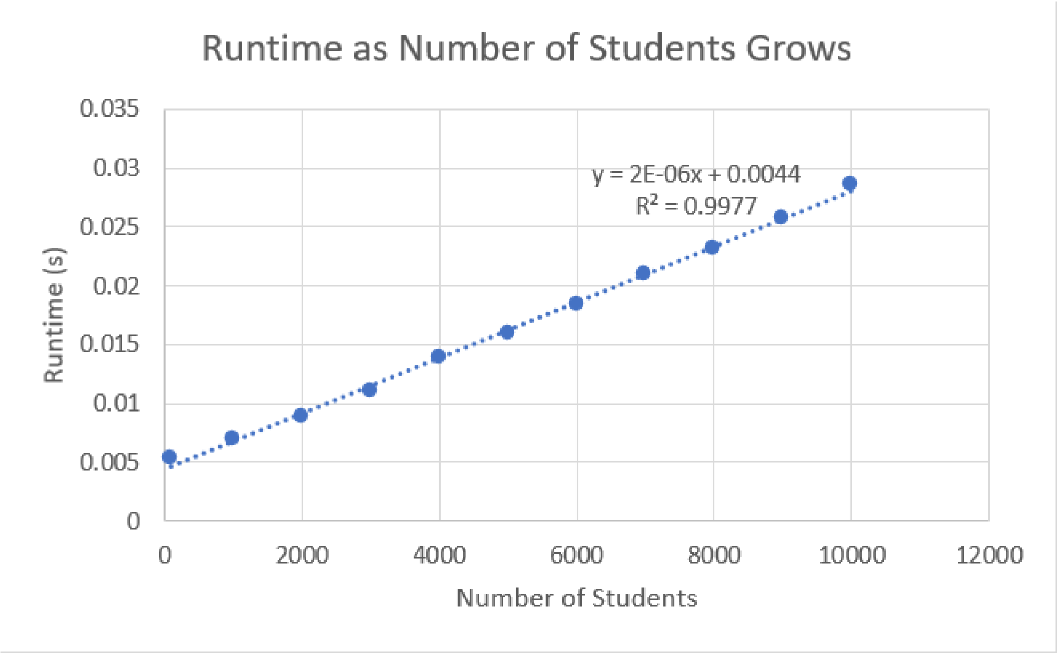
The graph below shows runtimes as the number of room grows. The constants are 100 classes, 100 students, 100 timeslots, and 50 professors. We first started with 100 rooms, then 1,000 rooms, then 2,000 rooms, ..., until we reached 10,000 rooms. We observed that each time we incremented the number of rooms by a thousand, the run time of our algorithm increased by about 0.02 seconds. Plotting each of our observations on a scatterplot and using excel's line of best fit function, we saw a linear relationship between the number of rooms and our algorithm's run time as our R^2 is closest to 1 under a linear line of best fit. Although we saw the time for sorting the rooms takes $O(rlogr)$, we see that our theoretical complexity $O(t * r * s^2)$ still dominates because a factor of logr for sorting is minimal compared to the size of the 100 students and 100 timeslots. We see that logr must exceed the product of the two constants t and s^2 to have the sorting of the rooms dominate.



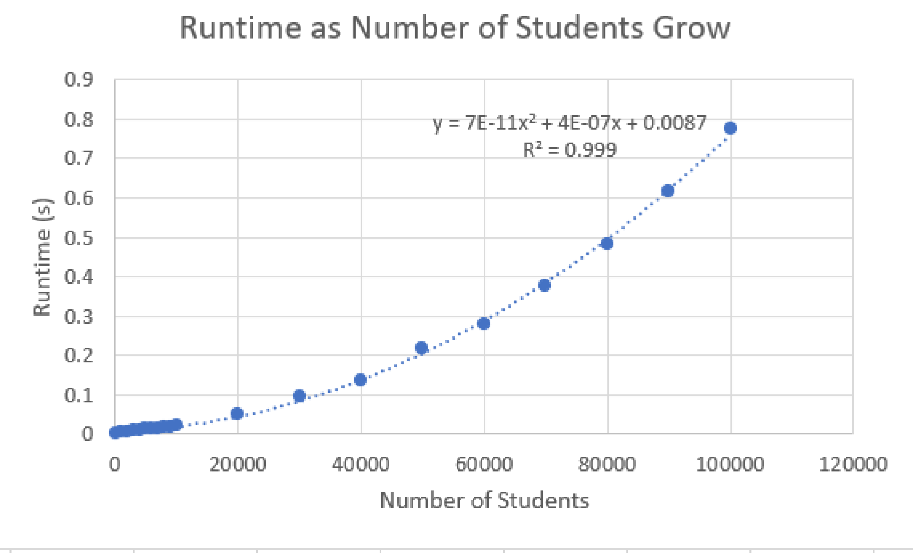
Runtime as the Number of Classes Grows The graph below shows runtimes as the number of classes grows. The constants are 100 rooms, 100 students, and 100 timeslots. The number of professors is always half of the number of classes. Similarly, we started out with only 100 classes, then 1,000 with increments of 1,000 classes until we reached 10,000 classes. We then saw that as we increased the number of classes by 1,000, the run time increased by about .004 seconds. Thus, we obtained a linear fit for the number of classes and the algorithm's runtime as the R^2 value is close to 1. As we noted in the time analysis, we see that the classToAssign function considers classes in the list of classes classPref, and we pop a class from classPref at the end of the function which takes $O(c)$. Therefore, we see that although, in the worst case, we would have a $O(s^2)$ complexity for classToAssign, we still see a linear growth $O(c)$ for the classToAssign function as c grows in size. Therefore, we see that growing c increases the runtime linearly as we also have to call classToAssign t*r times which dominates over sorting the classes. We are unable to show the case where sorting the classes will dominate because the number of classes must be less than or equal to the number of timeslots multiplied by the number of rooms in order for all classes to be assigned.



Runtime as Number of Students Grows (Average Case) The graph below shows runtimes as the number of students grows. The constants are 100 rooms, 100 classes, 50 professors, and 100 timeslots. This graph seemingly also reflects a linear growth as the number of students grows. We started out with only 100 students, then 1,000 students with increments of 1,000 students until we reached 10,000 students. We then plotted the runtimes and found a linear fit. Although our time analysis suggests that we should have quadratic growth as the number of students grows, we assumed the worst case for the function classToAssign having complexity $O(s^2)$. We would only have a quadratic function when we consider a class which every student prefers, and each of those students must have a conflict at that timeslot in order to pop every single student in that class. In that case, we would then achieve a quadratic fit, but in the average case we see that we have a linear relationship between run time and the amount of students as shown by the line of best fit and the R^2 which is close to 1.



Runtime as Number of Students Grows (Worst Case) The graph below shows the runtimes as the number of students grows but with most of the students having timeslot conflicts. We made the number of timeslots to be 1, number of rooms to be 8, and number of classes be 8 in order to get the worst case of every student having timeslot conflicts. Since there is only 1 timeslot, a student can only be enrolled in one course and has to be popped from their three other courses. We then started plotting runtimes with 1000 students to 10000 students with 1000 student increments. We saw a linear relationship at first, but as we plotted the runtimes for 10000 students to 100000 students with 10000 student increments, we determined that the best fit function is quadratic because the quadratic function best reduced the squared residuals and gave an R^2 value closest to 1. Once a student is assigned to a course in the single timeslot, we must pop the student from the rest of the courses in order to adhere to the constraint that a student can only take one course at a timeslot. Thus, in the worst case that we must pop many students, we see that the runtime is quadratic in terms of the number of students, which is what we determined in the time analysis as $O(t * r * s^2)$.



