CS 340 Group Project

Caesar Dai, Dylan Soemitro, Samuel Tan, Harrison Weinstock

September 2021

1 Abstract

Your paper should also include a one to two paragraph abstract that emphasizes your findings (but not, in detail, how you came to them). This should be readable to someone who was not in the class, so that it can be sent to the registrar as a set of recommendations!

2 Introduction

Initially all the students, classes, rooms, and time slots are unpaired. Each student has a preference list of four of the classes that they like. For each class, the instructor is known, all must be scheduled in one room with a specific time slot and a list of enrolled students.

A valid schedule would include a list of classes each assigned to a room, a time slot, with a professor, and a list of students that are enrolled in it. On top of that, each time slot cannot be assigned more than $\frac{|C|}{|T|}$ classes; No professors are assigned to teach two different classes at the same time slot; No student is enrolled in more than one classes that meet at the same time; No two classes at the same time slot are assigned to the same room; The number of students in each class must not exceed the classroom size.

3 Algorithm Description

We start by sorting the classes based on students' interests. A class's interest score will increment by one if a student has it on his or her class preference list. Subsequently, all of the classes will be sorted such that the class with the highest interest score will be at the start of the list.

Next, we sort the available time slots such that the one with the least number of overlaps with other time slots comes first. Subsequently, we sort all the available rooms in decreasing order based on their capacity.

Now, we can begin to match classes to professors, time slots, and rooms. To minimize conflict in the matching process, we iterate through the sorted list of classes and assign each class a professor that can teach that class, an available time slot with the least overlap, and the biggest room available at that time.

We will repeat the above process of matching classes, time slots, professors, and rooms until we have attempted to schedule every single class.

Finally, we terminate the scheduling process and begin to enroll students in as many classes as they have expressed preference for while making sure that those classes do not have overlapping time slots. This would allow us to generate a valid schedule.

4 Pseudo Code

S = A list of students and corresponding class requests for four classes
C = A list of classes is given
R = A list of classroom sizes is given in terms of the number of students the classroom can hold
T = A set of possible class times
P = A list of teachers and names of the classes they teach is given. Each teacher teaches two classes

Function `classSchedule`(S, C, R, T, P)

Let $sortedClass = sortClasses(S, C)$

Let $sortedClassTimes = sortedClassTimes(T)$

Let $roomsForClasses = identifyRoomsForClass(C, R)$;

`// roomsForClasses is a dictionary with classes as keys and ordered list of rooms as values`

Let $roomAvailability = setUpAvailability(R, sortedClassTimes)$;

`// roomAvailability is a dictionary with room as keys and a dictionary as the values. This other dictionary will have time slots as keys and boolean as values`

Let $profAvailability = setUpAvailability(P, sortedClassTimes)$;

`// profAvailability is a dictionary with professor as keys and a dictionary as the values. This other dictionary will have time slots as keys and boolean as values`

Let *matches* be a dictionary

for $c_i \in sortedClass$ **do**

  Let p be the professor for c_i

  for $t \in sortedClassTimes$ **do**

    if $matches[c_i]$ *exists* **then**

      break

    else if $profAvailability[p][t]$ *is true* **then**

      for $r \in roomsForClasses[c_i]$ **do**

        if $roomAvailability[r][t]$ *is true* **then**

          Set $matches[c_i]$ to (t, r, p)

          Set $roomAvailability[r][t]$ and $profAvailability[p][t]$ to false

          Multiply t's conflicts by the minimum between the number of students and the room capacity for r

          break

      end

  end

  Sort $sortedClassTimes$ by increasing number of conflict

end

Let *Score* be $enrollStudents(matches, S, R)$.

return *matches and Score*

Function `sortClasses(`S, C`)`

Let $classInterestCount$ be a dictionary, with the keys being all the classes and the values being set to 0
for $s \in S$ **do**
  for $p \in s$*'s preferences* **do**
    Increment $classInterestCount[p]$ by 1
  end
end
Convert $classInterestCount$'s key-value pairs into a list
Sort $classInterestCount$ by decreasing amount of interest
return $classInterestCount$

Function `sortClassTimes(`T`)`

Let $classTimeConflicts$ be a dictionary
Let $sortedClassTimes$ be a list
Sort T by earliest start times
for $t \in T$ **do**
  Set $classTimeConflicts[t]$ to 0
  for $t' \in T$ **do**
    if t *overlaps with* t' **then**
      Increment $classTimeConflicts[t]$ by 1
  end
end
for $t \in T$ **do**
  Insert $(t, classTimeConflicts[t])$ into $sortedClassTimes$
end
Sort $sortedClassTimes$ by increasing number of class time conflicts
return $sortedClassTimes$

```
Function identifyRoomsForClass(R, C)
    Let roomsForClasses be a dictionary
    for c ∈ C do
        for r ∈ R do
            if c ∈ roomsForClasses's keys then
                Insert r and its room capacity value into
                 roomsForClasses[c]
            else
                Set roomsForClasses[c] to be a list and insert r and
                 its room capacity value into it
            end
        end
        Sort roomsForClasses[c] in decreasing order of room
         capacity
    end
    return roomsForClasses

Function setUpAvailability(input P or R, sortedClassTimes
 T_s)
    //generating dictionary for Professor availability and Room
     availability
    Let availability be a dictionary
    for r ∈ input do
        Let isOpen be a dictionary
        for t ∈ sortedClassTimes do
            Set isOpen[t] as true
        end
        Assign isOpen to availability[r]
    end
    return availability
```

```
Function enrollStudents(matches, S, R)
    Let roomCapacities be a dictionary keyed by the rooms with
     values being another dictionary. Let the inner dictionary be
     keyed with possible time slots and values being the max
     capacity of that room.
    for s ∈ S do
        Let preferences be the classes s wants to take
        Apply interval scheduling to preferences and let
         enrolledClasses be the result
        for class ∈ enrolledClasses do
            Let r be the class for matches[class]
            Let t be the time slot for matches[class]
            if roomCapacities[r][t] > 0 then
                Increment Score by 1
                Decrease roomCapacities[r][t] by 1.
            end
        end
    end
    return Score
```

