CS340 - The Registrar's Problem

Audrey Yang, Jac McCarty, Juno Bartsch

May 31, 2023

1 Abstract

The algorithm we designed is popularity-based with a conflict scheduling improvement. This means that our algorithm will schedule the classes desired by the most students into the largest rooms. The conflict adjustment works by comparing scheduled classes and swapping them if it would increase the net number of students able to enroll. From our original algorithm, we implemented several extensions to make our findings more applicable to real-world data. We prioritized students majoring in the class's subject and students with higher seniority when filling classes (this data we generated ourselves as it is not provided). We also implemented Emily Balch/Writing seminars (Esems) during a specific time slot, thereby reducing conflicts with other courses. In addition, classes and rooms were assigned domains so that classes are placed in a room of a matching domain. Finally, rooms and students were assigned an access value that denoted if the room was accessible and if the student required an accessible classroom respectively. Classes requested by students who need accessible classrooms were placed only in accessible classrooms.

Based on these extensions, we make the following recommendations: give writing seminars their own time slot(s); avoid overlap in time slots that classes can be scheduled, create more or larger sections of popular courses, allow students to request accessible classrooms; and, maintain an accurate list of accessible classrooms across the BiCo. Having writing seminars as a distinct time slot (where no other courses with first-years can be scheduled) reduces time conflicts for first years and the need for students to swap writing seminars. This would relieve some of the burden on the registrar when manually re-assigning writing seminars. In addition, there were time slot conflicts with with around 60-80 identified time slots depending on the semester. Reducing this number would reduce overlap between courses and increase students' ability to take courses. We also determined that adding more sections of courses when handling the real-world data dramatically increased the fit, which is why we recommend adding additional or larger sections to popular courses whenever possible. Finally, we believe that centralized lists of accessible classrooms and students needing those accessible classrooms should be created. This will reduce the burden on both Access and Disability Services and the registrar in ensuring students are placed in the Recommendations section. Ultimately, these suggestions are designed to anticipate restrictions and conflicts in the schedule and mitigate them. These changes will reduce time conflicts and ensure that disabled students' learning is not restricted by the physical limitations of buildings.

2 Description - Base Algorithm

Initially, all classes $c = |C|$, time slots $t = |T|$, rooms $r = |R|$, professors $p = |P| = 0.5|C|$, and students $s = |S|$ are available. Assume that each class c_i has a professor p_i to teach it, where each professor can teach at most two courses. Also assume that $c > r$, but $c \leq rt$.

In the extended algorithm, classes, rooms, and students each have additional data. Assume each class has a domain d, a major m, and an Esem boolean e. Assume each room has a domain d and an access value a (i.e., is the room accessible or not). Assume each student has a major m, a class year y, and an access value a (i.e., does the student have accessibility needs). The number of domains and class years are bounded by constants 3 and 4, respectively (i.e., # of domains = 3, # of class years = 4). The number of Esems is bounded by $0.1|C|$. The number of majors is inputted by the user and is bounded by $|C|$. Access values are specific to each room and student.

Organize classes in descending student preference, (i.e., the class that the most students have on their preference list is ranked first).

Organize the rooms in decreasing size order. Let a "time conflict" be defined as when a room or professor is scheduled for two or more classes during one time slot.

Take the largest room that has yet to have all its time slots filled. Then, take the earliest time slot available for that room and pick the most preferred unassigned class c_i.

Attempt to pair c_i with its professor p_i. If p_i cannot be paired with c_i (i.e., the professor is already teaching another class in this time slot), set this course aside. Check the next most preferred class c_{i+1} and repeat this process until an available course that can have its professor assigned to it is found, and assign that class-professor pair to this room and time slot. Add all classes that were skipped back to the front of the list of classes. In essence, this will effectively skip all classes that have a time conflict for this time slot.

Once a class has been assigned a professor, time slot, and room, fill the course with students who prefer that class and do not have a time conflict for the chosen time slot until either there is no capacity in the room or all students who selected that course have been assigned. For every student who is assigned, increase the global student count by one. The global student count measures how many preferred classes the students end up enrolled in, and the overall fit of the algorithm can be determined by dividing the global student count by 4s.

Continue the above process with each room, time, and class until all rooms have had all of their time slots assigned a class and a professor or until there are no classes left to assign, and the selected class for that room has as many students as possible assigned to it. Output the assigned course lists as well as the student preference count.

Then, once each room's "schedule" of classes at times is generated, attempt to optimize that room's schedule by for each class, swapping its time slot with another class's (also scheduled within the room). If total enrollment increases with a swap, make the swap permanent.

3 Pseudocode

```
Function optimizeRoomSchedule(majschedule, whoPrefers, studentSchedules, profSchedules)
    for each time $t1$ in room-schedule do
        t1-class = room-schedule[$t1$]
        max-t1-class-enrolled = t1-class.enrolled-students, max-t2-class-enrolled =
          t2-class.enrolled-students, max-t2 = -1
        for each time $t2$ from 1 to ($t1$ - 1) in room-schedule do
            t2-class = room-schedule[$t2$]
            if t1-class.professor has a conflict for $t2$ OR if t2-class.professor has a conflict for $t1$ then
                continue
            end
            t1-class-enrolled = array of all students in whoPrefers[t1-class] who do not have a time conflict
              with t2 (stop adding students when length(t1-enrolled) == room.capacity)
            t2-class-enrolled = array of all students in whoPrefers[t2-class] who do not have a time conflict
              with t1 (stop adding students when length(t2-enrolled) == room.capacity)
            if t1-class-enrolled + t2-class-enrolled > max-t1-class-enrolled + max-t2-class-enrolled then
                max-t2 = t2
            end
        end
        if max-t2 $\neq$ -1 then
            t1-class.time = $t2$, t2-class.time = $t1$
            t1-class.enrolled-students = max-t1-class-enrolled, t2-class.enrolled-students =
              max-t2-class-enrolled
            room-schedule[$t2$] = t1-class
            room-schedule[$t1$] = t2-class
        end
    end
```

```
Function classSchedule(set of time slots T, set of room sizes R, set of classes C, set of professors P, set
  of student preference lists S)
    For any class, let preferredStudents[class] be the set of students with class on their preference lists
    Let classRankPriorityList be a set of all classes organized by how many students have the class on their
      preference lists in descending order
    For any class, let classProfessor[class] be the professor capable of teaching class
    For all rooms, let maxRoomSizeList be the set of all rooms organized by descending room size
    $globalStudentCount = 0$
    $Schedule$ = r x t Matrix, indexed by rooms $r$ and time slots $t$
    for each professor $p_i$ in $P$ do
        Let $p_i$.schedule be all the time slots for which $p_i$ is scheduled for a class
        Set $p_i$.schedule = []
    end
    for each student $s_i$ in $S$ do
        Let $s_i$.schedule be the time slots for which $s_i$ is scheduled for a class (i.e., attending a class)
        Set $s_i$.schedule = []
    end
    for each room in maxRoomSize do
        for each time in $T$ do
            if classRankPriorityList is empty then
                break out of both loops
            end
            $class$ = classRankPriorityList.extractFront
            while classProfessor[class] has a conflict for ts do
                append $class$ to holdClass
                class = classRankPriorityList.extractFront
            end
            Prepend holdClass to classRankPriorityList
            $professor$ = classProfessor[$class$]
            add $ts$ to $professor$.schedule (i.e., professor cannot teach another class at $ts$, it will be a
              conflict)
            Let $students$ be a set of students enrolled in $class$
            while size(students) < room.size and preferredStudents is not empty do
                $x$ = preferredStudents[$class$].extractFront
                if $x$ does not have a time conflict at ts then
                    append $x$ to $students$
                    increase $globalStudentCount$ by one
                end
            end
            $Schedule[room][ts] = class$
        end
        for each room-schedule from schedule[1] to schedule[room] do
            optimizeRoomSchedule(room-schedule, whoPrefers, set of all student schedules, set of all
              professor schedules)
        end
    end
    $studentPreferenceCount = globalStudentCount/4s$
    return $Schedule$, $studentPreferenceCount$
```