As a result, we see that from the experimental analysis that the algorithmic run time does grow as we expected in our time analysis, where the runtime increases linearly when the number of classes grow (classToAssign is linear in terms of c despite being dominated by s^2) and when the number of rooms grows. As well, we saw a quadratic relationship between the runtime and the number of students in the worst case when many students had to be popped from their preferred courses. Therefore, we do see that the complexity $O(t * r * s^2)$ holds in the worst case.

1.7.3 Solution Quality Analysis

Assuming that the best student preference value is the upper bound for the optimal value, we can stress test to find the lower bound of our experimental value when running our algorithm.

Stress/Cases Conflicts

First case: the number of classes equals the number of rooms multiplied by timeslots. We consider this a stress case because we will not have excess rooms and timeslots to accommodate conflicts for bigger classes (we might have to put bigger classes in smaller rooms). We expect the percent preference value to be lower. Inputs:

Number of rooms: 10

Number of classes: 30

Number of timeslots: 3

Number of students: 100

Number of professors: 15

We did 4 trials, and saw an average percent preference value of 61.75 percent.

Second case: the number of classes equals the number of rooms times timeslots. However, we see that the number of rooms and the number of timeslots are swapped. In one of the tests, we had class three that was not assigned to any room because the professor conflicted with the last timeslot available. We had an 89 percent preference value for that trial.

Number of rooms: 3

Number of classes: 30

Number of timeslots: 10

Number of students: 100

Number of professors: 15

We did 4 trials, and saw an average percent preference value of 90.25 percent.

Third case: we have a large number of students compared to the number of classes, rooms, and timeslots. We should have a lot of student conflicts because the size of the rooms only goes up to 1000, meaning we should have a low percent preference value because we must pop students due to conflicts.

Number of rooms: 3 (max room size is 1000)

Number of classes: 30

Number of timeslots: 10

Number of students: 7500

Number of professors: 15

Since the room sizes are variable, we saw that the percent preference values were erratic for the 4 trials with the lowest being 21 percent and the highest being 61 percent. The average percent preference value is 41 percent.

Fourth Case: we only have one timeslot, four classes, and four rooms where the maximum capacity is 1000. We had a total of 1000 students (maximum students allowed). Although the best or optimal preference value cannot be the best student preference value (4*s), we observed that the experiment preference value was about 25 percent with the lowest being 21.975 percent out of four trials.

Number of rooms: 4 (max room size is 1000)

Number of classes: 4

Number of timeslots: 1

Number of students: 1000

Number of professors: 2

We saw that the optimality value was hovering between 20 percent and 25 percent in this stress test. Since there is only one timeslot, we know that a student can never be enrolled in more than one course which gives us an upperbound of 25 percent optimality. We saw an average optimality of 23.8375 when running our algorithm on this stress case.

By forcing student conflicts and room capacity conflicts, we found that our algorithm had a lower bound of about $\frac{1}{5}$ of the upperbound of the best student preference value. Thus, our algorithm always achieves a preference value score of greater than $\frac{1}{5}$ of the best case student value.

2 Extensions

2.1 Methodology

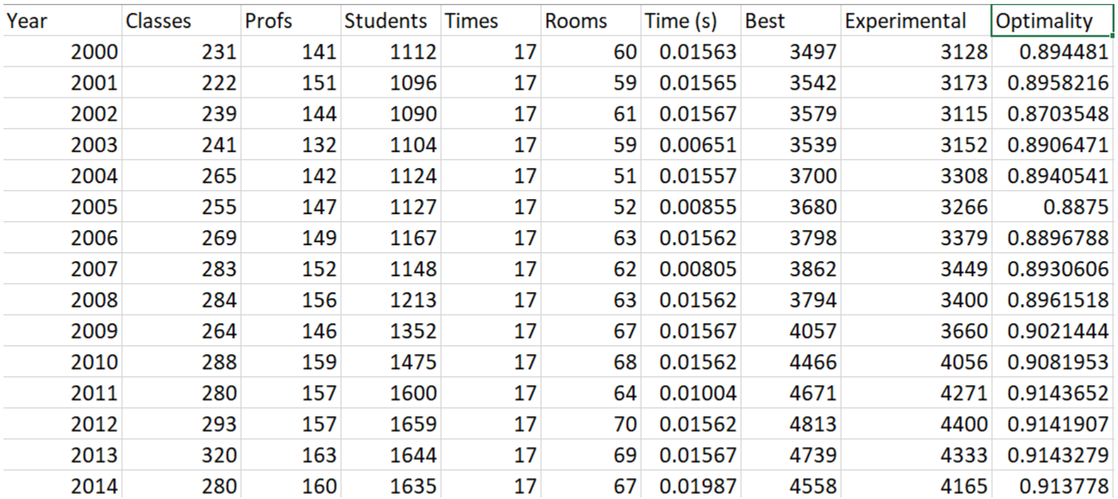
For extending our algorithm to real data, we parsed the enrollment information for the Bryn Mawr 2000-2014 Fall semesters to create constraints based on the classes and subjects taught, professors teaching, rooms occupied and their capacities, and create student preferences based on the courses the students were enrolled in. Note that in the enrollment data, we saw that there were classes where there were no teachers assigned, so we removed those classes and removed student preferences which had those classes with no professors. In total, the number of classes which had no professor were minimal such that there were only about 1 or 2 of these classes each semester.

As for the timeslots, we determined that a weekly schedule would realistically have about 17 timeslots without considering overlapping timeslots. We realized that each class must fulfill a 3 hour requirement, so we considered timeslots that were around 3 hours each. For each weekday (5 days), we considered the time intervals 8AM-5PM and 7PM-10PM. For the first time interval 8AM-5PM, we see that we have a total of 9 hours which we can place three 3 hour timeslots into (or 2 hour 50 minute classes with 10 walk time). We then see that we have a total of 15 timeslots across the five days. We then also considered night classes and placed only two more timeslots on Mondays and Wednesdays from 7PM-10PM. Thus, we came up with a total of 17 timeslots for a realistic weekly schedule, which we used for the number of timeslots for each Fall semester.

We then ran our original, unmodified algorithm on the data as a control, so we can compare optimality before and after implementing the extensions.

2.1.1 Analysis on Unmodified Data

After running our unmodified algorithm on the parsed constraints and student preferences, we observed that the optimality for each semester hovered around 90 percent. The table below shows our data collection for each fall semester as a trial. In each year, we observed that many of the conflicts came from students not being able to take more than one course in a particular timeslot, so we had to pop the students from the finalized classes. These conflicts attributed to much of the preference score that was lost. We also had a smaller conflict which was class preferences exceeding room capacities, but that was minimal because the student preferences was parsed from students that were already enrolled in a course which is not based on their actual preferences. Considering these conflicts, we implemented constraints and extensions that mimiced real life scenarios but also strived to reduce these conflicts.



Year	Classes	Profs	Students	Times	Rooms	Time (s)	Best	Experimental	Optimality
2000	231	141	1112	17	60	0.01563	3497	3128	0.894481
2001	222	151	1096	17	59	0.01565	3542	3173	0.8958216
2002	239	144	1090	17	61	0.01567	3579	3115	0.8703548
2003	241	132	1104	17	59	0.00651	3539	3152	0.8906471
2004	265	142	1124	17	51	0.01557	3700	3308	0.8940541
2005	255	147	1127	17	52	0.00855	3680	3266	0.8875
2006	269	149	1167	17	63	0.01562	3798	3379	0.8896788
2007	283	152	1148	17	62	0.00805	3862	3449	0.8930606
2008	284	156	1213	17	63	0.01562	3794	3400	0.8961518
2009	264	146	1352	17	67	0.01567	4057	3660	0.9021444
2010	288	159	1475	17	68	0.01562	4466	4056	0.9081953
2011	280	157	1600	17	64	0.01004	4671	4271	0.9143652
2012	293	157	1659	17	70	0.01562	4813	4400	0.9141907
2013	320	163	1644	17	69	0.01567	4739	4333	0.9143279
2014	280	160	1635	17	67	0.01987	4558	4165	0.913778

We computed a mean statistic of 89.86 percent optimality across 15 observations. We will then compare this mean optimality to our modified algorithms based on our extensions.