5 Time Analysis

5.1 Time Analysis for *setUpAvailability*

This function first creates a dictionary *availability* which is trivially $O(1)$. Next, it executes a For loop that has either $|P|$ or $|R|$ number of iterations. In each of the iteration, it creates a dictionary *isOpen* in $O(1)$ time, starts another For loop before assigning the *isOpen* dictionary to $availability[r]$. Since *availability* is a dictionary, accessing and assigning a value to $availability[r]$ should take $O(1)$ time. The nested For loop has t iterations. For each iteration, it performs an $O(1)$ time operation by setting $isOpen[t]$ as true since setting a dictionary's value takes $O(1)$ time. This allows us to conclude that the nested For loop takes $O(t)$ time which makes the outer For loop take $O(tp)$ and $O(|T||R|)$ time. Since the outer For loop dominates the runtime compared to all the other $O(1)$ time operations, the runtime for *setUpAvailability* are $O(|T||P|)$ and $O(|T||R|)$.

5.2 Time Analysis for $identifyRoomsForClass$

This function first creates a dictionary $roomsForClasses$ which is trivially $O(1)$. Next, it executes a For loop that has $|C|$ number of iterations. In each of the iteration, it executes another For loop and performs a sorting operation. In the inner For loop, it has $|R|$ number of iterations. In each of the inner For loop iterations, we will access the value $roomsForClasses[c]$ and assign a value to it. Since $roomsForClasses$ is a dictionary, accessing it and assigning values to it are $O(1)$ time operations. Inserting r and r's room capacity value can be done by putting them together into a tuple (r, r's room capacity) before inserting it into the list that is the value for $roomsForClasses[c]$. Setting up a tuple and inserting the tuple into a list are both $O(1)$ time operations. Since all the functions in the inner For loop are $O(1)$ time operations, the inner For loop runs in $O(|R|)$ for its $|R|$ number of loops. The sorting function in outer For loop requires us to sort the $O(|R|)$ length lists for some $roomsForClasses[c]$ which means that it is an $O(|R|\log(|R|))$ operation. Since this dominates the runtime for the inner For loop, each iteration in the outer For loop takes $O(|R|\log(|R|))$. Since we have $|C|$ number of iterations in the outer For loop, the outer For loop takes $O(|C||R|\log(|R|))$ time. Hence, we can conclude that the $identifyRoomsForClass$ function takes $O(|C||R|\log(|R|))$ time.

5.3 Time Analysis for $sortClassTimes$

Initializing $classTimeConflicts$ and $sortedClassTimes$ should take $O(1)$ time since they are all empty. Sorting T which is a list by earliest start times would take $O(|T|\log(|T|))$ time.

Next, we have our first For loop in the function. This For loop has $|T|$ iterations. In each of this iteration, we first perform an $O(1)$ time operation where we access the dictionary and set a value for $classTimeConflicts[t]$. Subsequently, we begin an inner For loop that has $|T|$ iterations. In each of the inner For loop's iterations, checking if t overlaps with t' is just a matter of checking if the start and end times noted in each t share some overlap and hence will be $O(1)$. Accessing $classTimeConflicts[t]$ and incrementing it by 1 will also be $O(1)$ for a dictionary. Hence, the inner For loop has $|T|$ iterations of $O(1)$ time operations, giving it a $O(|T|)$ runtime. Likewise, the outer For loop has $|T|$ loops with $O(|T|)$ runtime for each loop and hence will have $O(|T|^2)$ runtime.

Subsequently, there is another For loop that has $O(|T|)$ iterations. To check if t is during lunchtime, it is simply a matter of comparing t with a fixed lunchtime definition and checking for overlap. Hence, this has to take $O(1)$ time. Multiplying $classTimeConflicts[t]$ involves accessing the dictionary and modifying the value in the dictionary takes $O(1)$ time. Inserting a tuple that contains t and $classTimeConflicts[t]$ into a list $sortedClassTimes$ should take $O(1)$ time. Since the For loop has $O(|T|)$ loops that each take $O(1)$ time, this For loop ends in $O(1)$ time.

Lastly, the sorting operation sorts $O(|T|)$ items and hence will take $O(|T|\log(|T|))$ time. Since the nested For loop's $O(|T|^2)$ time dominates the run time, the $sortClassTimes$ function takes $O(|T|^2)$ time.

