CS340 - Final Project Paper

Linh Le, Rachel Lee, Angie Yang

December 7, 2020

1 Abstract

To solve the registrar's problem of creating schedules for a semester of classes, we designed an algorithm focused on the conflicts between two courses. The basic order in which the algorithm creates a schedule is that it first pre-processes the data, then schedules classes into timeslots, classes into rooms, and finally students into classes based on student preference lists. After the schedule is generated, the algorithm also computes the student preference score, putting the amount of classes that students were enrolled in from their preference list over the total number of classes in the student preference lists, i.e, the maximum score.

Using basic, computer generated data, we ran multiple simulations. With this, our algorithm created schedules that had student preference scores within the range of 78.42% and 97.58%. We then modified our algorithm to handle real data from Haverford College in addition to further extending the program to five extensions with the college data. Some extensions we decided to implement were prioritizing introductory courses, major requirements, and also STEM classes. The student preference score that our algorithm yielded with Haverford data before the extensions was 70.55% and the score after the extensions was 71.01%. After completing this project, we found that increasing the amount of time slots in a given semester was the most effective in increasing the student preference score. We think that this could be a good way to have more students get the classes they prefer and to spread out courses throughout the week so there are not too many classes jammed into one time slot.

2 Algorithm Analysis

2.1 Description

Initially, all students in S, classrooms in R, professors in P, and time slots in T are not assigned. We create a conflict matrix from the students' class preferences, which are scored based on how many students wish to enroll in a specific pair of classes. The conflict matrix is sorted in decreasing order from the most desired pair of classes together. The popularity of each individual class (how many times it appears in the entire student preference list) is also calculated. We also update the number of available slots remaining for each time slot after an assignment of a class to a time slot.

Schedule classes into time slots:
Our goal is to schedule each class into a viable time slot to maximize student enrollment. For each pair of classes in the conflict matrix, starting with the most desired pair, we make sure to schedule both classes into separate time slots if at all possible, and assign each of them to the viable time slot with the most availability currently. Viability is determined by lack of professor conflict (i.e. the professor cannot teach 2 classes at the same time.) At the end of this process, each time slot will be assigned a list of classes.

Schedule classes into rooms:
For each time slot, we sort the assigned classes by most to least popular. We schedule each class into a room, assigning the most popular class with the largest room, and so on, until all classes have been matched with a room.

Schedule students into classes:
By this point, we have classes (and professors) assigned to rooms and time slots. We go through each student and enroll that student in every course from their preference list as long as the class has space and the class fits in their existing schedule. We repeat for all students.

2.2 Pseudocode

```
Function preprocess(file-students, file-constraints)
    read files for num-t, num-s, num-r, num-c, num-p, room sizes
    s-pref[][] = index: student ID, value: preferred classes
    profs[] = index: class, value: professor
    class-pop[] = index: class, value: popularity score
    conflict() = key: class pair, value: priority score
    sort conflict()
return
```

```
Function sameProf(profs, int classId)
    cur-prof = profs[classId]
    for each class c different from classId do
        if prof[c] == cur-prof then
            return professor's other class
        end
    end
return

Function timeForClass(conflict, num-t, num-r, num-c)
    visited[] = index: class, value: boolean
    timeSlots[][] = index: time slot, value: list of classes
    slotAvail[][] = time slots and their remaining slots
    classTimes[] = index: class, value: time slot for pair,score in conflict do
        separate class1, class2
        paired1 = sameProf(profs,class1); paired2 = sameProf(profs,class2)
        if !visited[class1] then
            for slot in slotAvail do
                if slot unfilled and no paired1 then
                    add to slot
                    break
                end
            end
        end
        sort slotAvail in decreasing order of availability
        if !visited[class2] then
            for slot in slotAvail do
                if slot unfilled, no paired2 then
                    if exits another slot with no class1 then
                        add to slot
                        break
                    end
                    else
                        add to last remaining slot
                        break
                    end
                end
            end
            re-sort slotAvail
        end
    end
return timeSlots, classTimes
```

```
Function roomForClass(class-pop, timeSlots, num-r, num-c, roomSize)
    classrooms[][] storing classes and their assigned rooms
    roomSize = list of rooms sorted in decreasing capacities
    for each time slot t do
        for each class assigned to t do
            get individual popularity
        end
        sort classes in order of decreasing popularity
        for every class in sorted order do
            match with index-correspondent room in roomSize
        end
    end
return classrooms
```

```
Function studentForClass(classrooms, timeSlots, classTimes, s-pref, num-s,
 num-c)
    score = 0
    registr[][] = index: class. value: list of assigned students
    for each student do
        list of registered times
        for each class in student's preference do
            if class not full and no time conflicts then
                register the student
                score ++
            end
        end
    end
return registr, score
```

2.3 Time Analysis

Reading in the files requires for-loops for students, S number of iterations, classes, C number of iterations, and rooms, R number of iterations. This step totals to $O(S+C+R)$ time.

