CS340 - Project

Sam Aronson and Ziyao Wang

September 17, 2020

1 Abstract

This paper proposes an algorithm to schedule courses in a way that optimizes the number of student enrollments, given student preferences, room, instructor, and timeslot constraints. While this problem is NP-Hard, and thus challenging to solve correctly, optimally, and quickly, we propose a much faster polynomial-time greedy algorithm to approximate the optimal solution, where validity is still guarenteed. Over simulated test data, our algorithm routinely can satisfy 90% of student preferences, in a number of simulated scenarios from ones containing 1000 students to those containing >50000 students. After adapting our algorithm to run on real Haverford course registrations from the spring of 2015, we found that exactly 70% of student preferences could be satisfied. This demonstrates the viability of our approach. We recommend that when scheduling courses in the future, the Registrar encourages consideration of highly-conflicted courses, even outside of the department in which the course is held.

2 Description

Our goal is to develop an algorithm that will try to build a class schedule for a college or university that maximizes the number of student enrollments, that is, the number of student preferences to take some set of courses that each student can actually attend, considering course constraints. While this problem has been proven to be NP-Hard, we propose a greedy algorithm that should satisfy most student constraints in most scenarios that runs in polynomial time. The algorithm that we developed is as follows:

First, determine all of the pairs of classes that want to be taken by the same students, and determine how many students want to take each combination of two classes. For courses that have the same instructor, set their conflict score to infinity. Then, starting with the most popular pair of classes and descending until all classes have been scheduled, schedule each class in a pseudo-optimal timeslot (that is, where the largest number of students can be scheduled into that timeslot). To determine the number of students that can take a

class in a given timeslot, find the number of students that don't have a conflict, and find the smallest available room that can fit the entire class (if that doesn't exist, only schedule students in the largest available room). Once all classes are scheduled (or attempted to be), we will have a generated schedule with each class having an associated time and room.

From there, take every student preference and schedule them into each requested class so long as it does not double-schedule the student or over-enroll the class capacity. Then return the enrollment of each class, and with it its room, time, and instructor.

3 Pseudocode

Our code will first read in the following parameters from the printed input , which is then used by our scheduling algorithm:

$Capacities$: a sorted list of tuples (room number, capacity) that stores the capacity of each room
$teacher\_class$: a list where the index is class number and the value represents the teacher who teaches the class
$timeslot\_num$, $room\_num$, $class\_num$, $teacher\_num$, $student\_num$: the number of timeslots, rooms, classes, teachers, and students
$student\_prefs$: a list of list that contains student preferences

Function `findOptimalSchedule`(*capacities, teacher_class, timeslot_num, room_num, class_num, teacher_num, student_num, student_prefs*)

Create an empty list $classPopularity$ and a 2D array $classConflicts$ that stores both student and teacher conflicts
for *each student's preference* $p \in$ *student_prefs* **do**
  for *each class* $c \in p$ **do**
    Increment $classPopularity$ by 1
  end
  for *each pair of classes (i, j)* $\in p$ **do**
    Increment both entries (i, j) and (j, i) in $classConflicts$ by 1
  end
end
for *each teacher* t **do**
  Get the two classes i, j that teacher t teaches
  Set both entries (i, j) and (j, i) in $classConflicts$ to infinity
end
Create an empty list $conflictList$ of tuples $(class1, class2, conflictscore)$
for *each unique pair of classes (i, j)* $\in$ *classConflicts* **do**
  Get $conflict\_score$ at entry (i, j)
  Append $(i, j, conflict\_score)$ to $conflictList$
end
Sort $conflictList$ by $conflict_score$
for *conflict* $\in$ *conflictList* **do**
  for *each class* $c \in$ *conflict* **do**
    if *c is not scheduled* **then**
      Create a empty list *scores* to store the conflict scores of each timeslot for c
      for *each timeslot* t **do**
        Calculate $roomRestriction$, the difference between popularity of c and capacity of the largest available room at t
        Calculate the $conflictScore$, the sum of conflictScores with a course in $enrollment[t]$
        Compare $roomRestriction$ and the current conflict score
        Increment conflict score at t by the larger of the two values
      end
      Assign $class_time[c]$ to the timeslot with minimum conflict score from *scores*
      Assign $class_room[t]$ to the largest room that can hold all students who want to take c or the largest room available at t
    end
  end
end
Create a list of classes *enrollment* that contains empty lists of students
for *each student's preference* $p \in$ *student_prefs* **do**
  for *each class* $c \in p$ **do**
    if *the class current enrollment is smaller than its capacity and the student is available at the timeslot for the class* **then**
      Add student to the list of students for c in *enrollment*
    end
  end
end

