CS 340 Algorithms
Group Project

December 1, 2024

Cecilia Chen, Emma Yu, Yang Wu

---

1 Basic Algorithm

1.1 Description

Initially, we have the lists of classes C, classroom sizes R, possible time slots T, a list of teachers with the classes they teach P, and students' preferences with four courses they preferred S. First, we assign the *popularity* score to each class by counting the number of appearances in students' preferences S. Then we sort the classes in decreasing order based on their *popularity* scores. Also, sort the classrooms based on their *size* in decreasing order.

Next, we will decide the times and locations of the classes. The major criterion is to assign classes with higher *popularity* score to classrooms with larger *size*. We will iterate all available time slots t for each classroom r in decreasing order of their sizes. Every professor should not have two classes assigned to the same time slot for different classrooms. During iteration, starting from the first classroom from ranked C and its first time slot, we assign the class with the highest *popularity* to that classroom and time slot. Then for each classroom r_i and time slot t_i, check if any class in the set *empty* could be assigned to t_i in order[1], if so, assign them to r_i and t_i. Otherwise, store the class as the last element in *empty* and check the next class. We iterate over each row r_i and go to the next row only when we have filled the current row. Our traversal through the classrooms and time slots stops when we finish the last class in out list and the list *empty* is empty. At that point, all the classes will be assigned with classrooms and times slots with no conflicts of professors and popular classes are assigned classrooms as large as possible.

Then we can assign classes to students based on their preference lists. To ensure that every student can attend as many of their preferred classes as possible, we first assign every student a randomly selected class from their lists. If the class they pick at the first time is fully enrolled, then we assign them another randomly selected class from the preference list until they successfully enroll in a class. If all classes in the student's preference list are full before this student enrolls, we will leave the student aside until the end and we will randomly assign them an available class that doesn't conflict with their schedule. Now every student has a class from their preference list. Once all students are enrolled in one class, we repeat this process for three times more with an extra check when considering each class: we have to check if the target class has time conflicts with the classes that the student is already enrolled. If there is a conflict, we will skip the target class. We keep assigning students with a random class in their preference lists in each of the 4 rolls and at the end we will assign students who haven't enrolled in 4 classes random classes that still have space available and don't have time conflicts with their current schedule. In this way, each student enrolls in precisely a total of four classes.

[1] *empty* is the set that we store all classes that could not be assigned to the previous time slots but with a higher popularity

1.2 Pseudocode

Algorithm 1 Registrar Work

```
Procedure could(c, r, t)
    // test if class c could fit into classroom r at time slot t
1   if time slot of the other class of p[c.professor] is not t then
2       return True
3   end
4   return False
end

// Fix the class schedule first
5 Procedure fix_schedule(C, R, T, P, S)
    // count each course's appearance times in S and assign them to popularity
6   foreach s ∈ S do
        // for each s, check all four preferred classes.  We are counting the frequency
           of classes in all the student's preference lists as popularity
7       C[s.pref_course].popularity ← C[s.pref_course].popularity + 1
8   end
    // sort class
9   Sort all c ∈ C by popularity in decreasing order  // sort room
10  Sort all r ∈ R by size in decreasing order
11  empty ← None  class_curr ← Null  set one_done of each cell([r_i, t_i] False;
12  foreach r ∈ R do
13      foreach t ∈ T do
            // skip the following for loop if the list empty is empty
14          for each c ∈ empty do
15              if could(empty, r, t) then
16                  c.classroom ← r  c.time ← t  one_done ← True  remove c from empty  break the
                     for(c ∈ empty) loop
17              end
18          end
19          while !one_done do
20              if class_curr == |C| + 1 then
21                  break the whole loop, i.e. outer for loop
22              end
23              if could(C[class_curr], r, t) then
24                  C[class_curr].classroom ← r  C[class_curr].time ← t  one_done ← True
25              end
26              else
27                  add C[class_curr] as the last element to empty  class_curr ← class_curr + 1
28              end
29          end
30      end
31  end
32 end
```