A class popularity array is created that stores the number of times each class appears on students' preference lists. A conflict matrix is created that stores every pair of classes that appears in students' preference lists, and the number of times that pairing is seen. To create the popularity array and conflict matrix, each student's preference list is iterated through. Within the outer for-loop, two inner for-loops will repeat a constant 17 times. This results in $O(S)$ time.

In order to assign each class to a time slot, the conflict matrix needs to be sorted by a pair's number of conflicts. There can be a maximum of C^2 possible class combinations. The sorting of pairs takes $C^4 \log C^4$ time. The algorithm then iterates through each of these class pairs to assign each of the 2 classes, $class1$ and $class2$, to a time slot. For each of the classes, the algorithm finds the other class taught by the same professor, since it is assumed that every professor teaches two classes. This process takes C iterations to search for each class. Therefore, in total there will be $C \times 2C$ iterations, or O^3 time.

After finding the professors' classes, the algorithm will find a time slot in which $class1$ and $class2$ can fit. For each time slot, we check that the professor's class is not in the time slot, if the time slot already exists, and for $class2$, if $class1$ is already in the time slot. All of these checks take a maximum of $2R + T$ per class. After having assigned the class a time slot, we sort the time slots by availability, so the time slot with the most remaining space (based on the number of rooms) can be considered first when assigning courses. Sorting the slot availability array of time slots takes $O(R^2 logR^2)$ time. Since the algorithm will only attempt to look for a time slot if a class has not been visited yet, this process will repeat a maximum of C times, regardless of the number of class pairs in the conflict matrix. Therefore, the time analysis of this portion is $O(CRT + CT^2 + CRlogR)$.

To assign the suitable class size for each class, we need to sort the room and the popularity of classes in the same time slot, both in descending order. The sorting operation of the rooms with the list data structure takes O($RlogR$) time. We then iterate through each time slot. For each time slot we get the class popularity of each class in the slot ($O(R)$), sort those classes by class popularity ($O(R^2 logR^2)$), and assign each of those classes to a room according to their popularity ($O(R)$). After all of the time slots have been completed, we have our list of classes already assigned to suitable rooms, and one final sort is performed which takes O($ClogC$) time. In total, the time analysis of this method is

$O(TR^2logR^2 + C^2logC^2)$

The next method the algorithm assigns students to classes. In this method, first create a registration array of size C, then we iterate through each student and attempt to enroll them in each class from their specific preference list. Since each student has 4 classes in their preference list, this method will take $O(C+4S)$ time which boils down to $O(C+S)$.

Finally, the schedule method prints out the final schedule. It loops through each class and prints out the corresponding professor, time, room, and student list which takes $O(C)$. Since each student can be assigned at most 4 classes, the printing also can take a maximum of $O(4S)$ which comes to a total of $O(C + S)$ for printing out the table.

If we bring all the methods together, the algorithm will take $O(S + C^4logC^4 + CRT + CT^2 + CRlogR + TR^2logR^2)$. If we assume that the number of rooms multiplied by the number of time slots must be greater or equal to the number of classes since there must be a time and place to accommodate all classes, and that RT and T^2 are less than C^3, then we can further simplify this to run time of $O(S + C^4logC^4 + TR^2logR^2)$.

In order to guarantee this total run time, the following operations must be done in constant time.

