CS340 - Scheduling Project

Dylan Emery, Skyler Ellenburg, Noah Weinstein, Lake Giffen-Hunter

April 4, 2017

1 Abstract

Our algorithm uses class popularity (number of students who want to take that particular class) to assign classes to time slots. Giving the most popular classes priority, it moves through the list of classes and assigns classes to time slots based on the number of students who want to take the current class and any one class that is currently in the time slot. When the algorithm finds how many students want to be in any two particular classes, for each class it moves through each other class to find the number of conflicts for each class pair. It also iterates through each student when assigning students to classes, making the algorithm run in $O(c^2 + s)$. Because it minimizes the number of students who want to be in two classes that are in the same time slot and makes it so that more popular classes are generally in time slots with less popular classes, our algorithm creates a schedule that is very close to optimal for students.

In order to make the algorithm more similar to Haverford's class scheduling process, we implemented five unique extensions. Our first extension involved including dropped classes in the algorithm under the logic that a dropped class indicates some level of interest in taking that class. We found that including dropped classes is a good way to make a schedule more optimal for student preferences. It may be that students are dropping a course because it conflicts with another popular course, in which case not recognizing that conflict may lead to similar future unhappiness. The second extension involved ensuring that preference went to class majors when deciding who could stay in an overflowing class. This increased the number of students that we were able to schedule significantly, meaning the registrar should keep this practice when filling students into classes. Our initial algorithm didn't count unique time slots as distinct slots; classes in it were not considered to conflict with any class in a different time slot. In reality though, a MWF class that meets from 2:30pm-4pm conflicts with a F only class that meets from 1:30pm-4pm. The third extension we implemented checked for time slot overlap and counted classes that were scheduled in the same time frame as overlapping. This didn't affect the algorithm's optimality significantly and made the results more practicable. For our next extension, we consider if splitting

up the time slots could result in a better schedule. To clarify, the 10:30 to 11:30 Monday Wednesday Friday time slot is *actually* three time slots: 10:30 to 11:30 on Monday, 10:30 to 11:30 on Wednesday and 10:30 to 11:30 on Friday. We examine the results of splitting up the time slots in this manner and then assigning each course for three hours each week. In our final extension, we limit the rooms in which the teachers are allowed to teach to one building, since the different departments at Haverford tend to teach in certain buildings. Then each course can be scheduled into a subset of the larger set of rooms based on which professor teaches it. We assign rooms using this new limitation and examine the effects on the number of student preferences that are met. Based on an analysis of these five extensions, we can make better recommendations for Haverford's class scheduling process.

2 Algorithm Write-Up

2.1 Description

Our algorithm works by first generating a conflict 'matrix'. This 'matrix' allows for us to look up the number of conflicts between any two classes. We set classes taught by the same teacher to be infinite conflict, as these classes can never be put together. We also generate a list of tuples, (class popularity, class), and order it bases on popularity. We take the most popular class and add it to the first time slot, the second to the second, etc. until all of the time slots now have one class. We then try to add the next most popular class to a time slot until we find the slot that has the least conflict. We repeat this process until we have gone through the entire class list. Note that if a class time has been assigned a number of classes that is equal to the number of rooms we remove that time from consideration.

We have now generated a schedule of class times and need to assign rooms to the classes and assign students to classes. Assigning rooms is easy; since we added classes in the order of their size, for each time slot, we assign the biggest room to the first class we added, the second biggest room to the second class we added etc. Assigning students is a bit harder. First, we make a dictionary of classes where the values are the time slot they have been put in. Then we add all the students into the classes they do not have a scheduling conflict for; if a student has two (or more) classes they want to take that have been scheduled in the same time slot, we hold off on assigning them a class for that time slot. Once we have gone through these, we now consider potential 'over-scheduling'. We can check if the interest in a class is bigger than the room. So now for each student, we check to see if each class they want to take is 'over-scheduled'. If one of them is, and the other isn't, we put them in the one that isn't and remove the other from consideration. If both are, we assign them to the less 'over-scheduled' one. If they both aren't we can just add one arbitrarily and remove the other.