2.2 Extension1: *Evenly Spreading Same Subject Courses across Timeslots*

2.2.1 Description

Considering most of the loss in optimality comes from student conflict, we looked into the idea that generally students would want to take courses based on their interests or majors. Therefore, we would want to space out the subjects across all timeslots to reduce the amount of student conflicts based on subject interest. We considered the possibility that, at a liberal arts college such as Bryn Mawr, students would want to explore subjects outside of their interests, but we believed that students would still have to take a majority of their classes in a particular subject to fulfill their major requirements.

2.2.2 Implementation

To evenly assign same subject courses across timeslots, we set a dynamic course capacity for each subject across all timeslots, kept track of the total number of courses for each subject, and we kept track of the number of courses for each subject at a timeslot. Before we ever assign classes to timeslots, we set the course capacity of each subject to 1 to indicate that each timeslot can have at most 1 course with that particular subject. Once we start assigning courses to timeslots, we update the total number of courses for that course subject by 1 and update the number of courses of that assigned subject at the timeslot by 1 only if we have not already reached the course subject capacity. If we have reached the subject capacity at a timeslot, then we ignore courses with that subject at that timeslot and continue to the next popular courses. And, once all timeslots have reached the course subject capacity, we increase that course subject capacity by 1. We then continue this process until we have considered all timeslots and rooms.

2.2.3 Time Analysis

Alongside the new data structure of tSubjects we created, we see that we must analyze the time it takes to 1) build a dictionary of subjects from the parsed inputs, 2) update 3D array tSubjects in classToAssign, and 3) check if we can assign a class to a timeslot based on the subject capacity condition.

Let b denote the number of distinct subjects for all courses.

For 1), we first build the dictionary by going through every class and adding unique subjects to the dictionary as the keys and assigning value 0 to each key to indicate that we have not finalized any courses (thus no subjects have been tallied). Thus, we see that it takes $O(c)$ to build the first dictionary, which we can then copy in $O(b)$ to build tSubjects in $O(c + b * t)$. We see that the building of dictionaries and tSubjects is only done once in function scheduler, which does not affect our overall time of our original algorithm: $O(t * r * s^2)$ because the number of subjects is less than the number of classes.

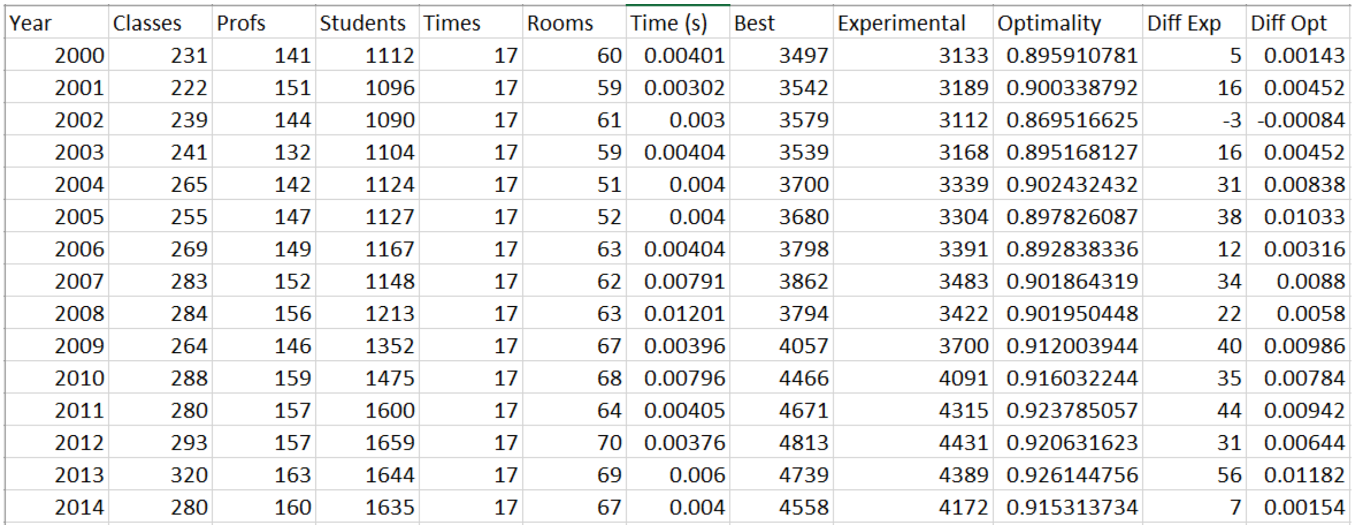
For 2), once we assign a course with arbitrary subject j to a timeslot in classToAssign, we update tSubjects' course subject j capacity in the first index, update total number of courses for subject j across all timeslots in the second index, and update the number of courses for subject j at the timeslot we are considering. Well, updating the total number of courses for subject j across all timeslots and the number of courses for subject j at the considered timeslot by 1 takes $O(1)$ because tSubjects is an array of dictionaries. Then, we must update the course capacity for subject j, such that if all timeslots have reached the subject j capacity, we increase the capacity by 1. By keeping a tally of the number of total courses with subject j, we can update the capacity by dividing (integer division) the total number of courses with subject j by the number of timeslots and then add 1 to the quotient (the 1 corresponds to the starting subject capacity of 1). Thus, since we are simply accessing values in tSubjects and doing arithmetic, we are able to do 2) in $O(1)$.

For 3), when we consider a class in classPref with subject j at a particular timeslot, we must also check for the condition that we have not reached the course subject capacity for subject j at the timeslot. Since we simply access the number of courses with subject j in the third index of tSubjects and compare it to the subject j capacity in the first index of tSubjects, we can check for the condition in $O(1)$. However, the number of classes we ignore is variable because we can ignore at most $O(c)$ courses before concluding that we are not able to assign a viable course to that particular timeslot. Thus, we see that we add a factor of $O(c)$ to the function classToAssign, but we still see that $O(s^2)$ dominates which was deduced in the time analysis of the original algorithm.

Therefore, since we know that the number of subjects is less than the number of classes, then we see that our algorithm modified with the subject capacity constraint maintains a time complexity of $O(t * r * s^2)$ in the worst case.

2.2.4 Experimental Result

Looking at the table below, we saw a general increase in optimality across all fall semesters barring the year 2002. We saw an increase in optimality of around 1 percent or less among all semesters. Although the increase may not seem significant, we had, on average, a loss of 10 percent optimality in the original algorithm. Reducing the loss of optimality to about 9 percent is a 10 percent improvement. We still observed many student not being able to take the preferred courses because of timeslot conflicts, which we can attribute to students wanting to explore different subjects and take courses outside of their major. Since we ran the algorithm on Bryn Mawr data – a liberal arts college – the results we collected most likely do not reflect the improvements we might see if we ran the algorithm on a non-liberal arts college where students do not have to take many courses outside of their major.