4 Time Analysis

First, we will define the following values:

- The number of student preferences s
- The number of classes c
- the number of rooms some constant r
- the number of timeslots some constant t
- the number of teachers $c/2$

For the sake of this time analysis, we will assume that the number of timeslots is some small constant, that cannot be reasonably grown as an input size. This is because we assume there is some reasonable cap in the number of independent timeslots that can be scheduled in a week. It seems that most schools will have fewer than twenty in most cases.

Notice that while we assume the number of teachers is $c/2$ by construction, this assumption can easily be loosened. In the case of a small liberal arts college like Bryn Mawr or Haverford, the number of teachers should never exceed the number of classes being taught. Even for larger institutions, the instructor constraint should not be orders of magnitude larger than the number of courses, so bounding it to some order of c should be appropriate in this case as well.

In the first *For* loop, each iteration consists of getting a list of possible pairs of classes from a student's preference and updating the corresponding values in a 2-dimensional array. Since each student's preference list is only visited once, there can be at most s iterations of the *For* loop. In the second *For* loop, each iteration consists of getting the currently most popular class and scheduling a timeslot for the class. Because each class can only be scheduled once, there can be at most $c \times t \times r = c^2$ iterations. Sorting L using quick sort consists of sorting $c^2 - c$ elements, which is thus bounded by $O(c^2 \log c)$. When scheduling the actual classes, we will do so for each class, meaning that the "scheduling" loop will occur c times. Inside, we will show that each iteration of the loop runs in $O(c)$ time, meaning that the overall time complexity of this loop is $O(c^2) < O(c^2 \log c)$. Finally, the outputted schedule is used to again sort students into classes. It will be shown that this is an $O(s)$ operation as well.

In order to run the algorithm in $O(s + c^2 \log c)$, the following operations will occur in $O(1)$. Operations marked with an asterisk(*) will occur in $O(c)$:

1. Getting a list of possible pairs of classes in a student's preference
2. Incrementing a class's popularity

3. Finding a pair of classes and increment its conflict score
4. Checking if a class is already scheduled
5. Determining if a teacher is available given a timeslot*
6. Finding a student who wants to take a class and don't have a conflict*
7. Finding the smallest room with capacity larger than the number of students who want to take the class and don't have a conflict*
8. Finding the largest available room*

Afterwards, the schedule will have been generated- that is classes will have assigned rooms and times. Then, the student preferences will be iterated once more and the following operations will occur. Each of these operations can happen in constant time, and will occur $4s$ times, leading to an overall time complexity of $O(s)$.

9. Each preference will be checked to see if the student is still available during the time of that class
10. Each preference will be checked to see if there is space in that room
11. The student's ID will be added to the enrollment for that class

Because each student's preferences can be bounded by $4s$, and these operations are constant, this part of the program can be achieved in $O(s)$ time as well.

Overall, the time complexity of the algorithm is $O(c^2 \log c + s)$.

4.1 Data Structures

The list of all possible pairs of classes can be stored in an 2D array, where the entry (i, j) represents the pair of classes i and j, and the value is the number of times the pair appears in all students' preferences. This is a matrix representation of the conflict graph. This allows us to increment conflict scores (3) and eventual determine if a teacher is availible in a current timeslot (5) in $O(1)$. Additionally, to find the number of students who want to take a class but don't have a conflict, any already scheduled class will need to be checked for double-scheduled students. There will be a maximum of r other classes, so there will be a maximum of r lookups. We can check the number of students who want to take any conflicting class with this data structure, with constant access time of $O(1)$. Notice that in all cases $r < c$, so to simplify our time complexity, we can consider this to happen in $O(c)$.
When the student's preferences are iterated, they will increment some indexed element in the 2D conflict array. This allows (2) to be completed in $O(1)$ time.
The list of classes that are scheduled can be stored in a matrix *classScheduled*, where the

key is class and the value is $True/False$. Initially, all values are $False$. When a class is scheduled, we set its value to $True$. This allows us to check whether a course is scheduled (4) in $O(1)$. Building such a list takes $O(1)$.