Algorithm 2 Continue registrar Work

```
// Fix the class schedule first
Procedure assign_courses(C, R, T, P, S)
    // Matching students with reasonable courses
    // input is a given student list, every student's preference list and the class
       schedule that we set before.  The order of students in the student list is
       random and the order of classes in each student's preference list is also random
    set avail of each student's preference list to True
     set mark of each class in each student's preference list to False
     for i = 1 to 4 do
        for each student s_j in the student list do
            if avail==True then
                for each class c_k do
                    if mark == False then
                        if c_k is not full and c_k fits into s_j's schedule then
                            assign c_k to s_j
                             set mark of c_k to True
                             if k==4 then
                                 set avail to False
                            end
                            break the for loop(for each class c_k)
                        end
                        else
                            set mark of c_k to True
                             if k==4 then
                                 set avail to False
                            end
                        end
                    end
                end
            end
        end
    end
    for each student s_m that hasn't been assigned 4 classes in total do
        assign a random class c_l that is not full and doesn't conflict with s_m's current schedule to s_m
    end
end
```

1.3 Time Complexity Analysis

Let's first look at the first part of the algorithm which is the course scheduling part. The for loop that computes the *popularity* for each class runs s times since it iterates through every student. For every student, we visit the 4 classes the student preferred. Hence the total cost of the for loop is $O(4s) \in O(s)$. We use merge sort for sorting the classes C. It will take $O(clogc)$ run time. Similarly, merge sort the classrooms R will cost us $O(rlogr)$. Then we have two nested for loops. The outer for loop runs r times, and the inner for loop runs t times. Inside the nested for loops, there is a for loop traversing every *class* in *empty*. When conflicting classes at time slot t is inserted into *empty*, they will definitely be assigned and removed from *empty* at following time slots because they could only have conflicts with classes at time slot t. Therefore, the max number of classes in *empty* is the max number of consecutively having

conflicts at one time slot. Consider any time slot t, there can be at most $r-1$ classes already assigned to t. If the $professor$ of every $class$ we get from C is already teaching a class at this time slot t, we will at most encounter $r-1$ consecutive conflicts. Therefore, there could be at most $r-1$ classes in $empty$ at any time, so the for loop at most iterates $r-1$ times. Hence the run time of this for loop is $O(r-1) \in O(r)$. The while loop inside the nested for loops traverses C until it finds a $class$ that can fit into the current cell. As we discussed above, we will at most encounter $r-1$ consecutive conflicts and finally get to a fitting class at the r^{th} iteration. Hence this while loop runs at most r times and costs $O(r)$. Therefore, the total run time of the first part is $T = s+clogc+rlogr+r\times t\times(r+r) = r^2t+s+clogc \in O(r^2t+s+clogc)$.

Then we look at the second part of the algorithm which is the course assignment for students. Initializing $avail$ for each student takes $O(s)$ because there are s students. Similarly, setting $mark$ for each class in each student's preference list takes $O(4s) \in O(s)$ because each student has 4 classes in their preference list. Then we have the nested for loops. Inside the nested for loops, we cumulatively traverse through every class in the preference list of every student. Since there are 4 classes for a total of s students, this takes $O(4s) \in O(s)$. At the end of the algorithm, some students only have less than 4 classes, and we assign them other classes so that they have a total of 4 classes. Since we have to traverse the list of students of find students that still need classes, the for loop also takes $O(s)$. The total run time of this part is $O(s)$.

If we combine the run time of two parts, the run time of our whole algorithm is $T = r^2t+s+clogc+s = r^2t+2s+clogc \in O(r^2t+s+clogc)$.

In order to run the algorithm in the expected run time, the following operations need to be done in $O(1)$:
1. assign and get any information ($professor$, $classroom$, $popularity$, $timeslot$) of a given class c
2. given $classc$, get the other class taught by the same professor
3. get the preference list of a given student s
4. get the $size$ of a $classroom$
5. get the capacity of a $class$
6. insert/remove a $class$ to/from $empty$
7. set and get $avail$ of each student
8. set and get $mark$ of each class in the preference list of each student
9. check whether a class is fully enrolled
10. check whether a student has enough classes
11. check whether a class fits into a student's schedule

We also simplified the time complexity of the algorithm. The runtime we discussed above is the worst case run time because we considered the number of collisions we can encounter at most in the scheduling process. Yet in real cases, collisions happen very seldom, so the worst case run time is way higher than most real run time of the algorithm. Thus we calculated an average run time which neglects the collisions. It is $rt+s+clogc+rlogr \in O(s+clogc)$.