Year	Classes	Profs	Students	Times	Rooms	Time (s)	Best	Experimental	Optimality	Diff Exp	Diff Opt
2000	231	141	1112	17	60	0.00401	3497	3133	0.895910781	5	0.00143
2001	222	151	1096	17	59	0.00302	3542	3189	0.900338792	16	0.00452
2002	239	144	1090	17	61	0.003	3579	3112	0.869516625	-3	-0.00084
2003	241	132	1104	17	59	0.00404	3539	3168	0.895168127	16	0.00452
2004	265	142	1124	17	51	0.004	3700	3339	0.902432432	31	0.00838
2005	255	147	1127	17	52	0.004	3680	3304	0.897826087	38	0.01033
2006	269	149	1167	17	63	0.00404	3798	3391	0.892838336	12	0.00316
2007	283	152	1148	17	62	0.00791	3862	3483	0.901864319	34	0.0088
2008	284	156	1213	17	63	0.01201	3794	3422	0.901950448	22	0.0058
2009	264	146	1352	17	67	0.00396	4057	3700	0.912003944	40	0.00986
2010	288	159	1475	17	68	0.00796	4466	4091	0.916032244	35	0.00784
2011	280	157	1600	17	64	0.00405	4671	4315	0.923785057	44	0.00942
2012	293	157	1659	17	70	0.00376	4813	4431	0.920631623	31	0.00644
2013	320	163	1644	17	69	0.006	4739	4389	0.926144756	56	0.01182
2014	280	160	1635	17	67	0.004	4558	4172	0.915313734	7	0.00154

We computed a mean statistic of 90.88 percent optimality as a result of evenly spreading out courses with the same subject across all timeslots. We see that the mean optimality of this constraint is about 1 percent higher than the optimality of the unmodified algorithm. Thus, we do recommend that the registrar look into spreading out courses with similar level subjects across nonoverlapping timeslots.

2.2.5 Additional Data Structures

Additional data structures for this constraint includes tSubjects. tSubjects is a 3D array that contains in its first index a dictionary called subject-capacities corresponding to the capacity of each subject. The second index contains a dictionary tally-subjects that corresponds to the total number of courses associated with a particular subject. The final index is an array representation the number of courses for a certain subject at a timeslot. Let b denote the number of subjects, where $b < c$ (or a subject constraint would become trivial). Each dictionary will have b keys and take $O(b)$ to construct. We see that tSubjects will have 2 + t dictionaries because the first two indices both have one dictionary and the last index has t dictionaries (one for each timeslot). Therefore, we see that it takes $O(b * (2 + t)) = O(b * t)$ time to construct tSubjects. We can then access and assign values in constant time because tSubjects is an array of dictionaries.

2.3 Extension2: *Evenly Spreading Same Level Courses across Timeslots*

2.3.1 Description

To reduce the number of student conflicts, we also looked at how students might want to register for their courses based on the levels of the courses. So, we considered that students most likely register for more of the same level courses each semester than having variability in the level of their courses. We attributed this to the notions that students need to take courses based on their year in college, where underclassmen tend to take introductory or level 100 courses in order to have the prerequisites for higher level courses and upperclassmen tend to take advanced or level 300 courses in order to fulfill their major requirements and graduate. We also considered the possibility that students might want to take easier level courses to balance out the difficulty of their schedule, but we believed that more students would follow such notions rather than go against them.

2.3.2 Implementation

Considering these notions, we modified our algorithm to spread out courses with similar levels evenly across timeslots. By having a dynamic level capacity across all timeslots, we will only add a class to a timeslot if the number of courses with that level has not reached the level capacity. Once we assign a course, we will update the total number of courses with that level across all timeslots and the number of courses with that level in that specific timeslot by 1. If we have reached the capacity for that level at a timeslot, then we will not assign courses with that level at that particular timeslot and continue onto the next popular courses. Once all timeslots have reached the level capacity, we will increase that particular level capacity by 1. We chose to have a dynamic level capacity to limit the amount of popular courses in the same timeslot, which would likely occur if we had a static level capacity.

2.3.3 Time Analysis

Alongside our data structure of tLevels which takes $O(t)$ to construct, we also have to consider the time it takes to 1) update our tLevels data structure in each call of classToAssign and 2) check if we can assign a considered class to a timeslot based on the level capacity.

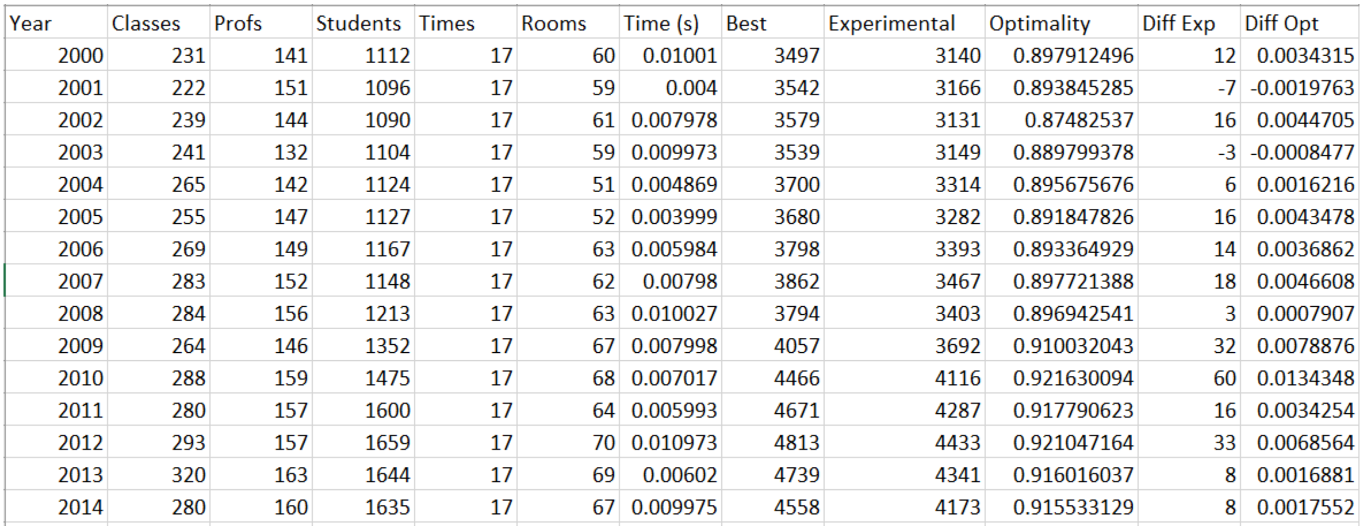
For 1), once we assign a class with arbitrary level l to a timeslot, we must update the total number of courses that have level l by 1 (second index of tLevels), update the number of level l courses at that timeslot by 1 (third index of tLevels), and if all timeslots have reached the capacity, we update the capacity of level l by 1 (first index of tLevels). Well, since tLevels is an array, we can update values in $O(1)$. We must now consider the time it takes to check if we should update the level l capacity or not. Usually, we would have to check all timeslots in the third index of tLevels to see if the number of level l classes is equal to the capacity of level l courses, but since we tallied the number of total level l courses in the second index of tLevels, we can simply update the capacity of level l to be one plus the total number of level l courses divided (integer division) by the number of timeslots (the one indicates the starting capacity we must carry through). Accessing and assigning array values takes $O(1)$ and doing arithmetic takes $O(1)$. Therefore, we see that 1) takes constant time.

For 2), when considering a class with arbitrary level l, we must check if the number of level l courses at the timeslot is less than the level l capacity. We see that accessing each takes $O(1)$ because we simply access the first index of tLevels to check for the level l capacity and we can access the third index of tLevels to access the number of level l courses in the timeslot. However, we see that if we have already reached the level l capacity before assigning the course, then we continue considering the next popular courses until we reach a class that has a level that has not reached the capacity at that timeslot. So, we see that the number of courses we consider in each iteration of classToAssign is now variable because we do not know which of the most popular courses are valid. In the worst case, we might see that no courses are valid and we might have to consider $O(c)$ classes before we say that there are no valid classes to assign to the timeslot. Thus, we see that the constraint adds $O(c)$ time to the function classToAssign, but since we see that classToAssign already had a $O(s^2)$ complexity before the extension, we see that the complexity of the function does not change.

Although it takes an extra $O(t)$ to construct tLevels and it takes $O(c)$ to check if a class has a valid level, we see that the original complexity of the algorithm $O(t * r * s^2)$ dominates.