4 Time Analysis - Base Algorithm

Recall that $|T| = t$, $|C| = c$, $|P| = p$, $|R| = r$, and $|S| = s$. Also recall that $p = c/2$, t is a small constant, and $c \leq rt$.

Sorting all classes beforehand will take $O(c \log c)$ time, and adding all classes to classRankPriorityList will take $O(c)$ time.

Initializing maxRoomSize involves adding r rooms to an array, for a total complexity of $O(r)$. Sorting this array in order of descending room size, which can be accomplished in $O(r \log r)$ time.

Thus, the *for each professor* p_i loop and *for each student* s_i loop can each be accomplished in $O(p) = O(0.5c)$ and $O(s)$ time respectively.

Consider the nested loops *for each room* and *for each time slot*. The first loop iterates through all rooms, and the second iterates through all time slots. Because $c \leq rt$ and both loops will be broken once classRankPriorityList is empty, the number of iterations for this nested loop set is not $O(rt)$, but rather $O(c)$. Each iteration sees exactly one class removed from the classRankPriorityList. Though classes are re-appended to the queue as time conflicts are encountered, the number of classes extracted from the queue is always one more than the number re-appended, which allows the classRankPriorityList to steadily decrement in size with each passing iteration. Thus, the nested for loops terminates after $O(c)$ iterations.

Inside the for loops, classRankPriorityList.extractFront can be completed in $O(1)$ time.

The first of the inner while loops - *while classProfessor[class] has a conflict* - can, at a maximum, iterate exactly $r - 1$ times. As there are only r possible rooms in which to teach, there can be at most r professors teaching at any one time. Whenever this loop is called, at least one of those rooms will be unassigned during this time slot. Thus, this loop will terminate after $O(r)$ iterations. Moreover, because classRankPriorityList is a sorted linked list composed of all classes, extracting the most desired class will take $O(1)$ time.

Likewise, at maximum, only r classes can be held at any one time - one for each room. Therefore, *holdClass* can be a maximum size of $r-1$, since the current room will be still unassigned when items are being appended to and removed from *holdClass*. Hence, there can be exactly $O(r)$ removals from *holdClass* and appends to *classRankPriorityList*. Since appending items and extractMax for a sorted linked list is $O(1)$ each, adding r items to *holdClass* from *classRankPriorityList* is an $O(r)$ operation.

The second inner while loop - *while* $size(students) < room.size$ - in a worst case scenario, would have s students in preferredStudents, with all of them having time conflicts for *time*. In this scenario, this while loop would terminate after a maximum of $O(s)$ iterations.

Next, consider the optimizeRoomSchedule() function. Within it is a nested pair of for loops, each one iterating t times. Determining whether or not the professors for $t1 - class$ and $t2 - class$ have a time conflict for $t2$ or $t1$ is an O(1) operation, as each of the professor's schedule can contain a maximum of two times they might be teaching at. However, since each whoPrefers[class] array might hold a maximum of s students, filling t1-class-enrolled and t2-class-enrolled is an $O(s)$ operation. Therefore, since all other operations are O(1), optimizeRoomSchedule() as a function is $O(t^2 s)$.

Thus, since for each iteration of the *for(each room in maxRoomSize)* loop, optimizeRoomSchedule is called for all room-schedules between 1 and *room* (which in itself is a function of r), optimize-RoomSchedule() is called $O(rt^2s)$ times for each iteration of an $O(r)$ loop.

Therefore, summing all of these complexities up, the time complexity of this algorithm is:

$$O(c \log c + c + r \log r + r + s + 0.5c + cs + r^2t^2s)$$

$$\Rightarrow O(c \log c + r \log r + s + 1.5c + cs)$$

$$\approx O(c \log c + r \log r + r + c + s + cs)$$

Because c cannot be larger than rt, r^2t^2s dominates over everything else, meaning the final time complexity is:

$$complexity \approx O(r^2t^2s)$$

In order for this algorithm to complete in $O(r^2t^2s)$ time, the following operations must be completed in $O(1)$ time:

1. retrieving professor from classProfessor[class]
2. determining whether professor has a conflict for a time slot
3. adding items to $holdClass$
4. adding a time to professor.schedule
5. extracting items from preferredStudents[class]
6. appending x to students
7. prepending $holdClass$ to $classRankPriorityList$
8. extracting the front from $classRankPriorityList$
9. choosing a room to assign a class to

4.0.1 Data Structures

All (class, professor) pairings can be stored in an array $classProfessor$, where $classProfessor[class] = professor$. This means that (1) can be accomplished in $O(1)$ time.

Each professor's schedule can be stored in an array of linked lists, where each professor's schedule is indexed by their professor ID number, and each linked list consists of all times for which the professor is teaching a class (i.e., $schedule[professor_i] = [time_a, time_b]$). Since each professor can teach a maximum of two classes, each linked list can be of maximum length 2. Thus, running schedule[professor].contains(time) would involve comparing *time* against only two other items in the list, accomplishing (2) in $O(1)$. Likewise, because finding the correct professor's schedule in an array is $O(1)$ and adding any item to a linked list is also $O(1)$, accomplishing (4) is an $O(1)$ operation.