We generated different sizes of datasets and then measure the time in scatterplot graphs to see the growth pattern and did experimental verification of theoretical big-O time analysis results.

r: The different room numbers that we tested are (50, 60, 80, 100, 120, 160, 200, 250).
The statistics that we got from running the regression model was:
Multiple R-squared: 0.08325, Adjusted R-squared: -0.06955
So the result is align with our hypothesis that r is not dominant in big-O.

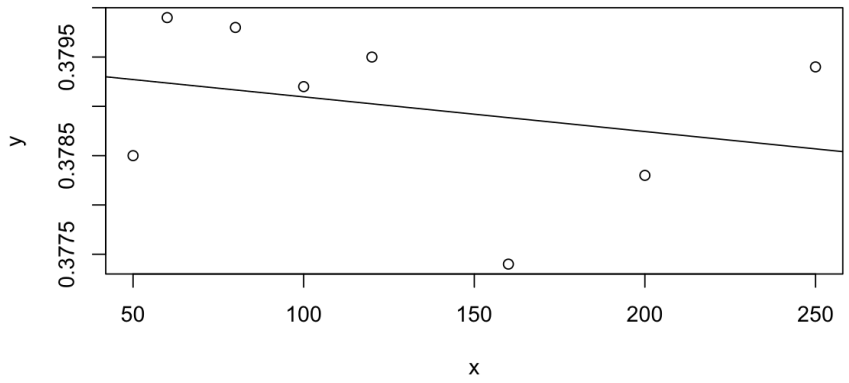

Figure 1: Scatterplot showing the relationship between number of rooms (x-axis) and total time complexity (y-axis)

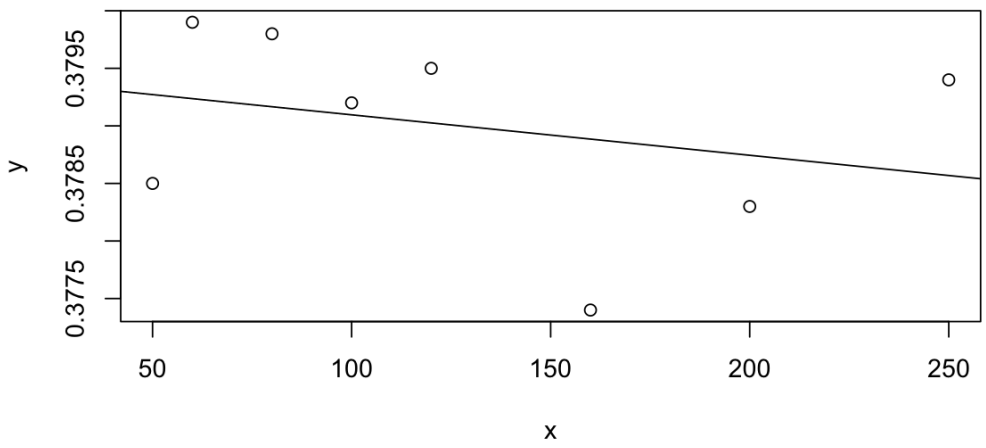

Figure 2: Scatterplot showing the relationship between number of time slots (x-axis) and total time complexity (y-axis)

t: The different time slot numbers that we tested are (50, 60, 80, 100, 120, 160, 200, 250).
The statistics that we got from running the regression model was:
Multiple R-squared: 0.08325, Adjusted R-squared: -0.06955
So the result is align with our hypothesis that t is not dominant in big-O.