1. Get each student's list of preference
2. Update and get the number of times a pair of classes is preferred by a student.
3. Get, append, access an element.
4. Get the other class taught by the same professor.
5. Check if a class has already been assigned.
6. Check if a time slot is empty, or if 2 paired classes might be assigned to the same time slot.
7. Update and get the number of times a class is preferred by a student

2.3.1 Data Structures

For this program, we read the input files into: 1) A 2D array keeping the lists of preferred classes by each student, with the student integer ID as indices; this data structure allows O(1) value accessing time, which lets operation (1), (3), and counting number of students interested in taking a class be done as quickly as possible. 2) A list with its indices as rooms int IDs to keep the capacity of all rooms, allowing retrieving information of a room's capacity in O(1). 3) A list with the indices representing the classes, and values as the professors' int IDs. Similarly, this data structures lets (3) be done in O(1). 4) A list to store all the possible pair combinations of among C classes, allowing quick manipulations of each element in a pair. 5) A dictionary with the all class pairs as keys and their counts as values, which acts as a conflict matrix. As we traverse through the list of all possible class pairs, once we see a class pair being chosen by the same student, we increment the count to decide the order of processing based on how close the correlation is. The dictionary data structure allows this operation (2) to be done in O(1), and finish building such a dictionary takes O(C^2) for C^2 pairs of classes. 6) A list with the indices as class id and value as how often each class appears on students' preference lists. Since the indices correspond to the classes, we can easily update the values when a student lists a class on their preference list, and we can easily access how popular a class is (7), which we later use to assign rooms to classes.

For arranging classes to available time slots, 2 arrays are used to mark if a class has already been assigned and record the time slot each class was assigned to. The array data structure allows (3), (5) to both be done in O(1) time. A 2D array with the professors ID as indices and 2 paired classes is used to get the class taught by the same professor for the purpose of checking before adding a class into a time slot. With the array data structure, this operation (4) takes O(1) time. Another 2D array is used to keep track of what classes are assigned to one particular time block. Similarly, getting the list of classes in each time slot, appending classes, and (6) checking if a time slot is empty all takes O(1) time. Such an array takes O(RT) to build, with R maximum classes in one time slot and T available time slots.

For assigning suitable rooms to classes, taking number of interested students into consideration, we utilize an array to sort R maximum number of classes in each time slot from most popular to least, so classes can be paired with a the sorted list of rooms. All operations are done in O(1).Additionally, an array of arrays is used to store the result of the function, with each smaller array being the class ID and the assigned room. This structure allow us to check the room assigned to each class very easily in O(1). Building this structure takes O(C) time, since there is a maximum of C successfully assigned classes.

Finally, to assign students to classes, we use one 2D array and one smaller 1D array. The smaller array is initialized with each student to keep track of the time slots in which they have already been enrolled in a class. Add operation takes O(1). Checking if a time slot is already in the array only takes at max O(4) for each student, since each student can only enroll in 4 classes max. The 2D array has class IDs as its indices, and values being lists of students enrolled in each class. Append operation on this data structure takes O(1), and building such an array takes O(C).

Since the operations carried out by the algorithm relating to each data structure is completed in time less than O($C^2 log C^2$), the algorithm runtime will not rely on the time it takes to run the operations mentioned in the Data Structures sub-section.

2.4 Proof of Correctness

Proof of Correctness. To prove that this algorithm works, two parts need to be shown.

1. Proof of Schedule Validity
2. Proof of Termination

Proof of Schedule Validity. In order to prove that the schedule produced by this algorithm is valid, we must prove that the schedule follows 5 rules:

1. There is no professor that teaches two classes at the same time.
2. No room is scheduled to hold two classes in one time slot.
3. All schedulable classes have been scheduled.
4. No classes are scheduled more than once.
5. No enrolled student has a schedule conflict.

Proof of no professor conflict (by contradiction):
Let us assume that the schedule produced by this algorithm assigns one professor to teach two classes in one time slot. This would mean that the two classes would be placed into one time slot. However, every professor's two classes were already placed into $profs$ in the previous step and the algorithm consults $profs$ before assigning classes to time slots, making sure that the classes in pairs in $profs$ do not get placed in the same time slot. So, the professor's classes could not have been placed in the same time slot. This contradiction proves that the schedule will not have any professor conflict.

Proof of no room conflict (by contradiction):
Let us assume that in a schedule produced by this algorithm assigns two classes, c_1 and c_2, to occur in the same room r_1 during the same time slot. According to our algorithm, in a given time slot, classes are assigned to rooms according to student preferences and the size of the room. Since only one class can be assigned to one room in a given time slot, if c_1 is assigned to r_1, then no other class can be assigned to room r_1 in this time slot. Since both c_1 and c_2 were assigned to r_1, then they must not have belonged to the same time slot. This is a contradiction, proving that the algorithm will not produce a room conflict.