For minimum appending and prepending overhead, $classRankPriorityList$ should be held in a linked list. However, this can provide an issue for sorting, as sorting this linked list, or adding in a sorted order, would have a complexity of $O(c^2)$. For this reason, we use two different data structures to initialize *classRankPriorityList*. Initially, we add all classes to a temporary array, which can be completed in $O(c)$ time. Likewise, if space issues are a concern, this array can be declared and initialized in a separate function to the main, so that the space used will be freed afterward. This array can then be mergesorted in $O(c \log c)$ time. Finally, all classes in the temporary array can then be appended, in order, to *classRankPriorityList* in $O(c)$, allowing for an initialization complexity of $O(c \log c + 2c) \approx O(c \log c)$ time - just as was described in the pseudocode. Finally, because *classRankPriorityList* is a linked list, (8) can be accomplished in $O(1)$ time.

holdClass should then also be a linked list. This will allow for (3) to be completed in $O(1)$ time. Likewise, because *classRankPriorityList* is also a linked list, (7) can be completed in $O(1)$ time as well.

$preferredStudents$ can be stored as an array of FIFO queues, where each *class* is indexed by a class ID number to a location in the array, and each *class*'s corresponding FIFO queue is made up of all students who put down *class* on their preference sheets. Finding the correct FIFO queue for *class* would be an $O(1)$ operation. Likewise extracting items from the FIFO queue would be an $O(1)$ operation, accomplishing (5) in $O(1)$ time. Similarly, $preferredStudents$ could be initialized by iterating through each student preference list, and then appending the student to the correct $preferredStudents[class]$ FIFO queue for each class on each student's preference list. Since each preference list has a maximum of only 4 classes, this can be done in $O(4s) \approx O(s) < O(rc\log(c)+cs)$ time.

For each queue in $preferredStudents$, sorting the students by class year and major will at most take $O(s)$ time.

The final list of students enrolled in each class can be held in a FIFO queue. This would allow for (6) in $O(1)$ time.

maxRoomSize is a linked list sorted in descending room size. This allows for (9) to be completed in $O(1)$ time.

4.1 Experimental Time Analysis

5 Proof of Correctness

Proof. To prove that this algorithm works, four parts need to be shown.

1. proof of termination
2. proof that no professor teaches more than one class for a single time slot
3. proof that no student is enrolled in two classes during the same time slot
4. proof that no room is scheduled for more than one class for a single time slot
5. proof that on completion of the algorithm, any unscheduled courses cannot be scheduled

5.0.1 Proof of Termination:

Pseudocode and time analysis sections show that this algorithm will terminate in $O(cs+r\log r)$ time. The first for loop (*for each professor*) will terminate in $p = 0.5c$ iterations, and the second for loop (*for each student*) in s iterations. Furthermore, the nested for loops *must* terminate after c iterations of the inner loop, as was shown in the time analysis section. The *while classProfessor[class] has a conflict* loop must terminate after at most $O(r)$ iterations, since there can be at most r professors teaching at any one time (one for each room). Finally, the *while size(students) < room.size and preferredStudents is not empty* loop must terminate in $O(s)$ iterations, as each iteration decreases the size of preferredStudents by one and the maximum size of preferredStudents is s.

5.0.2 Proof that no professor teaches more than one class for a single time slot:

Suppose some professor $prof$ teaches both class c_a and class c_b for some time slot $time_i$. This would imply that $prof$ had $time_i$ listed in $prof$.schedule, and then was assigned to c_b for time slot $time_i$ regardless. However, this is a contradiction, because as is stated in the pseudocode, if a professor has a time conflict, the algorithm will search through the classRankPriorityList until it finds the next available most preferred class with an available professor. At this point, all classes stashed in *holdClass* will be re-added to classRankPriorityList.

Furthermore, once the algorithm reaches the next time slot $time_{i+1}$, c_b will once again be the most preferred class in classRankPriorityQ. Recall now that each professor is capable of teaching two classes. Because $prof$ is already teaching c_a during $time_i$, and c_b has yet to be assigned a time slot at this point in the algorithm, there is no other possible class that could conflict with c_b at $time_{i+1}$.

Thus, no professor can teach more than one class for a particular time slot.

5.0.3 proof that no student is enrolled in two classes during the same time slot

Pseudocode shows that, in order for any student to be enrolled in a class during the greedy portion of our algorithm, they must not currently be enrolled in any classes at the same time. Thus, if a student was to be scheduled for two classes at the same time, the algorithm would need to enroll them in a class for which they had a time conflict in their schedule.

Similarly, in $optimizeRoomSchedule()$, in order to be added to $t1-class-enrolled$ or $t2-class-enrolled$ a student must not have a time conflict with the time their class would be swapping to.

Thus, when two classes swap times to increase the total student enrollment, no student will have swapped to a time already in their schedule.

Therefore, no student can be enrolled in more than one class for any particular time slot.

5.0.4 Proof that no room is scheduled for more than one class for a single time slot:

Let ts_r represent "a time slot for a room r." Pseudocode shows that for each room r, each time slot ts is iterated through, and for each ts_r, a class is assigned to it. Time slots never backtrack, and only one class is ever assigned during one iteration. Thus, no two classes can be scheduled for a single room for a single time slot. Both of these amount to a contradiction, so no room has two classes assigned to any single time slot.

5.0.5 Proof that on completion of the algorithm, any unscheduled courses cannot be scheduled (by contradiction)

Suppose that on completion of the algorithm, an arbitrary course c has been unscheduled and it could have been scheduled with an arbitrary room r and time slot t. For this case to have occurred, the algorithm must have not scheduled this course in that available slot. However, the algorithm works by adding an unscheduled class to the classRankPriorityQ and, once extracted, checking it against every succeeding room/time-slot combination until the algorithm finds a class/room/time-slot combination that does not have a conflict. This means that there is either a conflict for the course not to be added or a time slot was skipped, which is impossible by construction of the greedy algorithm. Both cases result in contradiction. $\square$

6 Description - Extensions

We implemented 5 extensions to make this algorithm sort classes more realistically.

1. During enrollment, prioritize students for a class if the class contributes to their major
2. During enrollment, prioritize students of a higher year
3. Certain classes can only be taught in certain categories of buildings - designated here as domains
4. Emily Balch Seminars - designated here as Esems - must not conflict with other classes. Likewise, all freshmen must be enrolled in an Esem.
5. Classes that enroll disabled students must be taught in accessible classrooms

6.1 Extensions 1 and 2

During enrollment, prioritize students for a class if the class contributes to their major, and prioritize students of a higher year.

This can be done by sorting the list of students who prefer each class in order of Senior Majors > Junior Majors > Sophomore Majors > Freshman Majors > Senior NonMajors > Junior NonMajors > Sophomore NonMajors > Freshman NonMajors.