2.2 Pseudocode

We have three helper functions, *conflictMatrix*, *popularityList*, and *fillStudents*, along with the main function *classAssignment*.

```
Function conflictMatrix(studentPrefs, Teachers)
    Initialize empty dictionary $conflicts$
    for student in studentPrefs do
        for each unordered pair of classes in studentPrefs[student], $\{c_1, c_2\}$ where $c_1 \leq c_2$ do
            if $(c_1, c_2)$ is not in $conflicts$ then
                Set $conflicts[(c_1, c_2)] = 1$
            end
            else
                Increment $conflicts[(c_1, c_2)]$ by 1
            end
        end
    end
    for teacher $\in$ Teachers do
        $(c_1, c_2) = Teachers[teacher]$
        $conflicts[(c_1, c_2)] = \infty$
    end
    return $conflicts$
Function popularityList(classesList, conflicts, teachers)
    Initialize empty list popularities
    for class $\in$ classesList do
        Set popularity $= conflicts[(class, class)]$
        Append popularities with $(class, popularity)$
    end
    Sort popularities by decreasing popularity
    return popularities
```

```
Function fillStudents(Students, classTimesDict, roomDict)
    Initialize dictionary $studentsInClass$
    for student ∈ Students do
        for class ∈ studentPrefs[student] do
            if There is room in the class and it doesn't conflict with any other of the
              student's class preferences then
                append student to $studentsInClass[class]$
                remove class from $studentPrefs[student]$
            end
        end
    end
    for student ∈ Students do
        for class ∈ studentPrefs[student] do
            out of the classes that conflict with class, let $class'$ be the one with the
              least potential for overflow that has not filled if it exists
            append student to $studentsInClass[class']$
            remove class and all classes that conflict with it from
              $studentPrefs[student]$
        end
    end
    return ($studentsInClass$)
```

```
Function classAssignment(classes, roomSizes, classTimesSet, teachers,
  studentPrefs)
    Initialize empty classTimes dictionary $classTimesDict$
    Let $conflicts = conflictMatrix(studentPrefs, teachers)$
    Let $popularities = popularityList(classes, cpnflicts, teachers)$
    for $class \in popularities$ do
        $bestSlot =$ null
        $bestConflictNum = \infty$
        for $classTime \in classTimesSet$ do
            $tempConflictNum = 0$
            for $conflictingClass \in classTimeDict[classTime]$ do
                $tempConflictNum$ += conflicts[conflictingClass, class]
            end
            if $tempConflictNum < bestConflictNum$ AND
              $classTimesDict.length <$ the number of available rooms then
                $bestSlot = classTime$
                $bestConflictNum = tempConflictNum$
            end
        end
        append $class$ to $classTimeDict[bestSlot]$ if $bestSlot$ is not null
    end
    for $slot \in classTimseDict$ do
        Assign each class in $slot$ a room based on popularity, biggest room goes to
          most popular class, store this in $roomDict$
    end
    Now we fill students into classes
    $studentsInClass = fillStudents(Students, classTimesDict, roomDict)$
    return Dictionary mapping each class to its room, teacher, time, and students
```

2.3 Time Analysis and Data Structures

We claim that our algorithm runs in $O(c^2 + s)$ time where s is the number of students and c is the number of classes.

Let s be the number of students and c be the number of classes. We first need to consider the process of building the conflict 'matrix'. The way we build this is by going through each student and considering each possible combination of classes. We assign each class a number and maintain that the pairs are order e.g. for $(c_1, c_2), c_1 \leq c_2$. We then check a dictionary for each pair to see if that pair already exists as a key. If it does exist, we add one to it; we found another student with a conflict for this class pairing. If it does not exist, we create the key and make the value 1. We do this for each student, so if we let s be the number of students we end up doing $10 * s$ steps, which is $O(s)$. Given two classes, we can now find the number of conflicts between them in $O(1)$. Making the list of classes with their popularity can be generated from this matrix in $O(c)$ time, by looking up each double pair e.g. (c_1, c_1) and adding it to a list. We can then sort this list in $O(c * log(c))$, so the generation of this list happens in $O(c * log(c))$. We now go through the list of classes and compare it to each of the classes that have already been scheduled, which is at most $c - 1$ classes, thus this process is done in $O(c^2)$. Since $O(c^2) > O(c * log(c))$, we are left with $O(c^2 + s)$.