Proof that all schedulable classes have been scheduled (by contradiction):
Let us assume that in a schedule produced by this algorithm, there exists a class c_1 that is schedulable but was not scheduled. A class is defined as schedulable if there is an available time slot with no professor conflicts (i.e. there is a time slot for the class that does not overlap with another class the professor is teaching) and the class is listed on at least one students preference list (i.e. the class would even have students enrolled). Since c_1 is schedulable, then it should be listed on at least one student's preference list. Based on

student preferences, the list of pairs *sList* is produced and iterated through. For a pair of classes that include c_1, c_1 must be assigned to a time slot that does not conflict with it's professor's other class. At this step, c_1 is only not scheduled if there is not an available time slot that does not conflict with the professors other class. Since the class was not scheduled, then there must not have been a time slot that did not have a professor conflict. This is a contradiction of the assumption, and so it is proved that all schedulable classes have been scheduled.

Proof of no classes scheduled more than once (by contradiction):
Let us assume that there is some class b that has been assigned to more than one time slot and one classroom in the schedule created by the algorithm. Before assigning a class pair to time slots, the algorithm always checks both classes if they already have been added to a time slot. If only one class in the pair is assigned and there are slots available that do not conflict with the class already assigned, the other class in the pair will be placed in that slot. If both classes in the pair are already assigned, the algorithm continues to the next pair, making no changes. So, b could not have been placed in more than one time slot, as the algorithm would not allow this.
After a class is assigned a time slot, the class is then assigned to a classroom. Since a class cannot be scheduled in more than one time slot and every class in the time slot is matched with one classroom each, no classes can be scheduled more than once. Using this algorithm, b would only be assigned to one time slot and one classroom, reaching a contradiction with the assumption and proving the statement.

Proof of no scheduling conflict for enrolled student:
Let us assume that there exists a schedule for student s_1 where s_1 is enrolled in two classes, c_1 and c_2, but c_1 and c_2 are scheduled for the same time slot. According to the algorithm, students are enrolled in classes one at a time, iterating through the classes on their preference list. For a class to be added to a students schedule, this class must not have the same time slot as any prior class that the student enrolled in. Since s_1 was able to enroll in c_1 and then c_2, it must mean that c_1 and c_2 do not share the same time block. Thus we have reached a contradictions of the assumption and we have proved that there must not be scheduling conflicts for enrolled students. □

Proof of Termination. Since the algorithm only iterates through each list of variables (classes, professors, students, time slots) and each list has a finite size, the algorithm is proven to terminate. □

□

3 Experimental Analysis

3.1 Stress Testing

For stress testing our algorithm, we focused on one variable at a time. Our control case had 50 rooms, 260 classes, 24 professors, and 1200 students. We decided these numbers based on Haverford College's real data. Since the basic algorithm assumes all time slots are unique, we adjusted the amount of time slots to account for this overlapping. We then stress tested on each variable 3 additional times, keeping all other variables the same and changing one variable, making sure there was a wide range of values spanning from small to large numbers.

Changing the number of rooms to feed to the algorithm did not affect the score very much. It stayed pretty much the same, within a range of .06%. Perhaps the smallest number of rooms we tried, 11, was enough for the algorithm to comfortably create a schedule. We noticed clear trends similar to what he had expected for the other three variables. As we increased the number of classes, the fit score went down. Since there are more classes, but the same amount of time slots, it is easy to imagine that there would be more conflict in trying to enroll students in their preferred classes. So, as we added more time slots, the score improved drastically. Because we did not change the number of classes, there were less classes per time slot and less conflict, allowing the algorithm to cater to the students' preferences more effectively. As we increased the number of students, the score decreased. This is also understandable because the more students there are, the harder it is to accommodate every student and schedule the classes they desire.

3.1.1 Real Time Analysis

Time Analysis: Verify that your algorithm performs as expected based on the theoretical time analysis (included in the write-up described in Section 3.1). Describe your experiment design and explain how the resulting numbers verify the time analysis.