5.4 Time Analysis for *sortClasses*

This function first creates a dictionary $classInterestCount$ that is trivially O(1). Next it executes a nested For loop. The two For loop on the outside run $4|S|$ number of iterations. The outer For loop runs $|s|$ times and the inner For loop runs 4 times since all the student's preference list has a fixed length of four classes. Then, because $classInterestCount$ is a dictionary, checking whether the professor teach any of the classes that are on the students' preference list, incrementing the $classInterestCount[p]$, and setting $classInterestCount[p]$ are O(1) time operations. Converting the $classInterestCount$'s key value pairs into a list will take $O(|C|)$ time since we will simply turn each of the $|C|$ key-value pair in the $classInterestCount$ dictionary into a tuple before inserting it into the list. Accessing the key-value pairs and inserting them into a list takes $O(1)$ time for each tuple and hence should take $O(|C|)$ time in total. The sorting function following the nested For loops require us to sort O($|C|$) length list for some $classInterestCount$, which means that it is an O($|C|log|C|$) operation. We know that there are always more students than class, so $|S| > |C|$, but not by a lot that in real life number of students is roughly two times the number of classes. Therefore, since $log|C| > 2$, we have $|C|log|C| > 2|c| > S$. Thus, we can conclude that $sortClasses$ takes $O(|C|log|C|)$ time.

5.5 Time Analysis for *enrollStudents*

Initializing our roomCapacities takes rt iterations, where r is the number of rooms scheduled in $matches$ and t are the number of time slots scheduled. The first For loop of this function takes $O(|S|)$ iterations, as we iterate through every student's preference list to determine what classes they can enroll in. The For loop nested within this one takes $O(1)$ time, as we know that the students have a fixed preference list of size 4. We look up each class c that the student wants to take in $matches$, which has constant lookup. We then determine if $roomCapacities[c] > 0$ for each c i.e. the class is not fully scheduled, and then perform interval scheduling on the classes which are not fully scheduled. Interval schedule takes $O(nlogn)$ time, but we have a fixed list of less than 4 classes, so it takes constant time. The rest of the operations in the for loops take constant time. So we know that enroll students has a runtime of $O(rt + |S|)$.

5.6 Time Analysis for *classSchedule*

sortClasses will take $O(|C|log|C|)$ operations to set up the *sortedClass* list.

sortClassTime will take $O(|T|^2)$ operations to set the *sortedClassTimes* list.

identifyRoomsForClass will take $O(|C||R|\log(|R|))$ time to set up the *roomsForClasses* dictionary.

setUpAvailability$(R, sortedClassTime)$ will take $O(|T||R|)$ to set up *roomAvailability* dictionary

setUpAvailability$(P, sortedClassTime)$ will take $O(|T||P|)$ to set up *profAvailabilty* dictionary

This function contains a triple nested For loop. The first For loop has $|C|$ iterations. The second For loop will run $|T||C\log C|$ iterations because the algorithm re-sort the class times by increasing number of conflicts after each match is inserted into *matches*. The third For loop will have $|C|$ iterations if every operation within operates at constant time. Inserting match to *matches* dictionary will have $O(1)$ operation. The inner For loop has O($|R|$) operation, the second For loop has O($|T||R|$) operation, and the outer For loop has O($|C|(||T||R| + |T|\log(|T|))$ iterations.

Since *profAvailability* is a dictionary, it will take $O(1)$ to check whether a professor is available at a given time and whether there is a corresponding STEM/humanity course time that conflicts with c_i. Checking whether c_i is a first year language/writing seminar will have $O(1)$ operation as well. *profAvailability* and *roomAvailabilty* are dictionary, checking their boolean value and setting them to false after inserting into *matches* will take $O(1)$ operation. Multiplying t's conflicts by the minimum between the number of students and the room capacity for r will also take $O(1)$ operation.

enrollStudents after the triple nested For loop will run O(S) times.

Which gives us:

$$O(|C|(||T||R| + O(|C|log|C|) + |T|\log(|T|)) + O(|T|^2) +$$

$$O(|C||R|\log(|R|)) + O(|T||R|) + O(|T||P|) + O(S)$$

Since $classSchedule$, $identifyRoomsForClass$, $sortClasses$, $enrollStudents$ runtime dominates other functions, and we drop $|T|$ because it is a small constant. Thus, we can conclude that the $classSchedule$ function will run:

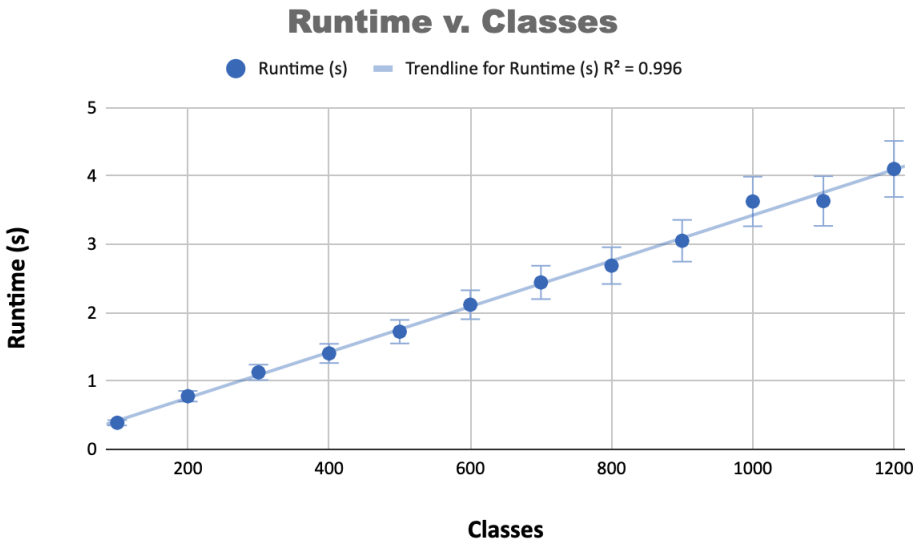
$$O(|C||R|) + O(|C||R|\log(|R|)) + O(|C|log|C|) + O(S)$$

$$\Rightarrow O(|C||R|\log(|R|)) + O(|C|log|C|) + O(S)$$

6 Experiment Design

To test how the number of students and classes affect our algorithm, we tested the *percent optimality* and *runtime* on different sizes of classes (150, 250, 500, 750) and students (450, 750, 1000, 1500, 2000, 5000, 10000, and 20000). We varied each variable independently while holding the other constant.