We also need to set the conflicts of classes taught by the same teacher to ∞. To do this, we just need to iterate through the list of teachers and access each pair of classes. This will take $O(t) < O(c)$ time, where t is the number of teachers. While we iterate through the teachers we can create a dictionary that maps each class to its teacher, this will take $O(c)$ time and will be useful when we return the final schedule.

This takes care of scheduling classes in time slots, but we still need to assign rooms and place students in classes. First we have to sort the list of rooms by size, which will be $O(r \log r)$. This will certainly be less than c^2 in any reasonable case of the problem. We will iterate through the time slot dictionary's lists of classes and assign the largest room to the first item in each time slot's list of classes, the second largest room to the second room, etc. This will result in the largest classes in each time slot being put into the corresponding largest rooms because both are lists sorted by size. In the loop we will fill a room dictionary that has a class as a key and the room it is in as a value. Initializing this dictionary will take $O(c)$ time. Each class is only assigned to one time slot, and we are iterating through the time slots only once, which means this loop happens in $O(c)$ time.

Next we need to handle placing students in classes. We create a dictionary of lists with classes as the keys called studentsInClass. These lists also need to track their length, but this is not a problem because we will only be adding to the lists, and we can increment the length in constant time. Initializing this dictionary will take $O(c)$ time.

Then we iterate through the list of students twice. The first time we then iterate through their preference list and then assign the student to any classes that do not share a time slot with any other classes in the student preference list. This requires a dictionary with classes for keys and the time slot of that class for a value. We can create this in $O(c)$ time by reversing our *classTimesDict*, which has a list of classes for every time slot. Then when we iterate through the student's preference list, we note the time slot of each class. If two classes share a time slot then we put them in a list inside of the student preference list. We do not add a student to any class that has been placed in a list during this first loop, but we do add each student to every class that is not in a list, as long as the class does not have more students than rooms. We remove any full class or class that a student has been assigned to from that students preference list. Also, as described above, we track how many students are in each class and we also have a dictionary of which room each class is in, so we can check this in constant time. Since the students preference list is a constant size, all of these calculations involving the list are constant, so the first loop runs in $O(s)$ time.

For the next loop we again iterate through the list of students. We removed any class that do not share a time slot in the students preference list, so all the classes left in a students preference list are grouped into lists where each list represents classes that share a time slot with each other. We need to be able to assign the student to the class with the lowest *popularity - room size* value that also has room in constant time. This is the class with the least possible overflow. We do this by looking up the popularity and room size in our dictionaries in constant time, then we can iterate through the list to find the class with lowest overflow, only counting classes that still have room. We can look up if a class has room left in constant time as described above. Thus, the second for loop runs in $O(s)$ time.

For either loop if all the classes are full, then do not add the student to any class.

Finally, we return a dictionary with classes as keys. Each value as another dictionary that maps the string 'Teacher' to the teacher of the class, 'Time' to the time slot of the class, 'Room' to the room of the class, and 'Students' to a list of students. We ignore any classes that are not in *classTimesDict*. We can create this by iterating through all of our dictionaries for each class, so this will be $O(c)$.

Our final complexity is $O(c^2 + s)$.

2.4 Proof of Termination

We have no while loops or recursion. All the for loops are guaranteed to terminate.

2.5 Proof of Validity

We claim that our algorithm always returns a valid schedule. That means it fulfills the following requirements (taken from the project description):

1. Every class is scheduled in one room, for one time slot, with one teacher and a list of enrolled students.
2. No class has more students in it then fits in that class's room.
3. No teacher is scheduled to teach two classes at the same time.
4. No student is scheduled for more than one class at the same time.