Using the computer generated data with $|R|$=50, $|C|$=260, $|T|$=24, and $|S|$=1200, the time that the algorithm took run was 0.1049 seconds. Our theoretical time analysis was $O(S+C^4logC^4+TR^2logR^2)$. According to this theoretical time analysis, with these values, we would expect to get a runtime of $O(1200+260^4log260^4+24*50^2log50^2)$. This simplifies to $O(146641860611.8024)$. We double $|S|$ to 6500 and keep all the other variables the same. The algorithm takes 0.3235 seconds. The calculated runtime using these variables is $O(146641861811.8024)$

With these two experiments, it is difficult to see a clear relationship between the theoretical runtime we calculated and the real runtime we achieved when running our algorithm. This may be due to the many smaller variables and constants we dropped from the equation when reducing the equation down to Big O notation. Perhaps the scenarios we tried were not drastic enough to have a big impact on the calculated runtime; increasing students more may have proved to be closer to what the calculated runtime would be.

Based on our graphs, the relationship between constraint variables and runtime can also be seen. Figure 5 shows runtime as a function of the number of students since we keep every variable but students the same. As students increase, runtime is shown to directly increase as well. The clear linear trend shows implies that the number of students is used in a linear (rather than polynomial or logarithmic) factor in the runtime. This is reflected in our time analysis equation as only $|S|$ with no exponent appears in the equation $O(S + C^4 logC^4 + TR^2 logR^2)$.

Figure 6 shows runtime as a function of the number of classes since we keep every variable but classes the same. As classes increase, the runtime is shown to increase as well, but in a nonlinear way. The trend line that fits this experimental data best is a polynomial trend line. This is reflected in our time analysis equation as $|C|$ is used in $C^4 logC^4$. Figure 7 and 8 show less distinct trends and the number of rooms and time slots seem to have less affect on the runtime of the algorithm. This can be explained by R and T being relatively small values respective to S and C.

3.1.2 Graphs

Below are some graphs to map out stress testing data and create a visual representation.

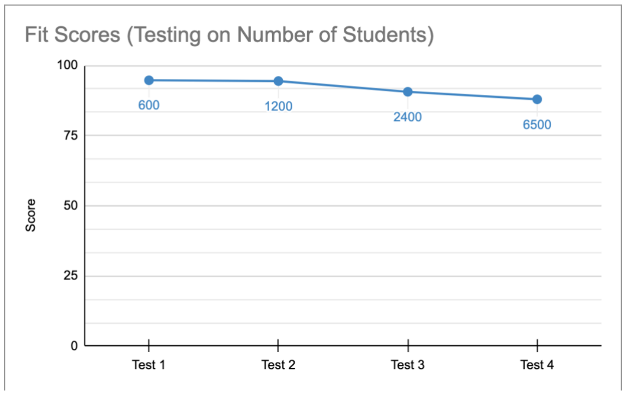

Figure 1: Fit Score of Schedules (Testing on Number of Students)

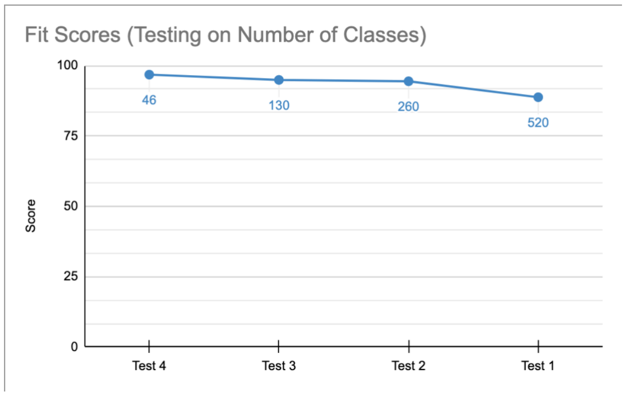

Figure 2: Fit Score of Schedules (Testing on Number of Classes)

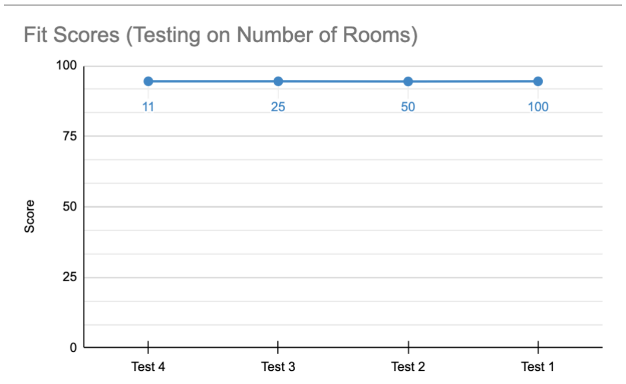

Figure 3: Fit Score of Schedules (Testing on Number of Rooms)