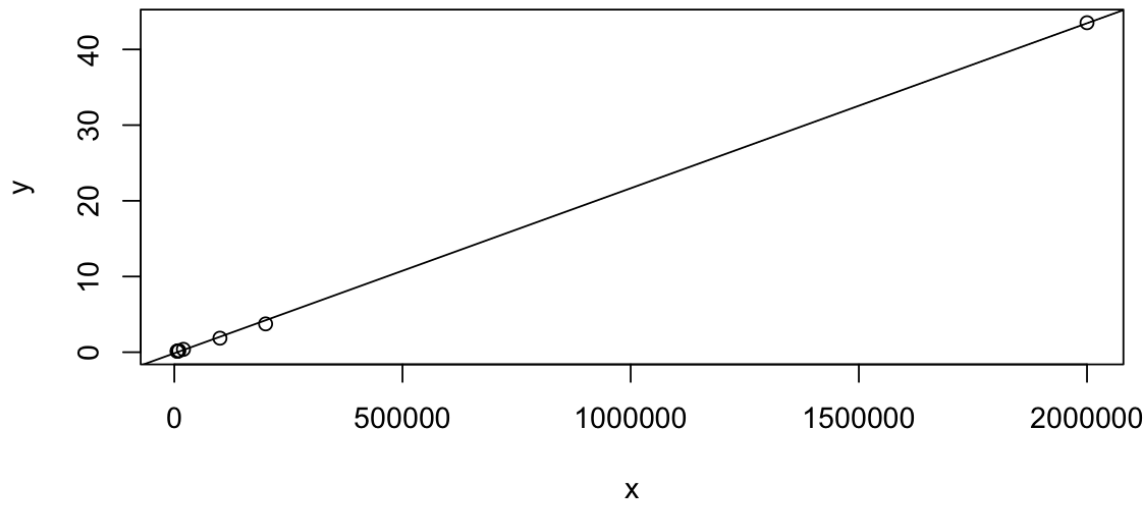

Figure 3: Scatterplot showing the relationship between number of students (x-axis) and total time complexity (y-axis)

s: The different student numbers that we tested are (6000, 7000, 8000, 10000, 20000, 100000, 200000, 2000000).
The statistics that we got from running the regression model was:
Multiple R-squared: 0.9998, Adjusted R-squared: 0.9998
So the result verifies our hypothesis that s is dominant in big-O.

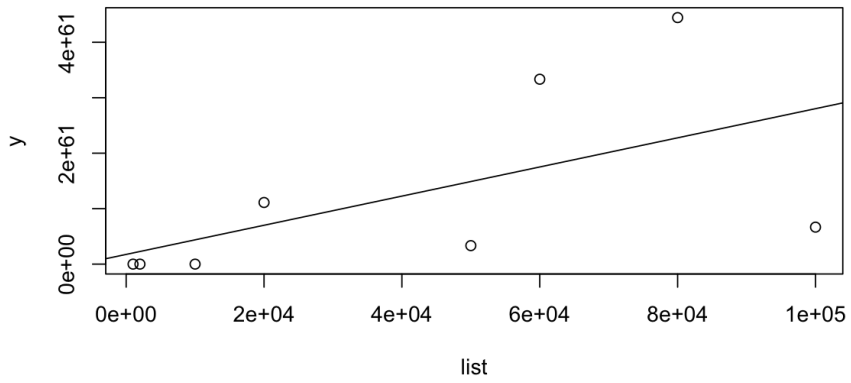

Figure 4: Scatterplot showing the relationship between number of classes (x-axis) and total time complexity (y-axis)

c: The different class numbers that we tested are (1000, 2000, 10000, 20000, 50000, 60000, 80000, 100000).
The statistics that we got from running the regression model was:
Multiple R-squared: 0.3368, Adjusted R-squared: 0.2263
So the result verifies our hypothesis that c is not dominant in big-O.

We also ran regression model on clogc and time complexity.