2.3.4 Experimental Result

Looking at the table below, we saw a slight increase in optimality across all semesters barring 2001 and 2003. We observed that the extra constraint generally increased our optimality as we expected, but we still observed many students being popped from classes because of timeslot conflict. This timeslot conflict attributes to the decrease in optimality in the years 2001 and 2003, which could be a result of students wanting to take varying level of courses to balance out the difficulty of their schedule. Although a 1 percent increase might not be much of an incentive to add this extra constraint to actual scheduling, we note that there is a high probability (13/15 chance when looking at the samples) that spreading out similar level courses across timeslots will increase the optimality and reduce student conflict. Therefore, we do suggest the registrar consider spreading out courses with the same level.



Year	Classes	Profs	Students	Times	Rooms	Time (s)	Best	Experimental	Optimality	Diff Exp	Diff Opt
2000	231	141	1112	17	60	0.01001	3497	3140	0.897912496	12	0.0034315
2001	222	151	1096	17	59	0.004	3542	3166	0.893845285	-7	-0.0019763
2002	239	144	1090	17	61	0.007978	3579	3131	0.87482537	16	0.0044705
2003	241	132	1104	17	59	0.009973	3539	3149	0.889799378	-3	-0.0008477
2004	265	142	1124	17	51	0.004869	3700	3314	0.895675676	6	0.0016216
2005	255	147	1127	17	52	0.003999	3680	3282	0.891847826	16	0.0043478
2006	269	149	1167	17	63	0.005984	3798	3393	0.893364929	14	0.0036862
2007	283	152	1148	17	62	0.00798	3862	3467	0.897721388	18	0.0046608
2008	284	156	1213	17	63	0.010027	3794	3403	0.896942541	3	0.0007907
2009	264	146	1352	17	67	0.007998	4057	3692	0.910032043	32	0.0078876
2010	288	159	1475	17	68	0.007017	4466	4116	0.921630094	60	0.0134348
2011	280	157	1600	17	64	0.005993	4671	4287	0.917790623	16	0.0034254
2012	293	157	1659	17	70	0.010973	4813	4433	0.921047164	33	0.0068564
2013	320	163	1644	17	69	0.00602	4739	4341	0.916016037	8	0.0016881
2014	280	160	1635	17	67	0.009975	4558	4173	0.915533129	8	0.0017552

We computed a mean statistic of 90.56 percent optimality as a result of spreading out courses with the same level across all timeslots. The mean optimality of 90.56 percent with the level constraint is about 0.70 higher than the mean optimality of our unconstrained algorithm.

2.3.5 Additional Data Structures

Additional data stuctures for this constraint includes tLevels. tLevels is a 3D array that holds the level capacities, the total amount of courses for a particular level across all timeslots, and the amount of courses for a particular level at a timeslot. At Bryn Mawr, we know that the levels of the courses range from 100 level courses to 300 level courses, thus the first index and second indices of tLevels hold 3 element arrays, while the third index of tLevels will have t elements each with 3 element arrays. Thus, the construction of tLevels takes O(3 + 3 + t*3), which can be simplified to *O(t)* because the number of timeslots dominates over the number of levels. We can reassign and access amounts of courses in a level in $O(1)$.

2.4 Merging Extensions 1 and 2

2.4.1 Description

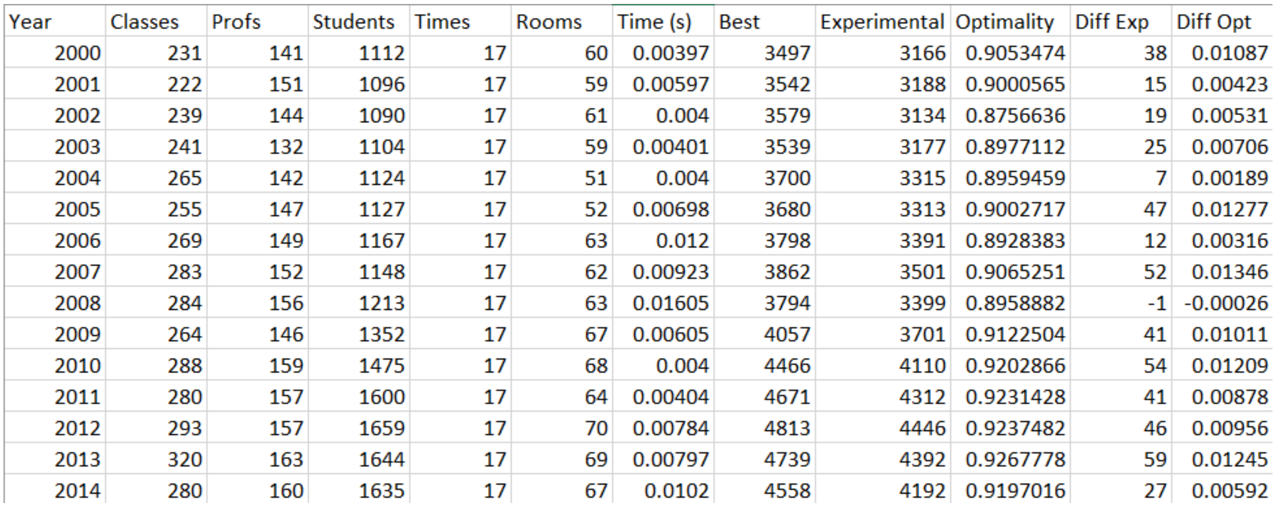
Seeing the general increases of both the subject and level constraint, we paired the two constraints together to modify the algorithm to evenly space out courses with the same subject and level.

2.4.2 Time Analysis

We saw that in both the time analyses of extension 1 and 2 that the complexity of the algorithm remains the same as the unmodified algorithm, therefore the combination of the two does not change the worst case complexity of the algorithm. So, the time complexity of the modified algorithm remains $O(t * r * s^2)$.

2.4.3 Experimental Results

When pairing the two constraints together, we saw an increase that was greater than each isolated constraint except in the year 2008. While we did see a greater increase, we also noted that there now existed classes that were not assigned to timeslots and rooms because constraints 1 and 2 are not disjoint. When considering the last room, we might see that although a timeslot has not reached a subject-capacity, we might have reached a level capacity, and thus we cannot assign a class that fulfills the subject condition but fails the level condition. The reverse can also happen where a class fulfills the level condition and fails the subject condition at a timeslot. This was apparent in the year 2008 where 9 courses were not assigned to a timeslot or room as a result of the two conditions not being disjoint.



Year	Classes	Profs	Students	Times	Rooms	Time (s)	Best	Experimental	Optimality	Diff Exp	Diff Opt
2000	231	141	1112	17	60	0.00397	3497	3166	0.9053474	38	0.01087
2001	222	151	1096	17	59	0.00597	3542	3188	0.9000565	15	0.00423
2002	239	144	1090	17	61	0.004	3579	3134	0.8756636	19	0.00531
2003	241	132	1104	17	59	0.00401	3539	3177	0.8977112	25	0.00706
2004	265	142	1124	17	51	0.004	3700	3315	0.8959459	7	0.00189
2005	255	147	1127	17	52	0.00698	3680	3313	0.9002717	47	0.01277
2006	269	149	1167	17	63	0.012	3798	3391	0.8928383	12	0.00316
2007	283	152	1148	17	62	0.00923	3862	3501	0.9065251	52	0.01346
2008	284	156	1213	17	63	0.01605	3794	3399	0.8958882	-1	-0.00026
2009	264	146	1352	17	67	0.00605	4057	3701	0.9122504	41	0.01011
2010	288	159	1475	17	68	0.004	4466	4110	0.9202866	54	0.01209
2011	280	157	1600	17	64	0.00404	4671	4312	0.9231428	41	0.00878
2012	293	157	1659	17	70	0.00784	4813	4446	0.9237482	46	0.00956
2013	320	163	1644	17	69	0.00797	4739	4392	0.9267778	59	0.01245
2014	280	160	1635	17	67	0.0102	4558	4192	0.9197016	27	0.00592

We computed a mean statistic of 90.96 percent optimality when we merged the two constraints into one algorithm. This mean statistic is the greatest optimality we obtained from all of our extensions and although we had conflicts for a few of the classes because of the conflicting level and subject condition, we recommend that the registrar evenly space out both same subject and same level courses to reduce the amount of student timeslot conflicts.