Proof. When we assign classes we give the biggest room to the most popular class in each time slot, then the next biggest room to the second most popular class, and so on. This ensures that each class is only scheduled for one room. We assume that each class is only taught by one teacher in the input, since the problem specifies that the number of teachers is half the number of classes and each teach teaches two classes. Additionally, when we place students in classes we produce a list of students for each class. This takes care of (1).

We know (2) holds because in all situations were we assign a student to a class, we first check to make sure the class has room. (3) holds because we set the conflicts between two classes taught by the same teacher to infinity. Assuming there is more than one time slot, no classes taught by the same teacher will end up in the same time slot because a class always gets assigned to the slot with the fewest conflicts. When finding the minimum conflicts, our starting value is infinity and we only change the minimum if it is strictly less than this vale, so a class will never be scheduled in a time slot its teacher already teaches a class. (4) holds because we only place a student into one of the classes for each set of classes that shares a time slot. Thus, our algorithm returns a valid schedule. □

3 Discussion

We believe our algorithm is effective because it quickly and directly pairs classes that have low amounts of conflict. This is the key to allowing as many students to be in as many classes that they prefer as possible. We decided to place classes into time slots first because based on the teachers and popularity we have the most control in ensuring that the classes with the most conflicts are placed in different time slots. We choose the rooms next because the rooms are affected by class size and time slots, so we already have the time slots chosen and the classes are organized by popularity, which can represent potential class size. Once classes have a time slot and room, we decide to assign students because these previous choices have been made based on popularity and conflicts, so conflicts are minimized and potential class size has been taken into account. With the student preferences, we decided to first put students in any non-conflicting class for clarity and a slight increase in speed since there are fewer checks in this step. Then we handle the conflicting classes by choosing the one with less overflow so that more students get into classes, and if a class does fill up we stop adding students to it. These choices serve to reduce conflicts and increase the number of fulfilled student preferences, so our algorithm is both close to optimal and fast.

Our main complication in choosing an algorithm was deciding on the hierarchy of the inputs; between classes, students, time slots and rooms we weren't sure what we consider first. With data structures, we started with adjacency lists and graphs but decided on dictionaries because of the quick look-up time and the relationships we want to represent. Based on our strategy and the choices we have made, the algorithm is greedy. We always add the classes that are most popular or rooms that are largest, and we only add to the choices made instead of making changes after a choice is made, i.e. we never remove a student from a class or change a room size. Our algorithm is similar to the min function when we are inserting classes into time slots because search through a list to find minimum conflicts.

We have examined the following possible extensions of our algorithm. Instead of four preferences, students could have five class preferences but still complete registration with four or fewer, and students could rank their class preferences. Right now the cap for classes is the room number, but we could give classes a cap initially. Also, if there is overflow for a class, each class could lists their priority for majors and minors and the students that are prioritized are registered for the class first. We are also very interested in looking at the actual data the registrar has to work with. For the final return statement, we intend to return a dictionary of dictionaries, where each class is mapped to a dictionary containing its teacher, room, time slot, and student list.