6.1 Classes



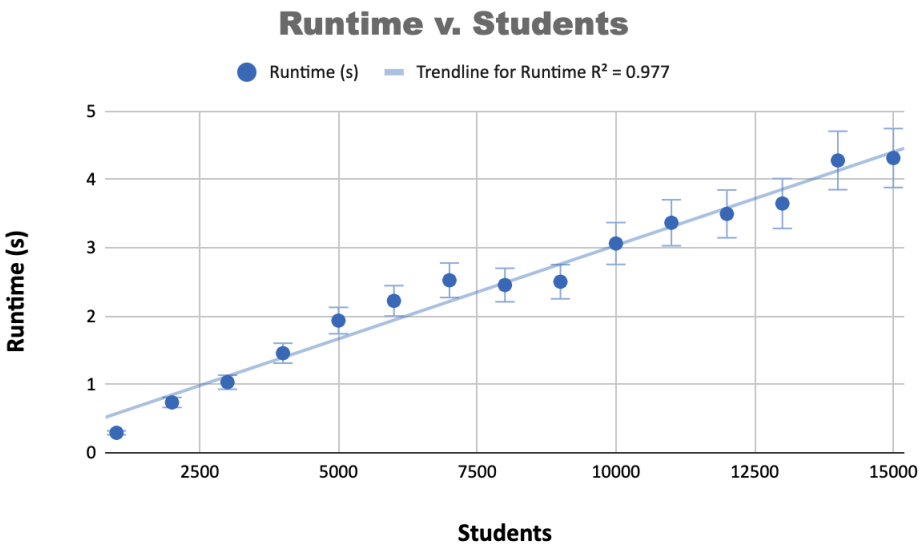
When C is varied independently, we obtain a new runtime:

$$O(|C||R|\log(|R|)) + O(|C|\log|C|) + O(S)$$

$$\Rightarrow O(|C|\log|C|)$$

We can see that our algorithm runtime appears linear with a R^2 value of 0.996 when we vary $Class$ as an independent variable. This is because the range of experimental data we used for $Class$ (from 200 to 1200) does not get large enough for the logarithmic term to be of effect. Since $\log(200) \approx \log(1200)$, $C\log(C)$ is effectively the same as C for the range of data that we tried. Thus, $log|C|$ can be expected to be a constant for any realistic class sizes even for large institutions and the linear runtime shown in our graph matches what we would expect from our algorithm.

6.2 Students



When S is varied independently, we obtain a new runtime:

$$O(|C||R|\log(|R|)) + O(|C|\log|C|) + O(S)$$

$$\Rightarrow O(S)$$

Since we can see that our algorithm runtime is increasing in a linear fashion with an R^2 value of 0.977 when we vary *Students* as an independent variable, our algorithm's runtime does scale linearly with S.

6.3 Experimental Data

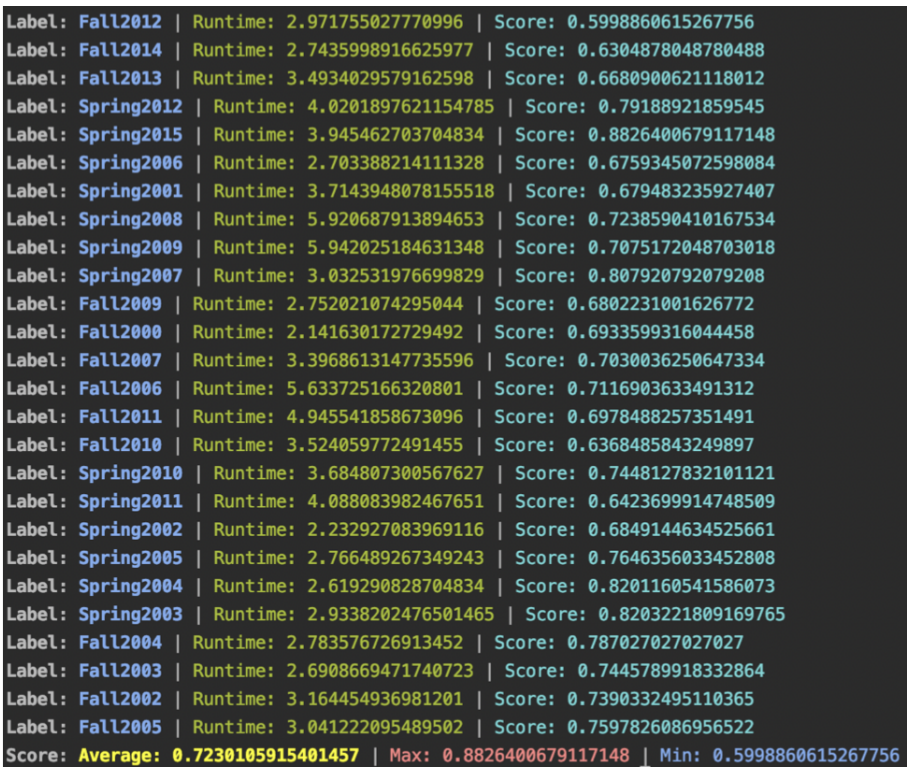
no.	Room	Classes	Times	Students	Best	Experimental	% Optimality	Runtime
1	30	150	30	450	1800	1309	0.72744	0.054675
2	30	150	30	1000	4000	2953	0.738175	0.12008
3	30	150	30	2000	8000	5951	0.7439125	0.212848
4	30	150	30	5000	20000	14376	0.71881	0.54147
5	40	250	30	750	3000	2333	0.777833	0.14226
6	40	250	30	1000	4000	3120	0.77995	0.18786
7	40	250	30	2000	8000	6284	0.78552	0.34251
8	40	250	30	5000	20000	15542	0.777115	0.85408
9	40	250	30	10000	40000	30313	0.75783	1.86618
10	40	500	30	1500	6000	5300	0.883366	0.561555
11	40	500	30	2500	10000	8837	0.88373	0.95605
12	40	500	30	5000	20000	17627	0.88137	1.7765
13	40	500	30	10000	40000	34365	0.859125	3.8215
14	40	500	30	20000	80000	66593	0.8324125	7.4153
15	40	750	30	2500	10000	9219	0.92185	1.21992
16	40	750	30	5000	20000	18386	0.9193	2.34594
17	40	750	30	10000	40000	36498	0.9124425	4.4525
18	40	750	30	20000	80000	71414	0.89268	9.01789