2.4.4 Data Structures

As stated in the data structures section of extension 1 and 2, we see that we have added 3D arrays tLevels and tSubjects.

2.5 Extension3: *Subjects in Specific Rooms/Buildings*

2.5.1 Description

When looking at the different classes we noticed that specific classes would require equipment tailored just for those classes. For example chemistry classes would at times require lab specific equipment or Computer Science courses would need computers for students to work on. As a result we decided that the best way to combat such a issue would be to assign certain course to buildings that will specifically meet their needs. So STEM classes would only be in Park since that is where equipment they depend on is held.

2.5.2 Implementation

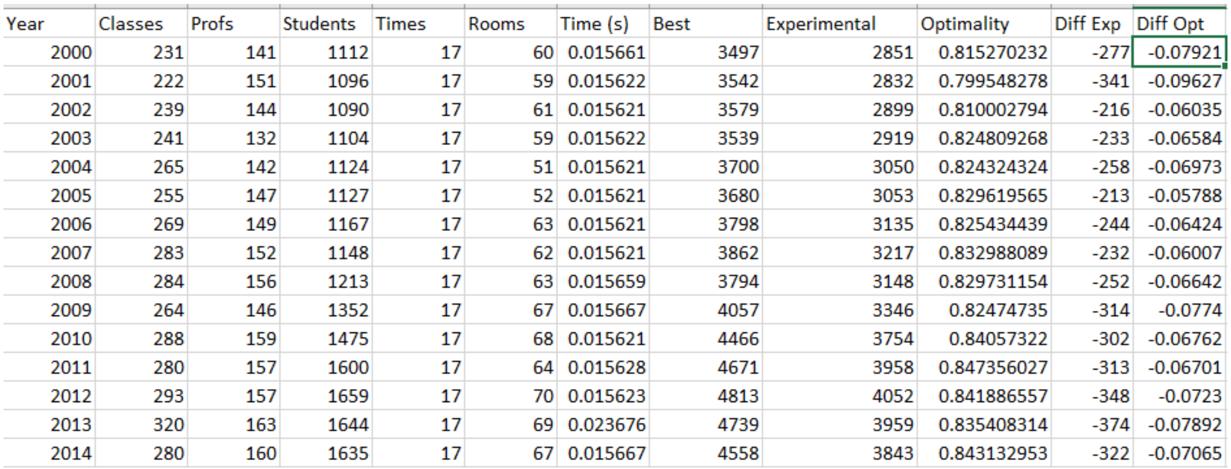
Knowing our goal, we modified our algorithm specifically, the main file when reading in the data, to parse out classes and their subjects with the associated teacher and a list of viable rooms that they can possibly be assigned to. From here, when assigning classes it was quite similar to the basic implementation with the added caveat of also checking to be sure that the room we will be assigning a class to is in the list of viable rooms associated with that class. If the class can be assigned to the considered room then we assign the class to the room, but if the room is not compatible, then we consider the next popular class until we find (or not) find a class that can fit in the considered room.

2.5.3 Time Analysis

In terms of modifications as mentioned before, classes will now each have a set of viable rooms which can be at most size r. With this list of viable rooms, we can check if the room we are considering is viable for a class in $O(r)$. Then, if a room is not viable for a class, then we continue onto the next popular classes until we find a class that can be assigned to that room. However, it is possible that the remainder of the classes cannot be assigned to the room, so we might have to do *O(cr)* work in the worst case where there is no class that can be assigned to the room. This gives us a total complexity of *O(*$c * r + s^2$*)* for each call of classToAssign when going through all courses and then assigning the final course considered to the timeslot and room while also having to pop every student from the class due to timeslot conflict. Therefore, we see that the algorithm will have a time complexity of O($t * r(c * r + s^2)$) as we call classToAssign number of timeslots multiplied by number of room times.

2.5.4 Experimental Result

Looking at the table below, we saw a high decrease in optimality across all years in comparison to the data from the unmodified algorithm. The mean optimality also decreased by over 6 percent. When studying the results we attributed this decrease to be due to the fact that since there are less options to assign classes, students are more prone to have conflicts based on timeslots, which negatively affects our optimality score.



Year	Classes	Profs	Students	Times	Rooms	Time (s)	Best	Experimental	Optimality	Diff Exp	Diff Opt
2000	231	141	1112	17	60	0.015661	3497	2851	0.815270232	-277	-0.07921
2001	222	151	1096	17	59	0.015622	3542	2832	0.799548278	-341	-0.09627
2002	239	144	1090	17	61	0.015621	3579	2899	0.810002794	-216	-0.06035
2003	241	132	1104	17	59	0.015622	3539	2919	0.824809268	-233	-0.06584
2004	265	142	1124	17	51	0.015621	3700	3050	0.824324324	-258	-0.06973
2005	255	147	1127	17	52	0.015621	3680	3053	0.829619565	-213	-0.05788
2006	269	149	1167	17	63	0.015621	3798	3135	0.825434439	-244	-0.06424
2007	283	152	1148	17	62	0.015621	3862	3217	0.832988089	-232	-0.06007
2008	284	156	1213	17	63	0.015659	3794	3148	0.829731154	-252	-0.06642
2009	264	146	1352	17	67	0.015667	4057	3346	0.82474735	-314	-0.0774
2010	288	159	1475	17	68	0.015621	4466	3754	0.84057322	-302	-0.06762
2011	280	157	1600	17	64	0.015628	4671	3958	0.847356027	-313	-0.06701
2012	293	157	1659	17	70	0.015623	4813	4052	0.841886557	-348	-0.0723
2013	320	163	1644	17	69	0.023676	4739	3959	0.835408314	-374	-0.07892
2014	280	160	1635	17	67	0.015667	4558	3843	0.843132953	-322	-0.07065

We computed the mean statistic of 83.33 percent for the optimality as a result of restricting classes to certain rooms and buildings based on the subject of the course. Since we saw a mean difference of about 6.5 percent between the mean optimality of the unconstrained algorithm and the mean optimality of the modified algorithm with subjects in specific room constraint, we suggest that the registrar not constrain course lectures to specific rooms and allow lectures (simple classes) be in any building and room.

2.5.5 Additional Data Structures

We added a new member variable to the classes class called viableRooms which is a list of rooms that a class can be assigned to. Since we have c classes and each of the viableRooms can have at most all of the r rooms, it would take $O(cr)$ to assign all viableRooms for all classes. We can then check to see if a room considered is in the viableRooms variable of a class in $O(r)$.

2.6 Extension4: *Increasing Room Size* by Increments of at Most Five

2.6.1 Description

At times, we saw that our original algorithm had to cut students who preferred a class if the room had a capacity lower than the number of students who could take the course (after removing students based on timeslot conflicts). To assuage the number of students cut from a course based on room capacity constraints, we decided to increase the room capacities by a controlled amount of five or less. We chose to only increase the room size by a small amount because we still considered physical classroom size as a real constraint and we believed that adding at most 5 chairs or desks would still be realistic and a viable recommendation.