The list of rooms with their capacity is stored in a sorted array of tuples $Capacity$, where the first element is the room ID and the value is the room's capacity. This allows (7), (8), and (10) to be finished in $O(1)$. Building and sorting such a dictionary takes $O(r \log r) < O(c \log c)$.

Getting a list of possible pairs of classes in a student's preference takes $\binom{4}{2} = 6$ iterations because each student has 4 classes on their preference list. This means (1) can be done in $O(1)$.

In the end, enrollment will be stored as an array of Python lists. That means that the enrollment of any class can be checked in constant time, and so can adding the student ID to the list, helping accomplish (10) and (11).

When each student's classes are being scheduled, they will maintain an array for each timeslot that, when a course is added, will be flagged to false for that timeslot. Access and modification of this timeslot occurs in $O(1)$, allowing (9) to happen in constant time.

5 Discussion

This algorithm is a greedy algorithm, as once a course is scheduled it will never be revisited. Now some of the algorithmic choices to make it near-optimal will be discussed.

To optimize this algorithm's results, the goal is to maximize the number of students who can take their requested classes. There are two ways for a student to be unable to register for a requested class: they have another class that overlaps the timeslot and the room size is not large enough to fit all students. To try to optimize these conflicts, we make sure to prioritize scheduling classes that have the largest number of shared registered students, trying to minimize the number of overlapping classes that students want to cross-register for.

Next, when a class is being scheduled, we try to optimize to ensure that room sizes are large enough. First, the smallest available classroom that can fit all students will be attempted to be found to economize space without sacrificing student enrollment. If such a room does not exist, the largest possible room will be taken, again to trying to maximize the number of students scheduled for the class. Then, a timeslot that can schedule the largest number of people will be chosen for that specific course.

While this by no means guarantees the ideal schedule, this algorithm does prioritize the two main ways of optimizing a schedule to maximize student placement, resulting in a

near-optimal schedule in many cases.

A challenge while implementing the algorithm had to do with the off-by-one errors incurred from storing associated data in an array (that starts at 0) and associating it with ID numbers that start at 1. In the future, this can be mitigated by introducing objects that store both an indexed ID that starts at 0, and the "external-facing" identifier (this will also allow strings to be used as those "external" identifiers), or by simply allocating an additional memory slot and leaving the 0th index blank.

Another mitigation strategy could have been using more advanced software engineering approaches, like developing classes to manage and represent these values or using type checking. In retrospect, this would have avoided a lot of pain.

6 Experimental Analysis

6.1 Time Analysis

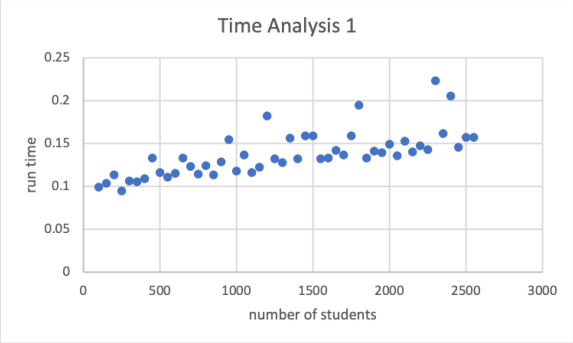

Figure 1: $O(s)$

The parameters that affect the run time the most is the number of students, s, and the number of classes, c. In Figure 1, the x-axis represents the number of students and the y-axis run time. The time complexity of the algorithm in terms of the number of students is $O(s)$. We can see in the Figure 1 that, as the number of students increases, the run time of the algorithm grows linearly. In this case, all other variables (number of courses, times and rooms) are held constant. In Figure 2, the x-axis is the number of classes and the y-axis is run time. This time, we held the number of students, rooms and times constant and increased the number of courses. The time complexity of the algorithm in terms of the number of classes is $O(c^2 \log c)$, as confirmed by this polynomial graph of a constant factor of $O(c^2 \log c)$.