This table summarizes our findings when we ran our algorithm on generated data. We varied the number of rooms, classes, students, and observed how it affected our optimality and runtime. Increasing the number of classes was the largest factor in improving our fit, which makes sense as we can schedule more students into more classes (providing that the preferences are randomly generated and there is not a huge skew in preferences for particular classes).

7 Solution Quality Analysis

A schedule is considered valid if it meets the class, room size, class time, teacher, and student preference constraints. In order to calculate the fit, we take the sum of the number of total class slots filled, divided by the total number of preferences. Thus, we calculate the best fit score with the formula:

$$Fit = \frac{Number\ of\ Preference\ Fulfilled}{Total\ Number\ of\ Preferences}$$



After running our algorithm on all of Bryn Mawr's unmodified registrar data, we can see that our algorithm is able to achieve an average of Fit score of 72.23%, maximum fit score of 88.26%, and a minimum Fit score of 59.99%. Thus, experimentally our algorithm is always able to achieve at least 59.99% of the best Fit score.

To further analyze the quality of the algorithm, we compared how the math courses were scheduled by our algorithm with how they were in reality.

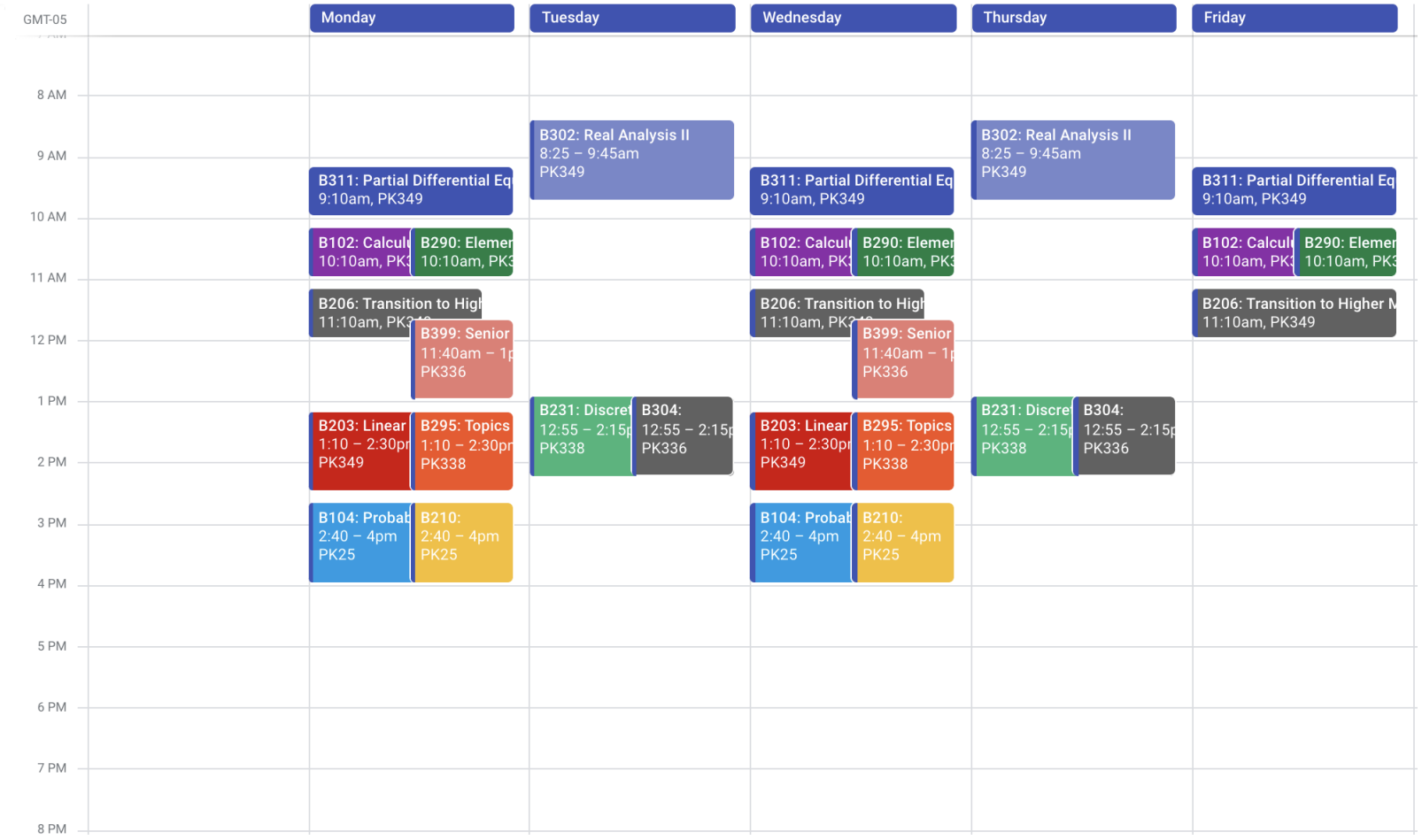


Figure 1: BMC Registrar Schedule Spring 2015

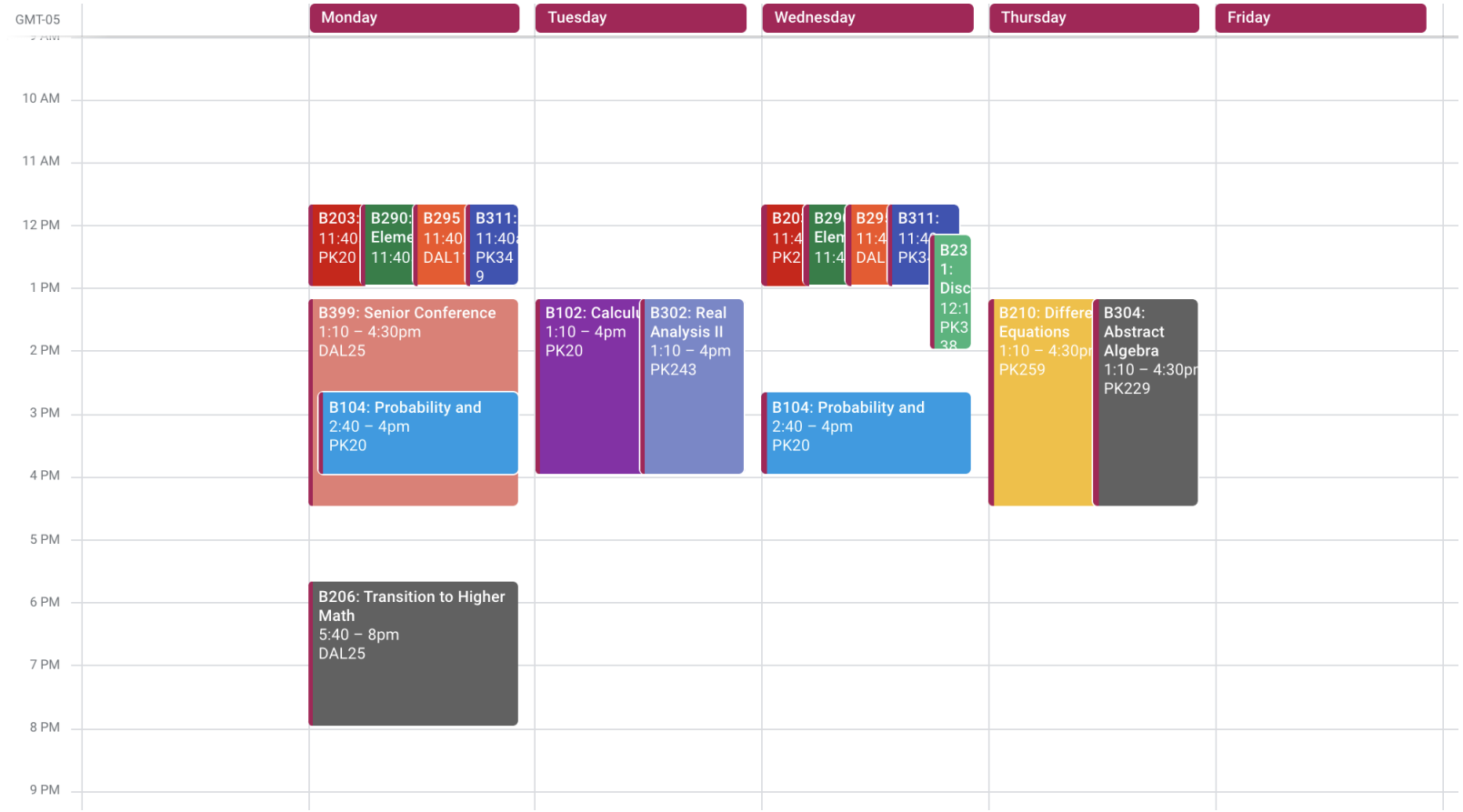


Figure 2: Algorithm Schedule for Spring 2015

8 Proof of Correctness

8.1 Proof of Termination

To consider whether our main function *classSchedule* terminates, we must first prove that all our secondary functions that *classSchedule* uses also terminate, since we define variables using these functions. First, consider the function *sortClasses*. We know the first For loop will have $|S|$ iterations, with the For loop inside it running 4 times since the student's preference list has a fixed length of four classes. Therefore, there are exactly $4|S|$ total iterations, and thus this function terminates.