2.6.2 Implementation

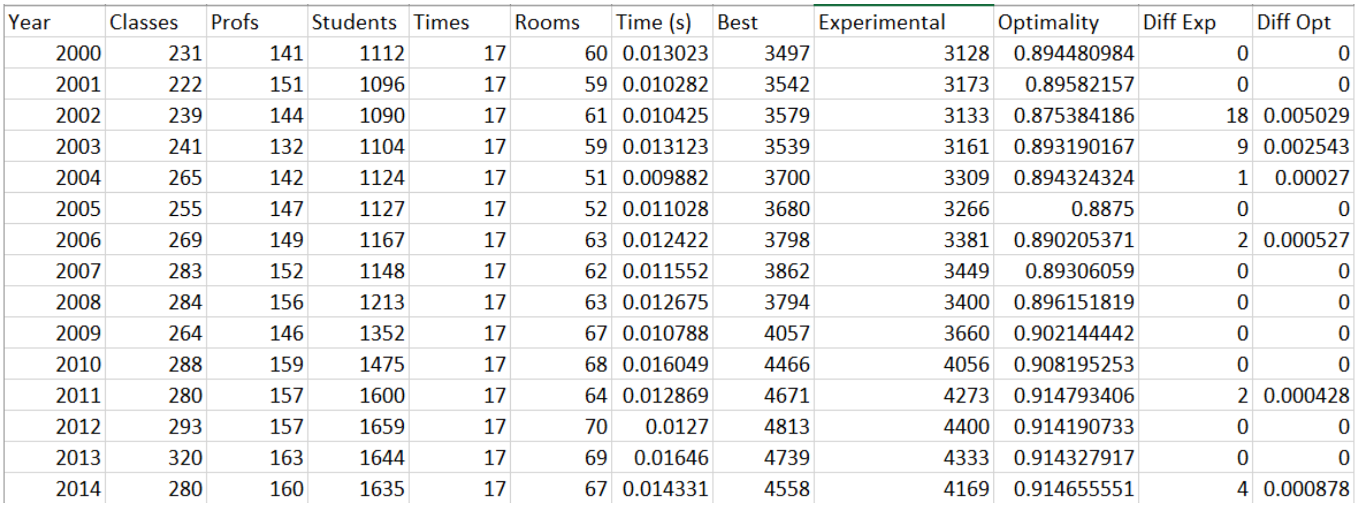
We simply changed the constraint input to increase room capacity by rounding room capacities up to the nearest multiple of 10 if rooms only need 5 or less seats. Then, we ran our unmodified algorithm on that constraint input.

2.6.3 Time Analysis

We did not modify our original algorithm to implement this extension, rather we edited the input constraints file to increase the room capacities. Then, we simply ran our original algorithm on that updated constraint input. Therefore, the algorithm with this extension still has a time complexity of $O(t * s * r^2)$ in the worst case.

2.6.4 Experimental Result

Observing the table below, we did not see a decrease in our experimental preference score or optimality compared to our unmodified algorithm. As we had expected from our unmodified run of our algorithm, class preference size being higher than room capacity was not much of an issue because we parsed student enrollment info and not actual preferences. We saw that our optimality either stayed the same or increased by a small amount because we only increased each room size by at most 5. Thus, we did not expect much of an increase in our optimality; however, if we were actually considering true preferences, then we would definitely see a greater increase as popular classes would become even more popular (as some students could not enroll in these courses).



Year	Classes	Profs	Students	Times	Rooms	Time (s)	Best	Experimental	Optimality	Diff Exp	Diff Opt
2000	231	141	1112	17	60	0.013023	3497	3128	0.894480984	0	0
2001	222	151	1096	17	59	0.010282	3542	3173	0.89582157	0	0
2002	239	144	1090	17	61	0.010425	3579	3133	0.875384186	18	0.005029
2003	241	132	1104	17	59	0.013123	3539	3161	0.893190167	9	0.002543
2004	265	142	1124	17	51	0.009882	3700	3309	0.894324324	1	0.00027
2005	255	147	1127	17	52	0.011028	3680	3266	0.8875	0	0
2006	269	149	1167	17	63	0.012422	3798	3381	0.890205371	2	0.000527
2007	283	152	1148	17	62	0.011552	3862	3449	0.89306059	0	0
2008	284	156	1213	17	63	0.012675	3794	3400	0.896151819	0	0
2009	264	146	1352	17	67	0.010788	4057	3660	0.902144442	0	0
2010	288	159	1475	17	68	0.016049	4466	4056	0.908195253	0	0
2011	280	157	1600	17	64	0.012869	4671	4273	0.914793406	2	0.000428
2012	293	157	1659	17	70	0.0127	4813	4400	0.914190733	0	0
2013	320	163	1644	17	69	0.01646	4739	4333	0.914327917	0	0
2014	280	160	1635	17	67	0.014331	4558	4169	0.914655551	4	0.000878

We computed a mean statistic of 90.19 percent optimality as a result of increasing room capacity. Since we saw no decrease in optimality and sometimes increases in optimality, we recommend that the registrar look into increasing the room sizes to allow for more students to take popular courses.

2.7 Extension5: *First Come First Serve*

2.7.1 Description

Instead of using student preferences as a means of measuring the popularity of a class, we use the student preferences input file as a means of simulating students registering for a course (which is true since the preferences are actually enrollment information). Since we no longer have preferences or popularity for classes, we have to assign classes based on another metric. We decided to then assign classes based on levels where the lower level courses are assigned to the bigger rooms and the higher level courses are assigned to the smallest rooms because every student is able to sign up for introductory courses while only a smaller portion of the student body can sign up for the advanced courses. Assigning classes to timeslots and rooms based on this heuristic allows for students to take alternative courses such as introductory classes when they are unable to take their preferred courses. The reasoning behind this new algorithm is to allow students to figure out their own schedules while the registrar has a hands off approach (without even needing a lottery).

2.7.2 Implementation

To implement our new algorithm, we used the same parsed inputs of constraints and student preferences as the arguments. We also still had a classes class to hold information about each class such as its room, professor, timeslot, list of students registered, and level. Afterwards, we sorted an array of classes by increasing levels. We also sorted the rooms by largest to smallest room capacity. We then assigned the lowest level courses to the biggest rooms if there are no professor conflicts because introductory classes are usually more popular than advanced classes and all students can sign up for these introductory courses as there are usually no prerequisites. Then, taking our list of student preferences, we shuffled the list to simulate randomness for the order in which students register for their courses. After shuffling, we iterate through the list of students and their preferences and mark students as registered or enrolled for their preferred course if they do not have another course in the same timeslot or the course is full. After every student has registered, the algorithm ends and the experimental preference value is calculated by the amount of preferred courses all students were able to register for. In a real scenario, there would be no preference value and the student would have to choose another course they might not prefer in order to satisfy the credit minimum each semester.

2.7.3 Time Analysis

Alongside the input of our unmodified algorithm, we added two new data structures to this algorithm a rand-order list of student preferences and a working schedule. Well, we see that rand-order is simply an array of students which takes $O(s)$ to construct. Both finalized schedule and working schedule take $O(c)$ to construct, but we also sort working schedule which takes $O(clogc)$. Afterwards, sorting the rooms takes $O(rlogr)$.