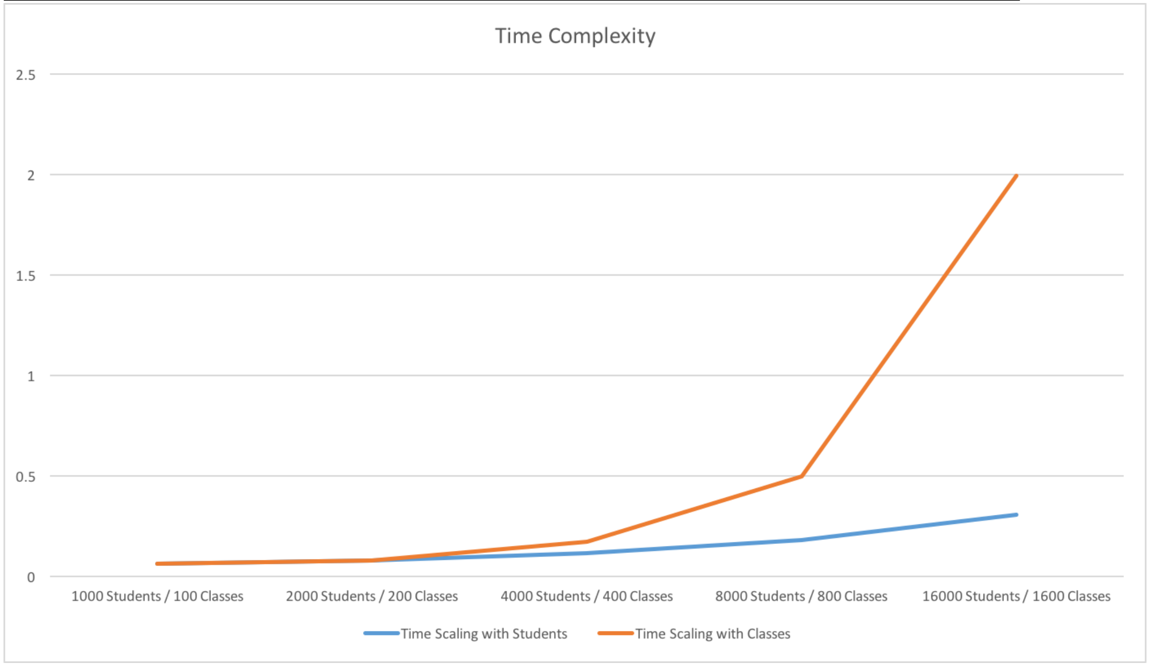
4 Experimental Analysis

4.1 Time

Running experiments revealed that our algorithm scaled approximately linearly with the number of students and quadratically with the number of classes, as expected. Our baseline for experiments was a school with 12 rooms, 100 classes, 10 time slots, and 1000 students. We decided that going below these values was not a realistic instance of the problem. Then we held all variables constant and varied the number of students by doubling it and recorded data for how our algorithm performed. We did the same with classes, but increased rooms and time slots as appropriate in order to contain the classes. In a standard case there will be significantly fewer rooms and time slots than classes and students, so we decided it was more important to focus on classes and students, especially since the number of classes usually correlates with the number of rooms and time slots.

The tables and graphs below show the result of our time experiments. Note that each time is actually the average time of 30 trials of our algorithm.

Students	Time Scaling with Students	Classes	Time Scaling with Classes
1000	0.064433	100	0.064433
2000	0.078666	200	0.077600
4000	0.113700	400	0.173633
8000	0.178833	800	0.499433
16000	0.308633	1600	1.99119



The data matches our expected time complexity. Doubling the number of classes about quadruples the time it takes for the algorithm to run and doubling the number of students about doubles the time. This matches the $O(c^2 + s)$ complexity that we predicted.