Consider $sortClassTimes$. The first For loop will have exactly $|T|$ iterations. Inside this for loop, we have a For loop that will have exactly $|T| - 1$ iterations. The second For loop also has $|T|$ iterations. So, we have $|T|(|T| - 1)$ iterations for the first loop, and the second loop has $|T|$ iterations. So, $sortClassTimes$ terminates.

Consider $setUpAvailability$. The parameter *input* is R, the set of rooms, or P, the set of professors. The first For loop has $|R|$ or $|P|$ iterations. $sortedClassTimes$ is a list, so the For loop nested inside the other loop has s iterations, where s is the size of $sortedClassTimes$. The For loop has total $|R|s$ or $|P|s$ iterations, so it terminates.

Now, consider the rest of the algorithm. In our time analysis, we determined that the nested for loop will take $O(|C^2||T||C \log C|)$. Therefore, we know that it terminates properly. As we showed all parts of our function terminates, the function terminates.

8.2 Proof of Schedule Validity

1. Prove that there is no teacher conflict, such that no teacher are scheduled to teach two classes at the same time.

   Proof. Assume that there is a professor scheduled to teach two classes at the same time. That means that when assigning the professor to the second class at the same time as the first, the professor was available at that time, i.e. $profAvailability[p][t] = true$. However, during the assignment of the first time, we set $profAvailability[p][t]$ to false. Therefore, a contradiction arises, and no professor can teach two classes at the same time. □

2. Prove that no classes are taking place in the same room at the same time.

   Proof. Assume that there is are more than one class taking place at the same room at the same time. That means that when assigning the second class to the same room at the same time as the first, the room was set to be available at that time, i.e. $roomAvailability[r][t] = true$. However, during the assignment of the first class to the same time, we set $roomAvailability[r][t]$ to false. Therefore, a contradiction arises, and no no classes are taking place in the same room at the same time. □

3. Prove that no rooms are scheduled to hold classes with a size larger than the room

   Proof. Assume that there is a room $r \in R$ that is scheduled to hold a class with a size larger than the room. That means that $roomAvailability[r][t]$ is true. Observe the function $identifyRoomsForClass$. If the room is too small to hold a class, it would not meet c's requirements. So, r would not be in the list $roomsForClasses[c_i]$, meaning that $roomAvailability[r][t]$ cannot be true. Therefore, a contradiction arises and no rooms are scheduled to hold classes with a size larger than the room. □

4. Prove that all classes that can be scheduled given an empty room and time slot

   Proof. Assume that there is a class c that cannot be scheduled given an empty room and time slot. That means we cannot insert (t, r, p) in $matches[c]$. First, assume there is an empty room r. That means that there exists a r in $roomsForClasses[c]$. We know know that an empty time slot t exists by our assumption. That means that $roomAvailability[r][t]$ must be true. If this is the case, then we can insert (t, r, p) in $matches[c]$, so a contradiction arises, therefore all classes can be scheduled given an empty room and time slot. □

5. Prove that no classes can be scheduled more than once

   Proof. Assume that there is a class c which is scheduled more than once. That means that we have inserted (t, r, p) to $matches[c]$ twice. However, this cannot be the case due to $matches[c]$ already existing, meaning that the For loop will break. So we cannot insert (t, r, p) to $matches[c]$ twice, and thus a contradiction arises, proving that no classes can be scheduled more than once. □

6. Prove that no enrolled student has a schedule conflict

   Proof. Assume that there is a student with a schedule conflict. That means that the student has at least 2 classes c_1, c_2, where c_1, c_2 are scheduled at overlapping or the same time, i.e. our variable $enrolledClasses$ in our function $enrollStudent$ has two classes scheduled at the same time. We assume that interval scheduling will return a valid schedule with no two classes overlapping. Therefore, there is a contradiction, and we prove that no enrolled student has a schedule conflict. □

9 Discussion

We decided to use a greedy approach to solve this problem for a few reasons. First, the size of the input data will be very large and therefore we need a fast solution. Utilizing a greedy approach allows us to give an efficient approach that is simple in its construction. Additionally, we wanted to have something that is easily implementable. Rather than get lost in implementing complicated pseudo code, utilizing a greedy approach makes the algorithm more structured and straightforward. Despite our algorithm being greedy, it does not share many similarities to other algorithms studied in this class.

Now, let us define some notation to discuss the algorithm. Let $s_i(c)$ denote the number of classes a given student s_i is able to take without conflicts given a provided schedule. We can then define $G = \sum_i^{|S|} s_i(c)$ such that G is the total number of students preferences we were able to fulfill. Then, let O be the total number of classes students wanted to take, such that if a schedule existed with no conflicts, O would be the value of G computed on that schedule. We can then formalize the goal of the algorithm as to produce a schedule that maximizes this value G and get as close to O as possible. This is a formalization of the idea of fit discussed above. Specifically, we want to be able to say that regardless of the input, there is some fraction of O that we can treat as a lower bound for G. This lower bound O is what we found to be 59.99% experimentally in the experimental results section above.

Now focusing on the algorithm itself, each choice in its design was made to optimize G. Because we are trying to maximize G, we decided that starting with the class with the most students interested, will allow us to increment G by a larger amount if successfully scheduled than any other class would be able to assuming that we can successfully schedule it. This is why we schedule the most desired class first and give it any professor, time slot, and room it needs in order to maximize the number of students that can take it.