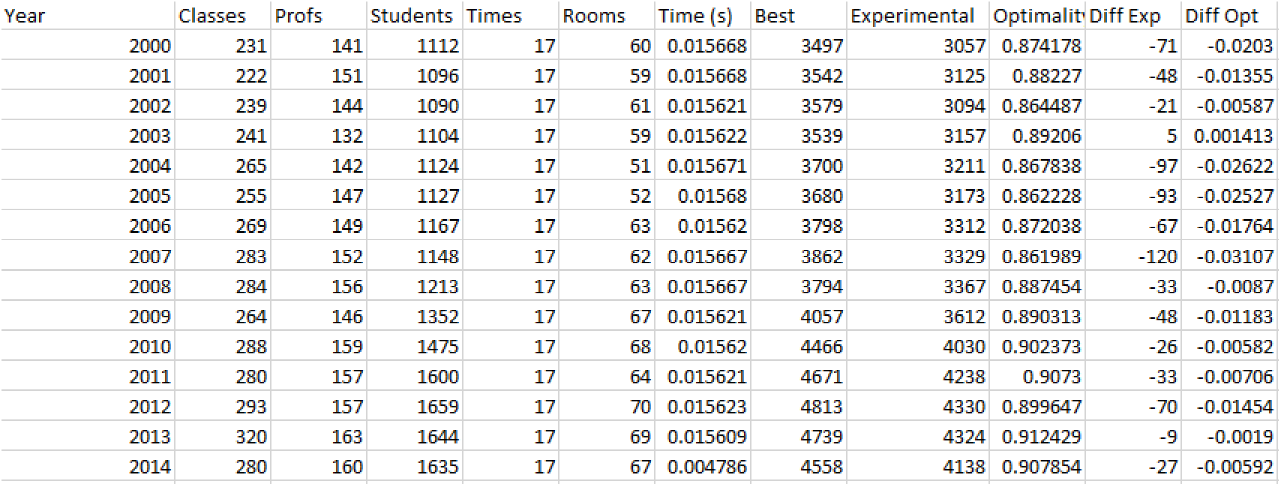
After setting up all the data structures, we then iterate through the biggest rooms and timeslots and assign the lowest level courses to the biggest rooms and the highest level courses to the smallest rooms and check for professor conflicts. If a considered class has a professor conflict, then we move onto the next class until we have found a class that has a viable professor. Once we assign a class to a timeslot, we update pTimeslots to reflect that a professor is now occupied at that particular timeslot. We see that we can iterate through at most $O(p)$ classes before finding a class that has a valid professor at that timeslot or not finding a valid class. Therefore, assigning classes to timeslots and rooms takes $O(t * r * p)$.

We then shuffle rand-order which takes $O(s)$, and iterate through rand-order to register students for the classes they prefer. We take the student from rand-order and access their preferences in S in constant time. We do not register a student in their preferred classes if the class was not assigned to a timeslot, if the student has a timeslot conflict, or if the class enrollment has reached the room capacity. We see that checking if a class was asssigned to a room or timeslot can be done in $O(1)$ by checking the member variable in the classes object. We know checking if there exists a student conflict takes $O(1)$ because we keep an array of lists called sTimeslots. And we know that checking if the class has reached its room capacity can be done in $O(1)$ by getting the length of the enrolled students list and checking it against the room capacity also stored in the classes object. So, we see that registering students to the courses takes time proportional to the number of students multiplied by each of their preferences. However, we know that the number of preferences is a small constant because a student taking a large number of courses is not feasible in a semester. Therefore, we see that registering students to classes takes $O(s)$ and the algorithm terminates afterwards.

So, we see that most likely the time it takes to assign classes to timeslots and rooms will dominate because the number of rooms multiplied by the number of professors will be larger than the number of students. Most of the time, the number of professors is about half the number of classes, which will grow as the number of classes grows. Therefore, we see that the algorithm for extension 5 takes $O(t * r * p)$.

2.7.4 Experimental Result

We see that the table shows optimality values for each semester for a single run of our algorithm. Since our algorithm shuffles the students to simulate random order in which students register, we see that each run of the algorithm yields a different schedule and a different optimality score as a result of studentss individual schedules changing based on not being able to register for a course because of room capacity. Overall, we saw a decrease in the optimality score when compared to the original algorithm, which we expected because we essentially assigned classes based on levels instead of popularity. However, the metric of success for this modified algorithm is not measured by how many students obtained their preferred courses, but rather by the metric that students who register first usually get the courses they want because they do not have to worry about room capacity. As such, there would not exist a student preference input or preference value for this modified algorithm because students would be the ones avoiding conflicts and creating their own schedules while the registrar locks students out of full classes.



Year	Classes	Profs	Students	Times	Rooms	Time (s)	Best	Experimental	Optimalit	Diff Exp	Diff Opt
2000	231	141	1112	17	60	0.015668	3497	3057	0.874178	-71	-0.0203
2001	222	151	1096	17	59	0.015668	3542	3125	0.88227	-48	-0.01355
2002	239	144	1090	17	61	0.015621	3579	3094	0.864487	-21	-0.00587
2003	241	132	1104	17	59	0.015622	3539	3157	0.89206	5	0.001413
2004	265	142	1124	17	51	0.015671	3700	3211	0.867838	-97	-0.02622
2005	255	147	1127	17	52	0.01568	3680	3173	0.862228	-93	-0.02527
2006	269	149	1167	17	63	0.01562	3798	3312	0.872038	-67	-0.01764
2007	283	152	1148	17	62	0.015667	3862	3329	0.861989	-120	-0.03107
2008	284	156	1213	17	63	0.015667	3794	3367	0.887454	-33	-0.0087
2009	264	146	1352	17	67	0.015621	4057	3612	0.890313	-48	-0.01183
2010	288	159	1475	17	68	0.01562	4466	4030	0.902373	-26	-0.00582
2011	280	157	1600	17	64	0.015621	4671	4238	0.9073	-33	-0.00706
2012	293	157	1659	17	70	0.015623	4813	4330	0.899647	-70	-0.01454
2013	320	163	1644	17	69	0.015609	4739	4324	0.912429	-9	-0.0019
2014	280	160	1635	17	67	0.004786	4558	4138	0.907854	-27	-0.00592

We computed a mean statistic of 88.6 percent optimality for our first come first serve model algorithm. We did see about a 1.2 percent decrease from our mean optimality of the original algorithm. Running our algorithm on real data, we observed that there were many cases where a student could not enroll in a course because the room capacity had been reached. The room capacity became as much of an issue as student timeslot conflicts, which contributed to the loss in optimality. Although the results show that the algorithm might not be optimal for when there does exist student preferences, we modified the algorithm for a completely different situation where students simply just need to register for courses based on their own volition and if the students are not able to register for the courses they wanted to, then they have the option to sign up for lower level courses which we put in the biggest rooms.

2.7.5 Additional Data Structures

Since we needed to make our first come first serve model accurately mimic students choosing their classes we needed to randomize the order in which students are being assigned to their chosen classes. To do so, we created a list called rand-order which is of size S that contains the list of all students but with their order shuffled. We see that the shuffle operation takes $O(s)$ because there are s students in rand-order. We then iterate through rand-order to get the order in which each student registers and we access input S to get the classes each students wants (all classes have already been assigned to a timeslot and room). The idea behind randomizing the student preference list is to simulate the random order in which students sign up for courses such that the students at the front of rand-order are equivalent to the students who register for courses first.

We also had two arrays of classes, of which one is called finalized schedule and the other is called working schedule, both of which hold the same class objects except that we will sort working schedule by class level. Well, each schedule takes $O(c)$ to construct, and we sort working schedule by increasing class level which takes $O(clogc)$.

We still also have pTimeslots and sTimeslots from our original algorithm to check for professor and student conflicts. pTimeslots and sTimeslots take $O(p)$ and $O(s)$ respectively to initially construct as noted before in our original algorithm. We can also update and access values in these two arrays in $O(1)$.

2.7.6 Discussion

For constraint 5, we found it difficult to actually measure the success of our algorithm because the idea was based on assigning classes to rooms and timeslots beforehand and letting students register themselves which is how scheduling for courses works in a real situation. The registrar does not have preference values to assign classes before students register, so we based it on levels of courses instead so that most students would be able to sign up for introductory courses if they do not get into their first pick classes. Also, our algorithm or model differs from the current Bryn Mawr model of a lottery because we implemented an algorithm under a model that once a student registers for a course, then they are enrolled in the course. In our case, there would be no lottery and a student is ensured to be in a course if they were able to register for it. Of course, exceptions could be granted for students who need a particular course to graduate on time. But, this model was inspired by looking at how other colleges handled student registration and from experience, they employed a first come first serve model because room capacity was not an issue at bigger institutions. We wanted to see how the model worked for a small liberal arts college such as Bryn Mawr, and concluded that doing a first come first serve model might not be ideal because room capacities limit the number of student enrollment.