6.2 Extension 3

Certain classes can only be taught in certain categories of buildings - designated here as domains.

This can be achieved by creating an array of sorted Classes lists, with each location indexed by the domain, where classes that are hosted in said location must also fall under the same domain. Then, for each (room, time) combination, a class will be retrieved from the sorted Classes list indexed by the room's domain.

6.3 Extension 4

Emily Balch Seminars - designated here as Esems - must not conflict with other classes. Likewise, all freshmen must be enrolled in an Esem.

This extension can be achieved by maintaining a separate array of sorted Class lists made up entirely of Esems, where each location within the array is once again indexed by domain. Esems can then be given priority in scheduling, where each Esem is scheduled for the same time slot - the esem time slot. No other courses may be scheduled for this time slot.

6.4 Extension 5

Classes that enroll disabled students must be taught in accessible classrooms.

This can be achieved by identifying every student that needs an accessible classroom. Then, you can identify each class that is preferred by a disabled student. Depending on the type of class, you can then subdivide the set of all classes into four groups, to enroll students in this order of priority: accessible esems, accessible non-esem classes, non-accessible esems, non-accessible non-esem classes.

Likewise, you can further subdivide the set of rooms into a list of accessible versus non-accessible rooms. When scheduling accessible classes, pick from the smallest accessible room that can fit all preferred students *UNLESS* the number of classrooms remaining $\leq$ the number of accessible classes needing to be scheduled.

7 Pseudocode - Extensions

```
Function miniSchedule(set of time slots T, set of room sizes R, set of classes C, set of professors P, set
  of student preference lists S, schedule, takenRoomTimeCombos, set of professor schedules, set of student
  schedules)
    Let each class have an attached domain variable, denoted class.domain, in which the class must be
      scheduled
    Let each room have an attached domain variable, denoted room.domain, which denotes the domain
      that room falls under
    for each room in R do
        for each timeslot in T do
            if (room, timeslot) exists in takenRoomTimeCombos then
                continue
            end
            clss = classes[room.domain].mostPreferable
            while clss's professor has a conflict for timeslot do
                move on to next most preferable clss in classes
            end
            prepend all skipped classes back into classes to retain sorted order
            enroll all students who prefer clss and do not have a time conflict for timeslot
            for each student enrolled in clss, append timeslot to their schedule
        end
        for each roomSchedule from schedule[1] to schedule[room] do
            optimizeRoomSchedule(roomSchedule, whoPrefers, set of all studentSchedules, set of all
              professor schedules)
        end
    end
```

```
Function accessibleSchedule(set of time slots T, set of room sizes R, set of classes C, set of esems E,
  set of professors P, set of student preference lists S, schedule)
    for each accessible room, sorted from least to greatest do
        takenRoomTimeCombos = []
        for each esem in accessibleEsems do
            if number of people who prefer esem can fit in room, schedule esem for room and enroll all
              students
            if number of people who prefer esem can't fit in room, then either continue on to next room, if
              there are enough rooms left for the number of accessibleEsems remaining, or schedule all
              students possible in room
            for each student enrolled in class, append time to that student's schedule
            schedule[room][1] = esem
            takenRoomTimeCombos.append((room, 1))
        end
        for each time in T (not including time 1) do
            for each class in accessibleClasses do
                if number of people who prefer class can fit in room and professor has no time conflicts,
                  schedule class for room and enroll all students
                if the number of people who prefer class can't fit in room, and if there are enough (room,
                  time) combinations left for remaining accessibleClasses, then move onto next room. If
                  not, schedule all students possible in room
                if professor has a conflict, skip over all classes with a professor conflict, then re-add all
                  skipped classes to accessibleClasses, and either use the current room if number of people
                  who prefer class can fit in room, or find a new room if there are enough rooms left for the
                  remaining accessible classes
                for each student enrolled in class, append time to that student's schedule
                schedule[room][time] = class
                takenRoomTimeCombos.append((room, time))
            end
        end
    end
```

```
Function extended(set of time slots T, set of room sizes R, set of classes C, set of professors P, set of
  student preference lists S)
    for each student's preference list in S do
        if student is disabled and prefers class, class must be accessible ⇒ remove from C, add to
          accessibleClasses[class.domain]
        UNLESS student is disabled and class is an Esem, in which case remove from C and add to
          accessibleEsems[class.domain]
    end
    for each class in C do
        Let whoPrefers[class] be a list of all students who prefer class, organized in order of student
          priority as was specified in extensions 1 and 2
    end
    Let classes, esems, accessibleClasses, accessibleEsems each be arrays of LinkedLists, where each
      LinkedList of classes is indexed by the domain those classes can be found in, and where each
      LinkedList is ordered from least to most preferential
    sort each accessible classroom from least to greatest capacity in accessibleRooms
    for each domain do
        sort accessibleEsems[domain] and accessibleClasses[domain] from most to least preferential
        sort all other nonaccessible classes[domain] from most to least preferential
        sort all other nonaccessible esems[domain] from most to least preferential
    end
    sort all rooms (including accessible rooms) from greatest to least capacity in rooms
    Let schedule be a 2D array, where each location schedule[room][time] is equal to the class scheduled in
      that room during that time slot
    takenRoomTimeCombos = accessibleSchedule(T, accessibleRooms, accessibleEsems, accessibleClasses,
      P, S, schedule)
    miniSchedule([1], rooms, esems, P, S, schedule, takenRoomTimeCombos)
    miniSchedule(T (except for first slot), rooms, classes, P, S, schedule takenRoomTimeCombos)
    return schedule
```

8 Time analysis - Extensions

8.1 miniSchedule()

Consider the function miniSchedule(). It contains a pair of nested for loops *for(each room in R)* and *for(each timeslot in T)*. This pair of loops will terminate after $O(rt)$ iterations, as the set of rooms in miniSchedule() has a size that is a function of r, and a set of time slots whose size is also a function of t. Because classes are stored in a linked list that was ordered from greatest to least preferred during preprocessing, retrieving the most preferable class is an $O(1)$ operation. The while loop, as was shown in the previous time analysis section will terminate in $O(t)$, because there are at most $t-1$ possible professors scheduled for any one timeslot. Similarly, checking whether a professor has a time conflict for any particular timeslot is an $O(1)$ operation, as each professor schedule can have a maximum of 2 different times.

Similarly, moving onto the next class in the *classes* linked list is an O(1) operation, as was stated prior, as is prepending all skipped classes back into *classes*. Finally, enrolling all students who prefer class and do not have a time conflict for timeslot is an $O(s)$ operation, as there are a maximum of s students who prefer *class*, meaning there are a maximum of s student schedules to check, each with a maximum length of 4 times, a constant.