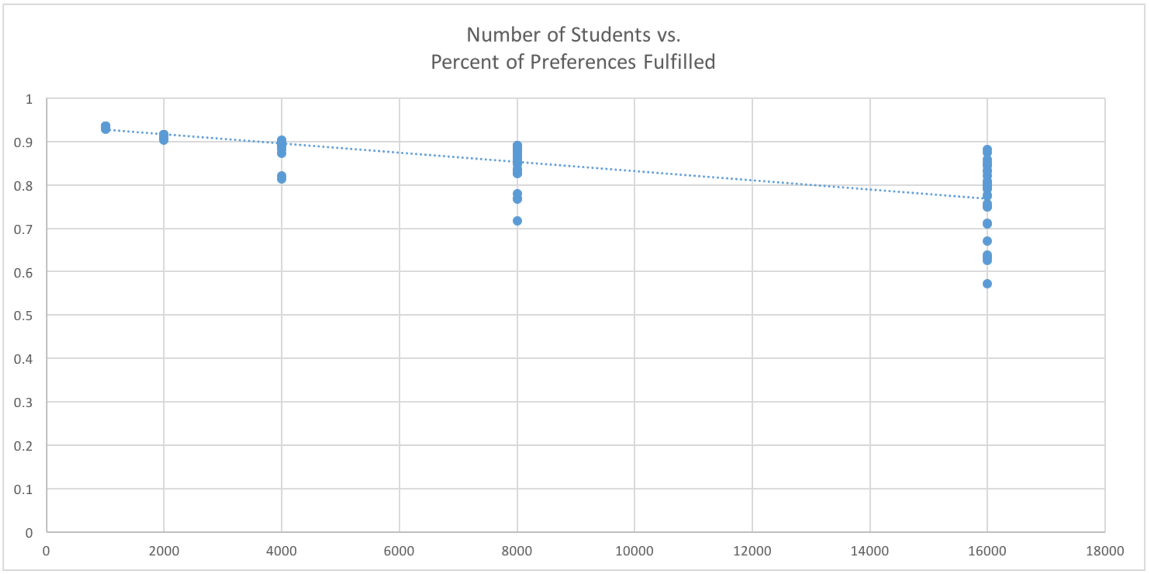
4.2 Optimality

We investigated how the optimality varied as the number of students varied. Changing the number of classes did not seem to have a significant effect on the optimality; increasing the number of classes only increased the end score. Therefore, the number of students is the most important variable to consider here. The table below summarizes our findings. We measured optimality by calculating the percent of preferences that were fulfilled. This is given by the algorithms preference value divided by the optimal preference value.

Students	Percentage of Preferences Fulfilled
1000	0.932466
2000	0.913650
4000	0.891577
8000	0.854518
16000	0.768720
32000	0.507344

In the first four rows the number of classes is held constant at 100, time slots at 10, and rooms at 12. For the last row classes are increased to 128, time slots to 11, and rooms stay at 12. This was essentially the bare minimum increase to support 32000 students.

Here we see a sharp drop off between 16000 and 32000 students. However, we decided that the 32000 student case is not a realistic example, since this is a huge number of students for the given size school i.e. the number of class rooms. In order to decide on the lower bound we examined all the experiments not just the averages. These are shown in the scatter plot below.



Using this information, it seems that for a realistic example our graph should not go far below 60%. When there are 60,000 students our lowest preference percentage is 57%, but this is the only point below 60%. If we only examine the 8,000 students and below cases, then the algorithm never goes below a 70% solution for 120 instances of the problem. Worth noting is that in a randomly generated Haverford sized instance of the problem (28 time slots, 50 rooms, 260 classes, 1100 students), the algorithm created a perfect schedule. Also, when run on the actual Haverford data, our algorithm fulfills 3167 out of 4004 classes, which is a fulfillment percentage of about 75%.

In conclusion, we think our algorithm can fulfill 70% of student preferences for most realistic constraints, but a safer lower bound would be 55%.

5 Extensions

5.1 Incorporating Dropped Classes

We noticed that the Haverford data contains both classes students ultimately enrolled in and classes that students dropped, most likely as part of shopping or just deciding which classes to take. The original script provided to pull data from the Haverford information ignored these dropped classes. However, we reasoned that a dropped class probably shows some level of interest, so we decided to see how the schedule improves if we include dropped classes in the preferences. We also added a limit of four to the number of classes a student can be enrolled in, since without this we would be enrolling students in more classes than they realistically could have planned on taking.

In order to implement this we first modified the get_haverford_info.py script to include dropped classes in the student preferences and constraints files. Then we modified our algorithm to not place a student in a class if they have already been placed into four different classes. None of this affected the original complexity of our algorithm.

When running our algorithm on the Haverford data with and without dropped classes we found that we got a student preferences value of 3124 without and 3850 with. There are 1153 students, so these are both out of a total possible student preferences value of 4612. This means that if we assume a student is interested in a dropped class, meaning they dropped it because of a scheduling conflict or because another class was only slightly more interesting, it is worth including the dropped classes in order to estimate interest for next year.

5.2 Majors

At Haverford, and especially in the Computer Science department, majors are given priority over non-majors when enrolling in the class. We wanted to find out if this preferential treatment had a positive or negative impact on the student preferences value as a whole.