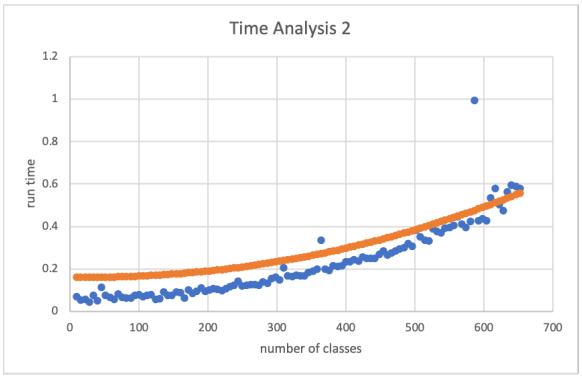

Figure 2: $O(c^2 logc)$

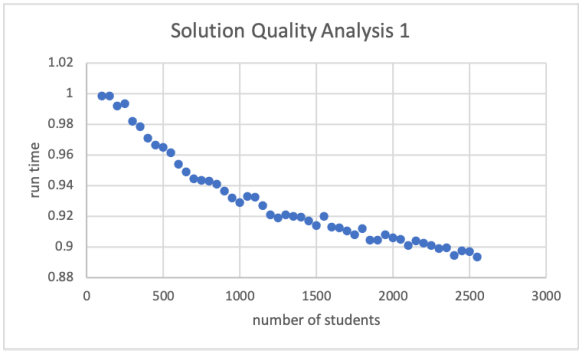

Figure 3: keeping the number of classes constant

6.2 Solution Quality Quality

In Figure 3, we keep the number of classes constant. As the number of students increases, we achieve a lower bound of around 90 percent. Similarly, in Figure 4, if we keep the number of students constant, as the number of classes increases, we still have a lower bound of about 85 percent. This is a very exciting finding. Digging in to the data, it seems that in most cases it is room capacities, and not schedule conflicts that are causing students to be kicked out of classes. This suggests that it could be room sizes that constrain this version of the algorithm. In real-world findings (discussed later), room capacity constraints do not seem to be the primary constraint of student scores.

7 Proof of Validity

While our algorithm is not guaranteed to provide an optimal solution, it is guaranteed to produce a valid schedule. First, we will prove a number of characteristics about our

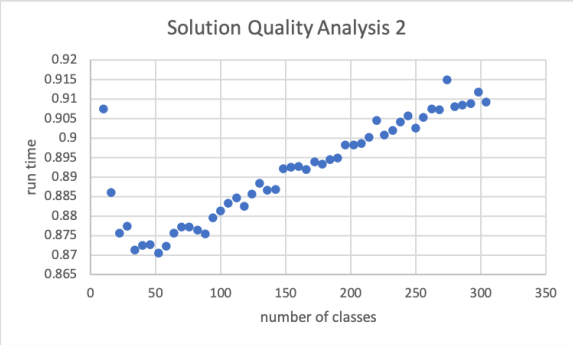

Figure 4: keeping the number of students constant

algorithm, and then prove why this implies validity.

Lemma 1. *No classroom will ever be overbooked*

Proof. For contradiction, suppose that a classroom is overbooked, and that course c_0 and course c_1 are scheduled in the same room r at the same time t, and that c_0 was booked before c_1. This would imply that when c_1 was booked, r was unscheduled at t. In turn, this implies that when c_0 was booked, that it was not added to the schedule. But because c_0 was scheduled, there is a contradiction and the classroom will never be overbooked. □

Lemma 2. *No teacher will ever be overbooked.*

Proof. For contradiction, suppose that a teacher is scheduled to teach class c_0 and c_1 at time t. Without loss of generality, assume c_0 was scheduled before c_1. In the conflict matrix, the pair of classes (c_0, c_1) have a conflict score of infinity, along with all other pairs of classes that are taught by the same instructor. Assuming there exist at least two timeslots, these classes being scheduled in the same timeslot implies that some finite cost is larger than the infinite cost of pairint c_0 and c_1 in the same timeslot. Therefore, this is a contradiction and c_0 will never be scheduled in the same timeslot as c_1. □