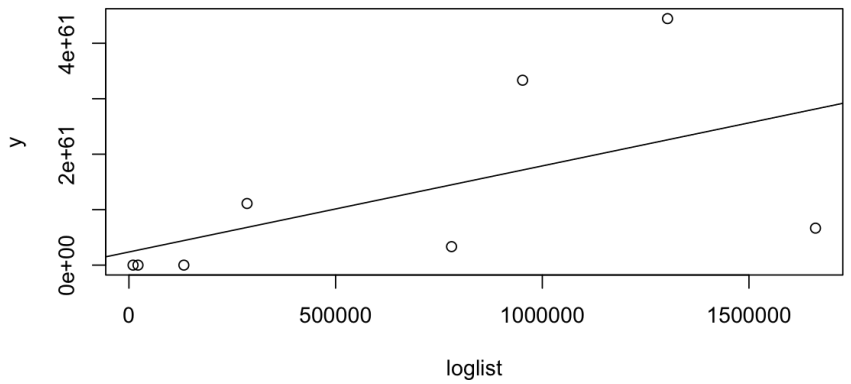

Figure 5: Scatterplot showing the relationship between clogc (x-axis) and total time complexity (y-axis)

The statistics that we got from running the regression model was:
Multiple R-squared: 0.326, Adjusted R-squared: 0.2136
We expected that the function of clogc and time complexity would be linear but the result fails to show that. Our guess is that it's because clogc is not dominant in big-O compared with s.
In conclusion, our theoretical big-O time analysis results are verified and we can further simplify O(s + clogc) to be O(s).

1.3.1 Data Structures

For the information of each class, we use separate dictionaries for every information of a class. The keys of the dictionaries are the classes, and the keys are respectively each information we need to store. When reading the input, we store the $professor$ of each class in a dictionary. When we computed the *popularity* of the classes, we store it in another dictionary. Then, after deciding the *classroom* and *timeslot* of a class, we store them in the other two dictionaries. These will allow (1) to be done in $O(1)$. Building each of these dictionaries takes linear run time.

We use another dictionary to store what two classes does each $professor$ teaches. The key is the $professor$ and the value is a 1×2 array that stores the two classes. We get the $professor$ of a *classc* by $O(1)$ as discussed in last paragraph. Then we use the dictionary we just built in this paragraph to get the two classes of this $professor$ and target at one of them also by $O(1)$. This together allows (2) to be done in $O(1)$. Building this dictionary takes $O(p)$.

Another dictionary is used to store the preference list of each student. The keys will be the students and the values are 1×4 arrays storing the 4 preferred classes. This allows (3) to be done in $O(1)$. Building this dictionary takes $O(s)$.

For the sizes of the classrooms, we use another dictionary where the keys are the classrooms and the values are the sizes. This allows (4) to be done in $O(1)$. Building this dictionary takes $O(r)$. Since the capacity of a class is the same with the size of the classroom of the class, we get it by first getting the classroom of a class with $O(1)$ and then getting the size of the classroom with $O(1)$. Hence allowing (5) to be done in $O(1)$.

For *empty* which stores the conflicting classes that are not assigned yet, we use a doubly linked list. Inserting to the end of a linked list takes $O(1)$. Removing from the head also takes $O(1)$. Since we always insert to the end and remove from the head, (6) will be done in $O(1)$. Building it also only takes $O(1)$ because we only have to initialize the head and tail both to null.

We will use another dictionary to store the *avail* of each student. The keys will be the students and the value is true or false. This allows (7) to be done in $O(1)$. Building such dictionary takes $O(s)$.

We also use a dictionary to store $mark$. The keys will be the students and the values are 1×4 arrays storing $mark$, which is either true or false. This allows (8) to be done in $O(1)$. Building this dictionary takes $O(s)$.

We use another dictionary to store the available seats in each class. The keys are the classes and the values are integers represent the available seats. After we scheduled all classes, we will initialize each value to the size of the classroom of the class. Then, when a student is assigned to this class, we decre-

ment the value by 1. This allows (9) to be done in $O(1)$ by simply checking if the available seats is 0. Building such dictionary takes $O(c)$.

Another dictionary is used to store the number of classes a student has successfully enrolled. The keys are the students and the values are integers from 0 to 4. Every time a student is assigned a class, we increment the value of that student by 1. This allows (10) to be done in $O(1)$. Building such dictionary takes $O(s)$.