In order to implement this we first modified the get_haverford_info.py script to also pull the subject for each class. Then, when it creates the student preferences, we had the script assign a major to each student based on the most common subject among classes they are interested in. Next, we modified our own algorithm to give priority to students whose majors matched the subject of a class they are interested in. We did this by placing students in classes that matched their majors before placing any other students in classes.

Running this extension on the Haverford data gave a student preferences value of 3561 versus the standard algorithm's value of 3167. This is unexpected; giving majors priority resulted in students getting into more of their desired classed. This is likely not true of every case. However, it does suggest that there is no penalty for including a major over a non-major, and since usually it is more important for a major to take a class, we recommend that majors have priority.

5.3 Overlapping Time Slots

Our initial algorithm treated each unique time slot as a distinct slot; classes in it were not considered to conflict with any class in a different time slot. In reality though, a MWF class that meets from 2:30pm-4pm conflicts with a F only class that meets from 1:30pm-4pm. This extension tracks the times and days of time slots and, when assigning classes to time slots, counts the classes in overlapping time slots as conflicting.

Initially, in course assignment, we create a dictionary, $classTimesDefDict$, that contains each time slot as a key, and a list of overlapping time slots as a value. In order to improve optimality, we sort the list of time slots by number of overlapping slots, in ascending order. This will ensure that the most popular classes are placed in the time slots with the least overlap with other time slots. As we assign each course to a time slot, we implement a second dictionary, $timeSlotConflictsDict$, that maps each time slot to the number of conflicts that particular course has with courses currently in that time slot. This memoization allows this extension to run in the same time order as our basic algorithm. For each time slot, it checks the conflict between the current course and the courses in the current time slot, and the courses in each of the time slots that overlap with the current time slot. Because *timeSlotConflictsDict* keeps track of the conflicts in any particular time slot after it has been iterated once, the algorithm only needs to run through the courses in each time slot a single time. It is also important to note that the $fillStudents$ function was slightly modified to account for each overlapping time slot when checking for class conflicts in a students preferences list after classes were assigned to time slots.

This extension does not have any negative impact on our algorithm's ability to create an optimal schedule for the Haverford data. It gave a student preferences value of 3167, as did the standard algorithm. This means in the case of that particular data set, our algorithm was able to deal with the time slots that overlapped in a way that did not hurt optimality. We created a slightly modified version of the Haverford Data CSV file and the value returned by the extended algorithm was, again, the same as the value of the standard algorithm. It is difficult to extrapolate how this would affect other data sets. However, it is interesting to note that when the time slots was sorted so the most popular classes were initially placed in the time slots that had the most overlap, there was a very small decline in optimality, down to a value of 3163. There was also a small decline, of 10 fewer classes scheduled, on the modified data set.

5.4 Breaking up Time Slots

Originally, our algorithm treated a time slot over multiple days as one time slot, but in reality, the 10:30-11:30 MWF time slot is 3 time slots. Currently, Haverford assigns classes in standard 'Monday Wednesday Friday' and 'Tuesday Thursday' options, as well as some one and five day a week classes . In this extension, we examine if we would be better off doing a less standard schedule e.g. having a 'Monday Tuesday Wednesday' class. Ultimately, we found that this 'wiggling' around of time slots made no difference, at least in our implementation.

In order to do this, we first needed to modify the data as we were taking it in; in 'wiggle_parse_inputs.py' as we take in the time slots, we split them based on the days of the time slot. For example, time slot 1 from 10:30-11:30 on Monday Wednesday Friday would have been broken into '1M','1W','1F'. We assume that each time slot is broken into three days; we will assign each class to three different time slots, thus simulating each class being scheduled for 3 hours a week. If some time slots were broken into two pieces we would end up with less time slots relative to the number of classes we are assigning. The algorithm still runs in this case, but it makes it difficult to meaningfully analyze how our modification is comparing to the standard algorithm. We also had to change the the constraints file and the student preferences file; each course gets broken into an 'a', a 'b' and a 'c' section. Each teacher was changed to match this in the constraints and the students preferences were also changed in reflection of this; if a student wanted to take course 1, they now want to take course 1a, 1b, and 1c. We make two new constraints files, and pass the modified data into make_schedule.py. In make_schedule.py, we now put each course into 3 different time slots, reflecting the fact that the time slots have been broken up and classes need to be scheduled for 3 hours. We assign the rooms the same way we originally do. We then need to assign students. As we assign students, we check to make sure that we can assign each student into all 3 of the time slots for a class; we wouldn't want a student to only be able to make class 1 out of the 3 days! After assigning students, we are done!