Lemma 3. *Classrooms will not exceed capacity*

Proof. Suppose, for contradiction that class c has more students registered than can fit in the room. However, the value s is defined to be the minimum of the number of students that want to take the class that don't have a pre-existing conflict and the room capacity for the candidate scheduled room. Therefore, class c will never schedule more students in a room than its room capacity, leading to a contradiction. Therefore no classroom will ever exceed its capacity. □

Lemma 4. *No class is scheduled twice.*

Proof. Suppose the class c is scheduled in two different timeslots. For a class to be scheduled, it must be the case that scheduled[c] is false. But once the class is scheduled the first time, the flag scheduled[c] is set to true. Therefore, it must not be the case that any class c is scheduled in two timeslots. □

Proof. To prove that this algorithm returns a valid schedule, the following criteria must be met:

- No classroom will ever be overbooked at time t.
- No teacher will ever be scheduled to have two simultaneous classes at time t,
- No classroom will exceed its capacity, that is, no more students will ben enrolled in a course than can fit into the room in which the course is scheduled, and
- No class will ever be scheduled twice.

By Lemmas 1, 2, 3, and 4, all of these criteria will be met. Therefore, the outputted schedule by the algorithm will always be valid. □

8 Extension to Real-World Data

One major goal of this algorithm-building project is to apply it to real-world data. In this case, we targeted Haverford's Spring 2014 semester enrollment data. We will describe certain assumptions that we made that were relaxed, such that a schedule can be generated that can help us make some conclusions about how the registrar should schedule courses. Descriptions of the assumptions and extensions that must be relaxed will now be explained. Excitingly, even after relaxing these assumptions our code ran near instantaniously.

8.1 Assumption Relaxations

There are two main ways in which our assumptions about the data needed to be relaxed: first, we made certain assumptions about the data that were incorrect. Additionally, there were some assumptions about the structure of the schedule that did not exist. These were as follows:

8.1.1 Course Identifiers

We assumed that course identifiers would be positive, sequential integers. This posed a major problem as data was stored for courses in terms of its index in an array. It was also assumed that data entries would be sequential (that is, each memory space in the array would be filled with some course entry) This was not the case. To combat this, as we

were reading in the courses initially, we assigned each course to a "memory ID," that is a sequential identifier that could be mapped to a memory slot in an array. These memory IDs were indices stored in a python list. This allowed us to continue with our approach through the algorithm part of the portion while only making slight modifications when reading and writing the data from files.

One major issue in this approach was that a number of "off by one" errors sprung up again, because of assumptions of adjustments that snuck into the code at multiple points, further underlining the importance of better organizational methods.

8.1.2 Room Identifiers

Like course identifiers, we assumed that room identifiers would be sequential integers. They were strings. We used a similar approach to the course identifiers, by saving a mapping from string to sequential integer in a python List. Just like with course identifiers, this allowed the algorithm to run nearly unmodified, while just modifying when data was being read and written.

8.1.3 Exclusion of Bryn Mawr Classes

When working through the data, there were some courses that did not have instructors assigned to them. Combing through the original data, it appears that those were actually Bryn Mawr courses, that were occasionally included in the Haverford registration data. Our algorithm ignores these courses, as they should not be registered by the Haverford registrar. By not scheduling these courses, this most likely impacted our student preference score, but it appears that there were only two instances of courses that were included in this way, so the impact is most likely negligible. In these cases, these courses were manually excluded from the constraints file because there were so few instances of them.

8.1.4 Student Course Credits

We found that students may pre-register for up to ten "courses" in the course registrations. This is for two main reasons: first, if a student is required to enroll in a lab or discussion section, that will count as a second registration (with a duplicate course number). Next, some courses award only a half-credit (like quarter-long courses, or independent studies or the like). Therefore, students may register for significantly more (or in some cases fewer) courses than the four that is expected in the basic simulations. To combat this, we did two main things: first, we programmatically removed duplicates in every student preference as it is being read in. Next, we also relaxed the assumption that students will take four courses, by modifying our loops that generate the conflict graph. In fact, we found students taking any number from 1-10 courses by this count.

Notice that in this case, our algorithm still ignores certain aspects of course scheduling, like

the scheduling of labs or discussion sections. This is something that in the future should be added to our algorithm, as it adds complexity that our current algorithm ignores. It is also important to notice that in many cases, time and room scheduling can be somewhat political (as there are less desirable rooms in which few established full tenured faculty teach).