Finally, calling optimizeRoomSchedule() is an $O(t^2s)$ operation, as was shown prior. Because it is called a maximum of $O(r)$ times for each iteration of the *for(each room in R)* loop, this adds an additional $O(r^2t^2s)$.

Therefore, miniSchedule will complete in $O(rt(t+s)+O(r^3t^2s) = O(r^2t^2s)$. Thus the function completes in $O(r^2t^2s)$ time.

8.2 accessibleSchedules()

The *for(each accessible room, sorted from least to greatest)* loop will iterate a maximum of $O(r)$ times.

The *for(each accessible esem)* loop will iterate a maximum of $O(c)$ times. Tallying the numbero f people who prefer the esem in question could be completed and stored during preprocessing and sorting the classes linked lists, making determining this an $O(1)$ action. Enrolling all students who prefer the esem is an $O(s)$ action. Appending the room and time combination to $takenRoomTimeCombos$ is also an $O(1)$ action, making this mini loop $O(cs)$.

The *for(each time in T)* loop will iterate $t-1 = O(t)$ times. The *for(each class in accessibleClasses)* loop will iterate a maximum of $O(c)$ times. Checking capacity of the room, professor conflicts, and the number of people who prefer the class are all three $O(1)$ operations, as was shown previously. Enrolling all students is an $O(s)$ operation, as checking each student's schedule for conflicts is only $O(1)$ and there are a maximum of s students who prefer any one class.

"Skipping" any room-time combinations has no effect on complexity, and would involve simply moving onto the next iteration of the *for(each time in T)* loop. Skipping over all professors with a time conflict is a maximum $O(t)$ operation.

Therefore, the *for(each class in accessibleClasses)* has a complexity of $O(tcs)$. Combining all of these observations shows that this algorithm will terminate in $O(r(cs+tcs) = O(rtcs)$.

8.3 extended()

The number of domains depends on input, however the number of classes itself is constant. Ordering each Linked List of classes can be achieved during preprocessing in $O(clogc)$ for some small constant number of domains.

Preprossessing also includes initializing each whoPrefers[class] array of students, and then sorting the list. This involves going through s preference lists, each of length 4. If whoPrefers[class] is an array or linked list, then adding new items can be achieved in $O(1)$ time. Sorting each whoPrefers[class] set can also be achieved in $O(slogs)$ time using a specialized mergesort that takes into account student major and year when ordering students. Thus this step can be achieved in $O(slogs)$ time.

schedule is a 2D array of size t x r. Creating and initializing this empty schedule is an $O(rt)$ operation.

$accessibleSchedule()$ is an $O(rtcs)$ operation, as was shown previously.

$miniSchedule()$ is an $O(r^2t^2s)$ operation, as was shown previously, and it will be called twice.

Therefore, the total complexity of the entire algorithm with extensions is:

$$O(clogc + slogs + rt + rtcs + r^2t^2s) = O(r^2t^2s)$$

9 Proof of Correctness - Extensions

In order to show that these extensions are correct, we must demonstrate the following:

1. Proof of termination for all extensions
2. Proof that no class with a student who needs accommodations will be scheduled in a non-accessible classroom
3. Proof that esems will not conflict with other classes
4. Proof that no two classes will be scheduled for the same room at the same time
5. Proof that each class will be scheduled for a room in the correct domain

9.1 Proof of termination

Pseudocode and time analysis show that this algorithm will terminate in $O(r^2t^2s)$ time.

More specifically, all of the for loops in $miniSchedule()$ must terminate in a set number of iterations. The while loop in $miniSchedule()$ must terminate after $O(t)$ iterations, as for any set of t timeslots, there can only be t possible professors scheduled at the same time. Thus $miniSchedule()$ terminates.

$accessibleSchedules()$ meets the same standards: all for loops must terminate after a set number of iterations. "Skipping" a room because all preferred students will not fit in a capacity will have no effect on time analysis and will not cause any backtracking. Skipping over classes due to professor conflict was also previously shown to be a terminating operation - terminating after $O(t)$ iterations. Thus $accessibleSchedule()$ terminates.

extended() has two for loops that must terminate after a set number of iterations. Sorting each accessible rooms and accessible classrooms must terminate, as there are a set number of rooms and classrooms. $accessibleSchedule()$ and $miniSchedule()$ were already shown to terminate. Thus this algorithm terminates.

9.2 Proof that no class with a student who needs accommodations will be scheduled in a non-accessible classroom

Pseudocode states that, for each student who requires accommodations will have all their preferred classes marked as needing to be accessible - the accessibleClasses and accessibleEsems sets. The set of all accessible rooms is then conglomerated in the accessibleRooms set. These are the class sets and rooms sets fed into accessibleSchedule(). Therefore, for any student requiring accommodations, all classes scheduled *must* be in an accessible classroom, as these are the only classrooms available to choose from.

9.3 Proof that esems will not conflict with other classes

Pseudocode dictates that all Esems will be scheduled in a single time slot - a time slot that is excluded from the schedule for non-esem classes. Therefore, no non-Esem class can have a time conflict with an Esem.

9.4 Proof that no two classes will be scheduled for the same room at the same time

$accessibleSchedule()$ marks each scheduled (room, time) combination in $takenRoomTimeCombos$.

$miniSchedule()$ iterates through each room and time combination with no backtracking. If any such (room, time) combination exists in $takenRoomTimeCombos$ (i.e. already has a class scheduled there), that combination will be skipped. Likewise, $miniSchedule()$ does not schedule non-Esem classes in the Esem time slot, and only schedules Esems in the appropriate Esem time slot.

$optimizeRoomSchedule()$ has already been proven to not schedule two classes for the same room at the same time. Therefore, no two classes can be scheduled at the same time.

9.5 Proof that each class will be scheduled for a room in the correct domain

Pseudocode indicates that during preprocessing, each class will be assigned to a specific linked list of classes based on the class's domain - i.e. the domain of room in which the class must be taught. Then, when going through each room, both miniSchedule() and accessibleSchedule() will choose an appropriate class based on the current room's domain. Thus, each class will be scheduled for a room that falls within the class's assigned domain.

10 Algorithm Discussion

Why did you choose this algorithm?

This algorithm was designed to maximize student preferences while also running in a reasonable time frame. Our original greedy approach assigned classes to the largest possible rooms in order of student preferences to ensure that as many students as possible are able to take the classes they prefer. Iterating by most preferred courses also made it simple when assigning room sizes, as the biggest rooms would go to the courses that were preferred by the most students. We later added conflict adjustment in the hope that it would increase fit. While we were able to slightly increase the fit, it was still less than we had originally hoped.