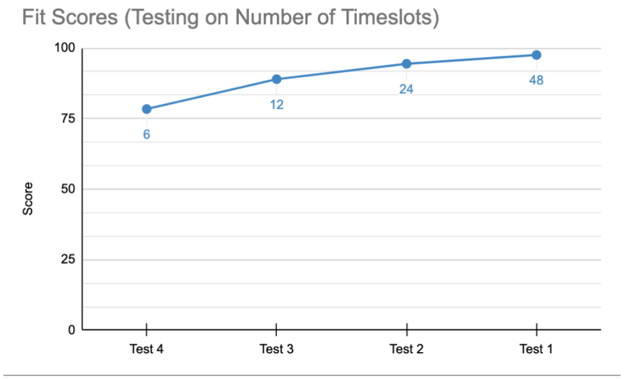

Figure 4: Fit Score of Schedules (Testing on Number of Time slots)

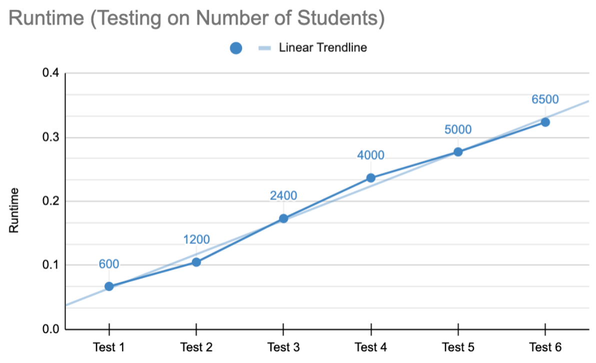

Figure 5: Runtimes of Creating Schedules, Testing on Number of Students

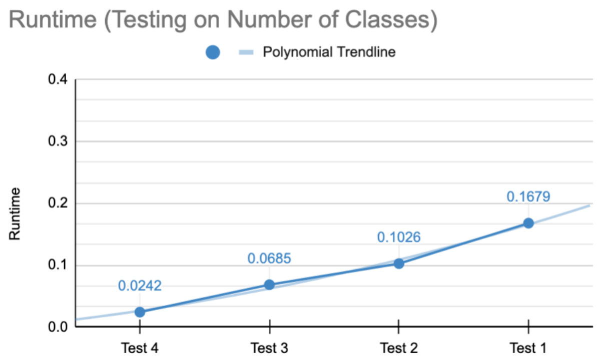

Figure 6: Runtimes of Creating Schedules, Testing on Number of Classes

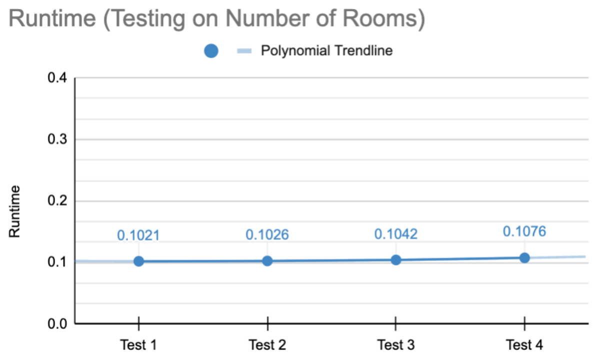

Figure 7: Runtimes of Creating Schedules, Testing on Number of Rooms

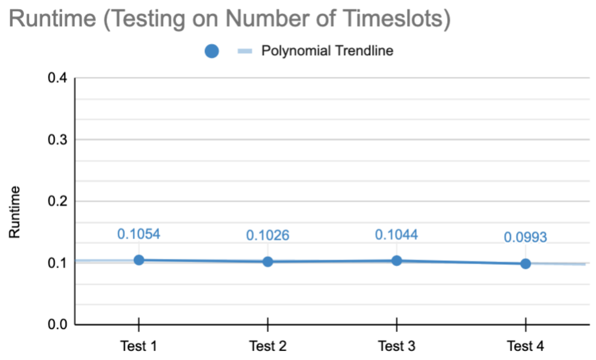

Figure 8: Runtimes of Creating Schedules, Testing on Number of Timeslots

3.2 Discussion

3.3 Solution Quality Analysis

Originally, our algorithm assigned classes to time slots with some extent of randomness: as soon as it found an unfilled time slot without a conflict in the professor's schedule, the class was automatically considered assigned. The percentage of fit was fairly low, at 69 percent, due to this implementation. We realized that this approach caused many time slots to be filled up much quicker, thus limiting the number of time options available for classes that were processed later in the list.
To resolve this issue, we are taking each time slot's number of remaining available spaces into consideration. Classes will be put in the time slots with the greatest number of remaining capacities (at the time of assignment) that meet the requirement of no professor's conflicts. This way, the classes will be spread out more evenly across the time slots, leaving more time options for the incoming classes to be assigned. With this improved approach, our algorithm achieved a lower bound of 78.42 percent and higher bound of 97.58 on the basic generated data.

3.4 Extensions

We modified our algorithm to accommodate Haverford College data and to factor in 5 extensions as followed:

1. Students can prefer a maximum of 5 classes instead of the usual 4.
    - Modifications to the algorithm: In order to allow students to prefer 5 classes, we decided to not put a cap on the number of classes the algorithm could accept.

We then programmed the algorithm to find combinations of two for as many classes as they had, and not just for the first 4.

2. Classes that are in the introductory level (level 1) are prioritized.
    - Modifications to the algorithm: We changed the file that read in the Haverford data to also read in data regarding the level of each class. We also modified our algorithm so that it would assign more weight to a class pair if one class was an introductory level course with a level of 1.
3. Prioritize pairs of courses in one student's preference list that are in the same subject (presumably major requirements).
    - Modifications to the algorithm: For this extension, we also had to add a few lines of code to the file reading in Haverford data to read in the subject each class was in. Then, we checked if both classes in a pair were of the same subject. If they were, we tried to prioritize that pair by assigning more weight.
4. Give extra weight to STEM courses in students' preference lists.
    - Modifications to the algorithm: We hard-coded an array of subjects that we considered to be STEM subjects. Then, when the algorithm assigns weight to each pair, we also had the algorithm check if any class in the pair was a STEM course. If it was, the pair recieved a greater weight.