8.1.5 Instructor Courses

We assumed in the basic version of the data that instructors only taught 2 courses. We found that instructors often teach more or fewer courses, and similarly loosened our assumptions about the number of courses instructors teach by modifying our loops that read in courses.

Notice that in the actual scheduling process, there are many more considerations that have to be made with respect to instructors. Some (like adjunct professors) may only be on campus two days a week, or have other scheduling conflicts that they work into their course schedule. All of these preferences are ignored, but should not be when actual scheduling occurs.

8.1.6 Differing timeslots imply availability

One major assumption that had to be overcome was with respect to timeslots. Normally, a number of courses would be scheduled during overlapping timeslots (that is, a course might be offered from 9:00AM-10:30AM Monday and Wednesday, and then a different course might be offered 9:00AM-10:00AM Monday, Wednesday and Friday). We assumed that differing timeslots implied that a student or instructor could attend a course in both. This is not the case. To mitigate this affect, we estimate that there are around 18 independent timeslots during the week that could be scheduled, and revised the constraints document to reflect that before running our scheduling algorithm.

Notice that this is not a perfect solution. It is possible, for example, to consider three classes: class A scheduled on Tuesday and Thursday from 2:30-4:00PM, class B scheduled on Tuesday from 1:00-4:00PM, and class C scheduled on Thursday from 1:00-4:00PM. In this case, a student could take courses B and C, but not A and B or A and C. Therefore, a more complex relationship between timeslots should be developed. With our structure, we should be able to do so without significantly impacting our time complexity.

8.2 Experimental Results

We found that our algorithm, when run on the Haverford data, satisfied 3498 student preferences. Data analysis suggests that in this semester, there were 5031 student course requests. Therefore, we satisfied 70% of student requests. While this number is significantly lower than our generated test cases, this is to be expected as real-world data looks rather

different in many ways compared to our randomly-generated sets. This is most likely because the topology of our conflict graph is different- in this case, the verticies (courses) and edges (conflicts) most likely follow more particular patterns that could be used to generate a better-fitting algorithm.

We also ran a number of experiments to try to determine what constraint led to this score. First, we tried restricting the number of rooms. We noticed in the first running of the algorithm that the room choices that were made were rather strange, and in many cases inappropriate. We found by cutting in half the number of rooms, that our accuracy rate of 70% remained unaffected. This is encouraging, and at first seemed to show that physical classrooms did not offer significant constraints to course schedules, which is in line with what is expected

Next we tried to see if restricting the number of timeslots affected our accuracy rate. We restricted the number of timeslots from 18 to 10. In this case, our accuracy rate dipped slightly to 69%, but quite honestly we were expecting more of a difference.

Next we tried to expand the number of timeslots such that each course could be assigned its own (260 timeslots). We would expect student satsfaction to be 100%, because at each point a new timeslot could be chosen for each course. Yet, we recieved the same exact accuracy rate as our first pass, of 3498/5031. This is certainly suspicious and suggests a bug in the adaptation to the Haverford implementation. Encouragingly, no invalid schedules were ever produced.

Looking forward, we would expect for there to be little pressure on room assignments, but our approach seems to indicate that looking at course conflicts is a fruitful path forward. We would like to run an experiment where a course was never scheduled with the three (or k) most popular courses that are taken with it, and to instruct each instructor to avoid scheduling these courses in such a way that would conflict with the k most popular cross-courses. Our graph suggests that the department in which these shared courses is not relevant, except as a heuristic to approximate the most popular courses. We would expect an instruction like this to improve student satisfaction in an implementable way for the registrar when scheduling for the next term.

9 Conclusion

Overall, the algorithm proposed in this paper returns a schedule that satisfies 90% of student preferences on simulated data and 70% on real-world data. By using data structures such as a conflict matrix, the algorithm runs almost instantaneously on data sets of the size of a small liberal arts college. We also give recommendations on the assumptions and extensions that must be relaxed: course identifiers, room identifiers, exclusion of Bryn Mawr classes, the number of student course credits, and the number of courses each instructor

teaches. This also validates the process of considering this problem with a graph-based approach. In spite of the restrictions, our algorithm is valid and produce a viable schedule that satisfies most student preferences.