What complications did you encounter while creating it?

One of the biggest factors that inhibited our greedy approach was professor conflicts. We created holdClass to ensure that when professor conflicts arise, the courses are put into the next-optimal place, if such a slot is available. This became useful when implementing extensions because many more kinds of conflicts became possible. We also had significant difficulties implementing conflict scheduling. The way we initially implemented the code decreased our fit. We were eventually able to debug the numerous issues with conflict schedule, but it did not give us the boost in fit that we were hoping for.

What characteristics of the problem made it hard to create an algorithm for?

The real-world constraints and data tanked our fit and were difficult to implement. The "base" data as we call it (without modifications or additional data) worked great and had reasonable fit. However, restricting classes to certain rooms and other restrictions had a huge impact (much larger than we initially anticipated). Also, the real BMC data had conflicting time slots, which were a nuisance to handle. It would be much simpler to have non-overlapping time slots, even if classes did not take up the entire thing. For example, 1-2:30 classes could be treated in the same "afternoon block" as 1-4:00 classes.

What algorithmic category or categories does your algorithm fall into?

The base algorithm was a greedy algorithm optimized by student preferences and it did not back-track. With conflict scheduling enabled, it is not strictly greedy because it allows back-tracking. However, it does improve the fit from the base greedy approach.

Which algorithms that we have studied is your algorithm similar to?

This algorithm is unlike most of the algorithms we have studied in class. It is similar to box stacking in the sense that we are trying to maximize using multiple inputs, but this problem uses conflict scheduling to improve the greedy approach.

11 Experimental Analysis

11.0.1 Testing Results

For our analysis, we created five series of theoretical data (a-e), one series modeling Bryn Mawr data (f), and one series with extreme constraints to serve as a stress test (s). A series consists of a number of rooms, classes, class times, students, and majors, which we list below:

- a: 10, 50, 5, 500, 10
- b: 10, 100, 10, 1000, 20
- c: 15, 150, 10, 1000, 20
- d: 25, 200, 8, 2000, 30
- e: 25, 250, 10, 2000, 30
- f: 65, 500, 6, 2000, 40
- s: 5, 100, 5, 1000, 10

Each series contained 10 randomized constraints and preference files, the results from each constraints/preference pair we averaged to create the values listed in the tables below. The results are separated by run files: RP (no extensions) and RP2 (extensions).

For time analysis, we also created variable-isolated data sets (shown below). For each series, several sets of 10 files were created and the average was taken. All variables are the same as the baseline except for the one specified for that set. All time analysis was run on the same machine to ensure continuity.

- Baseline: 10 rooms, 50 classes, 5 times, 500 students, 5 majors
- TimeSeries: # of time slots adjusted in sets of 10, 20, 40
- RoomSet: # of rooms adjusted in sets of 20, 40, 80

- ClassSeries: (factored in $O(cs)$) # classes in sets 100, 200, 400, 800
- StudentSeries : (factored in $O(cs)$) # of students in sets 250, 750, 1000, 1250

The general fit of the first two approaches will be shown graphically in the next several subsections, but the data is also listed in the table below.

Fit (out of 1.0)	**A**	**B**	**C**	**D**	**E**	**F**	**S**
RP	0.7478	0.830125	0.85025	0.811738	0.8531	0.6889	0.259125
RP2	0.56275	0.66755	0.6971	0.6841	0.719263	0.626388	0.216475

Table 1: Lists the fit ratio (out of 1.0) for sets A-F and S for RP and RP2

11.0.2 RP (no extensions)

First, we will examine the fit of base RP (all graphs shown in blue). The fit for sets A-F and S are illustrated on the following bar graph.

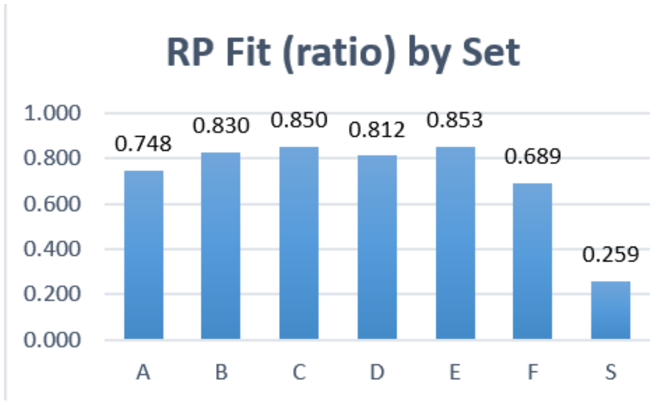

Figure 1: Average RP fit ratio (with 1.0 being optimal) for sets A-F, S

RP performed relatively well with the randomized data, with the lowest fit outside of the stress test being on set F (designed to be similar to BMC data) at 0.689. For sets A-E, our algorithm was able to meet approximately 75-85% of student requests. This indicates that our algorithm performs reasonably well on simulated data.

Using our variable-isolated data sets for time analysis, we found positive linear correlations with increases in the number of time slots, rooms, and students. This is attributed to the modified time analysis with conflict scheduling, with the dominant term being $O(r^2t^2s)$. Of these, time slots had the lowest R^2 at 0.8949, but a quadratic regression did not improve the fit. We found no correlation with the number of classes. Overall, the observed run times were in line with our theoretical time analysis.