Because we start with the most desired class, we decided to give this class the biggest room available because we aren't going to have a class that needs a bigger room later on. We also give this class a time slot with the least potential conflicts with other time slots. We decided to do this because we assume that as the schedule fills up, most if not all time slots will ultimately begin to have some classes, so filling the ones that have the least conflicts first, could allow us to minimize the number of conflicts later

on. For example, if we consider a 2 and a half hour slot in mid-morning, this time slot might conflict with many other time slots. Therefore it makes little sense to put a highly desired class in this time slot.

10 Additional Constraints:

We decided on the following constraints:

1. **First year Writing Seminar:** We know writing seminars are necessary to schedule correctly for every student who is a first year. Additionally, we know that a student cannot take 2 writing seminars consecutively. Therefore, we can schedule them in similar time slots in the smallest room available given their size (as seminars), at the end of our main loop.

   In order to implement this, we must reconsider how we sort the classes by their interest. We must move all of the first year seminars to the back of the list, such that we process them last as we know that they cannot take another writing seminar (so it is okay to schedule them in timeslots with more conflicts). Additionally, we must rethink how we match them with classes. Rather than giving them the biggest room available, we want to give them to smallest room available, as logistically it is better to have a smaller classroom for a seminar to facilitate discussion.

2. **First year language requirement:** Because students generally take only one language, we can schedule them last in timeslots with other languages. We can handle first year language classes identical to how we handle first year writing requirements. We schedule them last, and give them the smallest possible room to facilitate discussion.

3. **Classes in appropriate classrooms:** Realistically speaking, it will not be possible for a Chemistry laboratory class to be held in an auditorium. Hence, we thought that it would be important for us to make sure that our matching algorithm matches classes to the right set of possible classrooms.

   In order to implement this, we can adjust how we preprocess the classes into the $roomsForClasses$ dictionary. Specifically, we would need this dictionary to contain only the possible rooms that match a given classes requirements. This allows us to check if a room is in the building we want a class to be in and that it meets any other requirements

the class needs. In order to determine what rooms are eligible for a given class, we can parse all historical data and accumulate a list of all rooms that are have been scheduled for a given subject, and use that information to determine if we can schedule in a given room.

4. **Avoid conflicting high crossover major classes:** We realized that an important fact when considering how students choose their classes in that their classes rarely if ever chosen independently. For example, if a given student wants to take a class in biology, we would consider them more likely to also take a class in chemistry. The same might apply to Computer Science and Math, or Physics and Math. Generally, at every level (100,200,300) there is one corresponding class in the other department at that same level that we do not want to schedule at the same time, so students can take both classes.

   In order to implement this, we would need to define a list of 'corresponding' classes for each class. By 'corresponding', we mean that two classes have a high number of students that need to take both for their major requirements. As an example, this could include a Physics class and its corresponding lab. This could also include discrete math and other Computer Science classes. Another example could be 200-level Biology with 200-level Chemistry, since these classes are often taken together by majors in either department. Once we have this definition, we can add an additional check before scheduling a class that it does not conflict with any of its corresponding classes.

5. **Schedule additional classes for professors on the day/s when they already teach:** We consider the fact that professors would prefer to come in to teach not every day if possible. So when a professor is already assigned a class, when we schedule another class for that professor, in line with the greedy algorithm we check the timeslots that have the same number of minimum conflict, and schedule the non-overlapping timeslot which has the same days as the other class (if it is a valid timeslot to schedule).

6. **Pandemic Protocol:** In the case of a pandemic, we want to reduce the number of students in each classroom to facilitate social distancing. We decided that dividing the classroom sizes by 3 would help lessen the spread of a pandemic between students and professors.

In addition to the constraints above, we also included all extensions present in the real data. This includes:

1. Overlapping Time slots.
2. Time slots that are different times on different days.
3. Professors are eligible to teach an unlimited number of classes but will only be matched with at most 2.
4. Students can enroll in more or less than 4 classes.

With these constraints, we are able to run it on the real data unedited, with none of the real information missing.

11 Recommendations

Based on the results and output of our algorithm, we can analyze the factors that cause our algorithm to develop a higher percentage of fit and see how those factors could be optimized in a real life setting to allow for less overall time conflicts.

Based on the analysis of the math classes scheduled by our algorithm and how they were actually scheduled, we can suggest the following:

1. **Increase the number of classes scheduled in once a week, longer time slots**: This can be seem by comparing the generated and real schedule and noticing how heavily the algorithm prefers scheduling classes in these long slots. Because our algorithm works by picking the time slots with the least overlap first, this implies that these long slots actually minimize conflict when attempting to schedule the classes.
2. **Schedule 100 and 300 level classes together in each department**: This is an idea that is also present in the sample output schedule where it has many of the 300 levels overlapping with some of the introductory courses in a given department. This idea also makes sense intuitively since the chances of a student enrolling in both a 100 and 300 level in a given department is highly unlikely. Additionally, scheduling them at the same time minimizes their potential total conflict with other courses, allowing more students to take them, and the other courses they are interested in.
3. **Schedule each class for a given department around the same time of day**: We noticed in the sample schedule that the math courses were entirely scheduled in the afternoon of Monday through Thursday. While some students will take 3 math courses in a given semester, or

3 courses in any given department to finish their major, they are in the minority. Therefore, to increase total enrollment, it makes sense to condense these classes together such that they only take up a small section of the day allowing for flexibility elsewhere. In the example of the math schedule, we have that scheduling the courses in the afternoon will give students the entire morning to take non-math courses without them having any conflicts with math.