Running this algorithm on the Haverford data gave a student preference value of 9,501, exactly 3 times as much as was standard, meaning it did no better. Further tests yield the same result; breaking up the time slots results in exactly triple the standard student preference value. While we didn't find a case where breaking up the time slots was better, we still believe one exists. However, modifying the algorithm in a way that only helps in some special case, a case that didn't show up across numerous tests, is really not a beneficial modification. On top of this fact, it is likely that some teachers and students would be less happy with a nontraditional schedule. This leads us to not recommend breaking up the time slots.

However, it certainly *seems* like this would lead to an improvement in student preferences. This leads us to believe that perhaps a very different implementation of this idea would lead to an improved schedule. We did not attempt these further modifications because this was already a fairly significant modification to the algorithm and in order for this modification to be successful, we believe that we would need to do a near complete overhaul of our original algorithm. Thus, if utilizing the same basic premise as our algorithm, splitting up the time slots is not recommended, but with the disclaimer that there may be a way to split the time slots and make generate a better schedule.

5.5 Limiting Professors to Buildings

At Haverford, most professors have offices in or near certain buildings, and usually certain subjects are taught exclusively in one set of classrooms. Though it seems that the registrar currently schedules some courses in random buildings or spreads subjects out across a few buildings, we wanted to extend our algorithm to see if limiting courses to be taught in these certain sets of classrooms, which we call buildings, would change the resulting student preferences value.

The original algorithm assigns courses to rooms based on room size and course popularity, where we pair the courses in each time slot in descending order of popularity with the rooms in descending order of size. The extension algorithm takes into account the buildings from the Haverford data and generates two new dictionaries to use in assigning courses to rooms: room_buildings and prof_buildings. The room_buildings dictionary maps all buildings to the list of rooms within them, and the prof_buildings dictionary maps professors to the buildings they can teach in. In the data, professors can teach in multiple different buildings, but in an effort to simplify our extension we chose to limit each professor to only one building. We modified the get_haverford_info.py script to get building and room information, included that new data in our constraints file, and accounted for the new data in the parser, parse_inputs.py.

Instead of assigning rooms from a dictionary including all rooms, now we assign from room_buildings. Like before, we iterate through each course in each time slot. For each course, we find the teacher for that course and look up the building that the teacher can teach in from prof_buildings. Then we find the rooms list for that building in room_buildings, which will be sorted by descending room size like before, and put the course in that first room in the time slot. We keep track of which rooms have been used with a pointer, so we move this pointer forwards in that particular building and continue with the next course. If all of the rooms in a certain building have been used up for that time slot, we cancel the class. In other words, we do not add it to the dictionary of classes that we use for filling students, and therefore it is not in the schedule.

We predicted a decrease in the student preferences value for this extension because we are limiting a constraint, so courses can be removed from the schedule, so students would not be able to enroll in these courses. The experiment confirmed our hypothesis, with a student preference value of 2697, which is 470 less than the previous 3167. Therefore, we would recommend that the college not restrain the classes to buildings at all, so any subject can be taught in any building. To extend this algorithm further, it would be interesting to examine the results if instead of removing the class we placed it in a random room out of the leftover rooms, which seems similar to Haverford's current strategy. Another extension could be to account for professors being able to teach in multiple buildings on campus.

Collaboration

Dylan Emery, Skyler Ellenburg, Noah Weinstein, Lake Giffen-Hunter