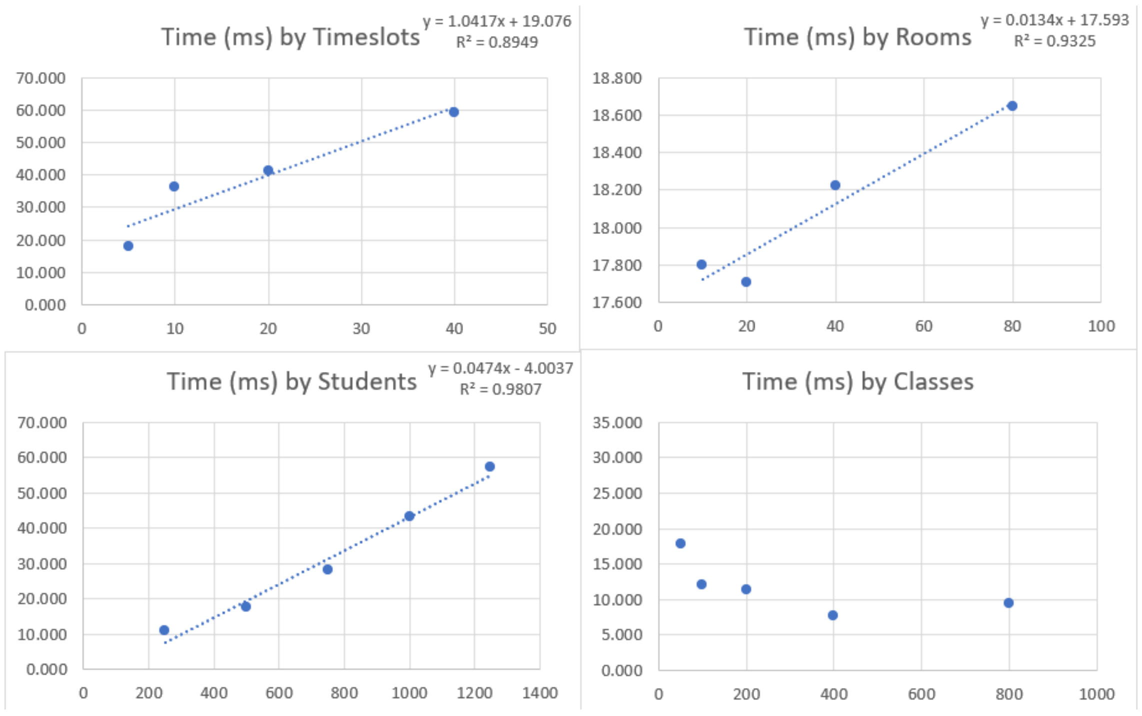

Figure 2: Average of 10 times taken to run each set in TimeSeries, RoomSeries, StudentSeries, and ClassSeries on RP. Linear regressions indicate positive trends for the former three with $R^2 = 0.8949$, 0.9325, and 0.9807. There was no significant correlation for ClassSeries.

11.1 RP2 (extensions)

Next, we will examine the fit of RP2 (all graphs shown in green). The fit for sets A-F and S are illustrated on the following bar graph.

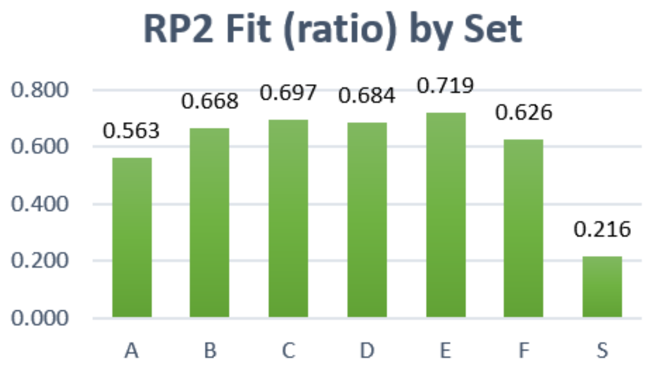

Figure 3: Average RP2 fit ratio (with 1.0 being optimal) for sets A-F, S

RP2 performed noticeably lower than RP, which is to be expected given the restrictions of the extensions. The lowest fit outside of the stress test was on set A at 0.563. For sets B-F, our algorithm was able to meet approximately 62-72% of student requests. This indicates that our algorithm performs reasonably well on simulated data given the constraints of the extensions.

Using our variable-isolated data sets for time analysis, we found positive linear correlations with increases in the number of time slots, rooms, and students. This is attributed to the modified time analysis with conflict scheduling, with the dominant term being $O(r^2t^2s)$. Of these, time slots had the lowest R^2 at 0.7709, but a quadratic regression did not improve the fit. We found no correlation with the number of classes. Overall, the observed run times were in line with our theoretical time analysis.

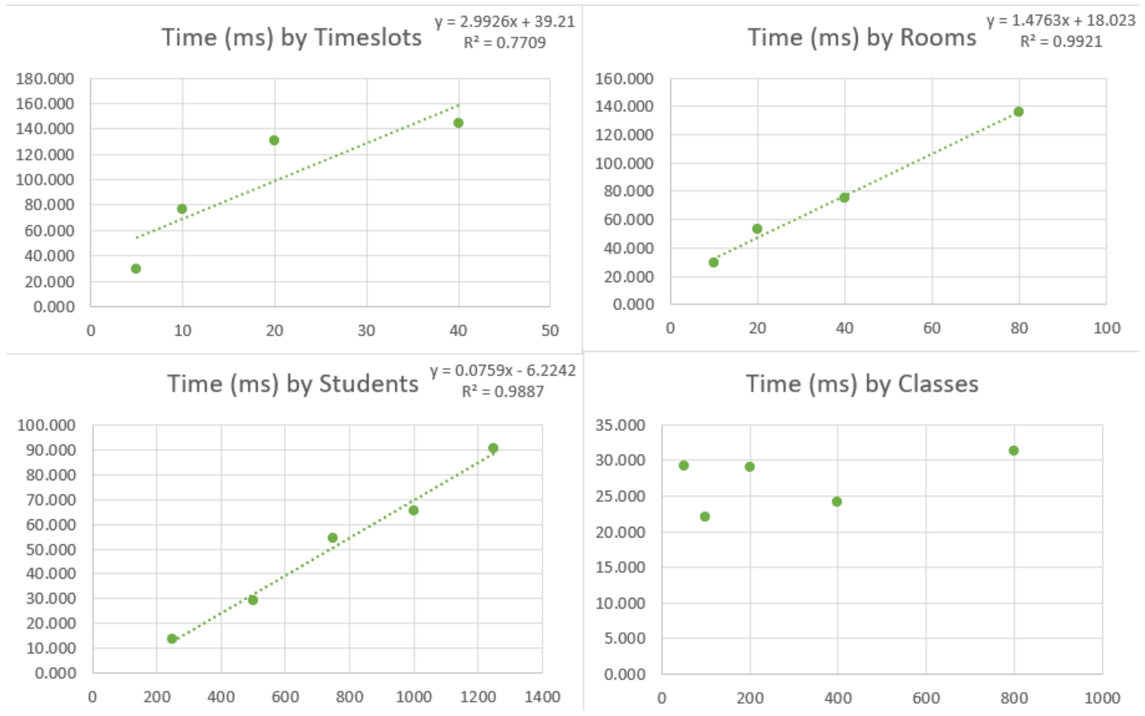

Figure 4: Average of 10 times taken to run each set in TimeSeries, RoomSeries, StudentSeries, and ClassSeries on RP2. Linear regressions indicate positive trends for the former three with $R^2 = 0.7709$, 0.9921, and 0.9887. There was no significant correlation for ClassSeries.

11.2 RP-BMC (extensions w/ BMC data)

RP-BMC implements the same extensions as RP2 does, the difference being RP-BMC's constraints and preference parsers are modified to be able to read the provided real life Bryn Mawr registrar data. Some factors - such as room accessibility, student major, and student accessibility needs - were unavailable to us in the conglomerated "real life data" sets, yet were still required for our extensions. For our implementation, room accessibility was determined anecdotally based on buildings known to have elevators and that meet other accessibility requirements. We assumed all time slots were nonoverlapping.