5. Make sure timeslots do not overlap.
    - Modifications to the algorithm: In order to do this, we created a new dictionary full of timeslots and a corresponding list of timeslots that a certain timeslot conflicts with, making sure to check all cases of overlap. Before assigning a class to a timeslot, the alorithm checks if there is a conflict.

3.4.1 Analysis

Right after we modified our algorithm to handle Haverford's real data, before we coded any extensions, the student preference score was 0.7055. The algorithm took 0.0568 seconds to create a schedule. After we wrote in the extensions, the score rose a small amount to 0.7101 and the time had no substantial change at 0.0541 seconds. Although negligible, the algorithm with extensions actually completed slightly faster than the basic algorithm without. We were a bit surprised by this, especially considering that we created a whole new data structure to hold the overlapping timeslots. The algorithm is also longer and has to consider more cases as it processes through each class pair. Despite these factors that may have made the algorithm worse (time and score wise), there are also places were the algorithm was improved with these extensions. For example, since many introductory classes tend to have greater capacities, more students that prefer those classes would be able to get enrolled. This was also a factor behind why we chose to implement this extension, so more students could get into these introductory courses. It is also possible that higher level courses that have more prerequisites and are more specific to a major have a smaller number of students wishing to enroll. We think that adding weight to introductory courses and pairs of classes in the same subject possibly helped more students get into the courses they prefer.

3.4.2 Recommendations

After contemplating this problem for months, piecing out our own algorithm, and analyzing our experimental results and graphs, we have come to realize just how complex the issue of class registration is. However, we do have some recommendations for the registrar that could be of use. Especially based on our stress testing, there is a clear trend of student preference score with number of classes, number of students, and number of timeslots. Although the registrar may not be able to change the number of students or the number of classes, it could be possible to increase the number of timeslots. By offering a larger variety of times when classes are held, the registrar could accommodate for more student's preferences. This could possibly mean more satisfied students who are getting into more of their preferred classes, as well as students who are less worried about fitting in all their major and college-wide requirements.

Since the registrar already prioritizes major requirements and makes sure that timeslots do not overlap, there is not much in our extensions that we can recommend to the registrar. Oftentimes introductory courses also have a large number of students who want to take the class, over-enrolling the class. If the registrar were to prioritize introductory courses for everyone who enrolls, it would make no difference since some people inevitably will have to be dropped from the course.