Student majors were automatically generated based on the subject each student had most often on their course schedule (if all subjects appeared an equal amount of times, one was randomly selected as the "major"), unless the student was a sophomore or freshman, in which case an "undeclared" major was assigned. Student class year and accessibility requirements were randomly generated, with each student having a 1/100 chance of having accessibility needs. Similarly, domains were determined by class subject, with "STEM" courses such as MATH, CMSC, PHYS, CHEM, etc, being assigned to the "STEM" domain, which only included rooms in Bryn Mawr's Park Science Center. All other buildings and subjects were relegated to the "humanities" domain.

Because of these randomly generated elements, all generated schedules varied in fit, even for the same semester data file. For this reason, for each semester data file, ten different schedules were generated using this algorithm, each with different random elements. From these ten schedules, an average runtime and fit were determined.

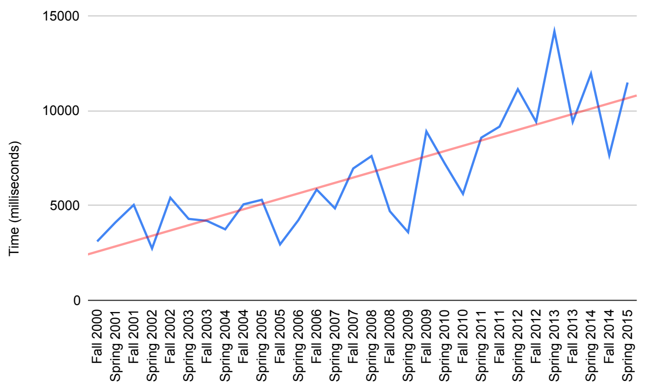

Figure 5: Average runtime for each semester data file - compiled from 10 different trials.

Loosely speaking, average runtime tended to increase somewhat in later semesters. This can be attributed to an increase in the number of time slots and courses over the period of data that was provided. We also saw a difference between the fall and spring semesters, with fall semesters taking more time.

From these findings, we believe our fit is generally consistent in the 40-60% range. This is a significant improvement from our trials without multiple sections, with an increase of approximately 20%. This graph also lists results when accessibility is not a bottleneck (with all classrooms as

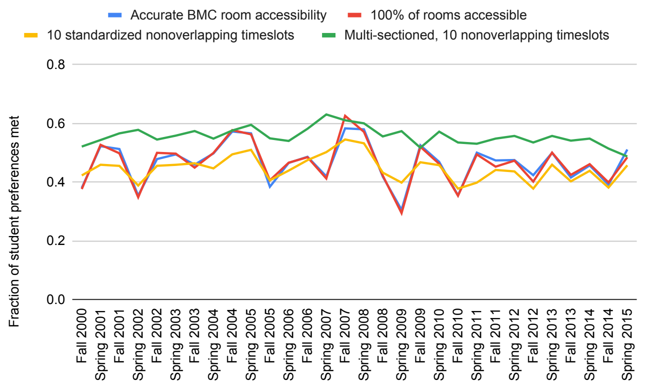

Figure 6: Average fit for each semester data file - compiled from 10 different trials. Additional scenarios were also tested, such as all rooms being accessible, limiting the time slots to only 10 non-overlapping time slots, and allowing multiple sections of individual classes. Note the multisectioned one, while also being truest to life – as each data file assumes multiple sections for some classes – does not take into account multiple professors for each class, as it relies on the generated constraints and preference files, which only list one professor per class - so in actuality, a fit would likely be much higher.

accessible). We saw the highest fit across almost every semester with multiple sections of popular courses and non-overlapping time slots.

12 Recommendations to the Registrar

- Implement set times for Haverford writing seminars similar to BMC Esems
- Remove overlapping time slots to reduce conflicts between courses
- Add more and/or larger sections of popular courses
- Create a checkbox in Bionic or another centralized system to maintain a list of students requiring accessible classrooms
- Maintain an up-to-date list of accessible classrooms on campus and work to increase the number of accessible classrooms on campus

As writing seminars are required courses, it would make sense to have them at a handful of set times. Bryn Mawr has already implemented this change to reduce conflicts with other courses. Haverford should follow suit because it would ultimately reduce manual work on the registrar's behalf: when students need to swap writing seminars due to conflict with required courses for their major(s), the registrar has to manually make those adjustments. There would be far fewer writing seminar swaps if the only reason was due to students preferring a different course.

Next, the registrar should reduce or eliminate overlapping course time slots. For example, there should not be classes on the same campus where one runs from 10am-11:30am and another runs from 10:30am-12pm. This method of scheduling courses creates more conflict between rooms on different time schedules, as well as more student conflicts between courses. It also leads to students leaving one class 15 minutes early and showing up 15 minutes late to the next, for which the students must get permission codes for both courses and miss content simply because of inefficient scheduling. This should also be taken into account when scheduling between campuses. Taking into account the Blue Bus is a useful strategy, but only when it is feasible to do so: for disabled students, 15 minutes is not enough time to get between campuses, especially if the Blue Bus is full. Setting a clear offset of 20-30 minutes would likely improve students' abilities to take classes across campuses without sprinting to catch the bus or make it to class on time.

Additionally, the registrar should allow for more and/or larger sections of popular courses. While this may not always be possible due to staffing restrictions, it did dramatically increase our fit for the BMC data. There are many courses that are over-enrolled every semester. For example, at Haverford, the capacity of introductory computer science (offered only in the fall) leads students to take up more space in the Bryn Mawr equivalent or drop computer science entirely. This is a frequent occurrence across all departments, which is why sections should be managed by the registrar. Using previous registration data will make it clear which classes need sections the most (since it is likely impossible to add sections to every over-enrolled course).

In terms of access issues, we found that in RP2, the primary reason that classes were not added to the schedule was due to the lack of accessible classrooms. In the simulated data, we estimated that 50% of classrooms are accessible. This number could be improved by working with ADS and facilities to modernize buildings on campus. We make two recommendations that the registrar can implement to reduce access conflicts before students even enroll. The first step is to have a list of students who require accessible classrooms. This will make it simple to identify whether students are given accessible classrooms as soon as they enroll. This information is not currently maintained (to the best of our knowledge), but would be incredibly useful when assigning courses to rooms or making swaps. The second step is for the registrar to keep a list of accessible classrooms. This should be done in coordination with Access and Disability Services as well as disabled students such as Disability Advocacy for Students at Haverford (DASH) and the Haverford Students' Council Officer of Access and Disability Inclusion.