We will use another dictionary to store the classes a student has been assigned to. The keys are the students and the values are 1×4 arrays. We initialize every element in the array to null, and update it when a class is assigned. This allows (1) to be done in $O(1)$ because we can simply traverse through the class array of a student and compare the time slots. If no time slot is the same, the class can fit into the student's schedule. Building such dictionary takes $O(s)$.

With the above data structures, the course scheduling algorithm takes $O(r^2t + s + clogc)$ to run because all the other operations take $O(1)$ to finish. The run time of building all the data structures is linear.

Thus, the time complexity of the course scheduling algorithm is $O(r^2t + s + clogc)$.

2 Proof of Correctness

2.0.1 Proof of termination

The termination of *fix_schedule* For the procedure *fix_schedule*, there is a small loop starting from line *8*, which will check all 4 preferred classes for all students and adjust the *popularity* score for the class. It terminates after $4|S| = 4s$ times. The sorting in line *11* takes $c \log c$ times to finish where $c = |C|$, and the sorting in line *12* takes $r \log r$ times to finish where $r = |R|$. Then, for the nested loop, this stops after finishing assigning the room and time for each class in C, which needs $c = |C|$ times iterations. Thus, the procedure *fix_schedule* iterates.

The termination of *assign_course* All the for loops in the algorithm will ultimately go to an end. So the algorithm will terminate.

2.0.2 Proof of correctness

In order to proof the correctness of this algorithm, we need to prove the following to show the schedule is valid:

1. No more than one class will be scheduled in the same room during one time slot.
2. No teacher will teach more than one class during one time slot.
3. No student will be scheduled for more than one class during one time slot.

Also, we can also show the optimality of the student-class matching given the fixed schedule.

No more than one class will be scheduled in the same room in one time slot We're going to prove this by directly showing for a given time in each room, there will not be more than one class scheduled. For a given time in each room, there are three cases:

1. The class from the *empty* list will be scheduled. Then the for loop starting at line *18* will break since line *24*. And since we set the boolean *one_done* to be *True*, the loop will go to the next time slot for the current classroom if the current time slot is not the last one of the week **or** go to the first time slot for the next classroom otherwise without accessing the while loop starting from line *27* (*).
2. The class $C[curr\_class]$ will be scheduled. This happens only if *empty* is empty. Thus, line *34* will set the boolean *one_done* to be *True*, which ensures (*).
3. All class in C has already been scheduled, i.e. $class\_curr == |C| + 1$. Then, the whole nested loop will break due to line *29*.

Because of the property of loop, after skipping a given time in each room, it will not be accessed again. Therefore, for a given time in each room, there will not be more than one class scheduled. Then, no more than one class will be scheduled in the same room at the given time.

No teacher will teach more than one class during one time slot. If the teacher of the class has already registered for their other course at the current time slot, then the procedure $could(c, r, t)$ will return False. For all cases we might consider assigning the class to a specific classroom and time slot, we all check $could(c, r, t)$ (line *19* and line *31*). Thus, no teacher will teach more than one class at a given time.

No student will be scheduled for more than one class during one time slot. Based on our algorithm, we check whether a class has conflicts with a student's current schedule before assigning the class to that student. So it's guaranteed that no student will be scheduled for more than one class during one time slot.

Given the fixed schedule, the student-class matching is optimal. During the process of setting class schedules, we assign a more popular class a room as large as possible, which makes sure that as many students as possible will be able to enroll in this class. Then during the process of assigning students classes, we assign each student a class from their preference lists during each of the 4 rounds as long as the class is not full and can fit into the student's current schedule. After the 4 rounds, we just fill the left space in students' schedule with random classes that are not full and can fit into the student's schedule. Therefore, the student-class matching is optimal.

2.1 Solution Quality Analysis

In order to evaluate the optimality and runtime of our algorithm, we generated combination for the input size, $S = \{100, 200, 400, 600, 800, 1000, 2000, 3000\}$, $C = \{20, 40, 80, 100, 200\}$, $R = \{5, 10, 20, 30\}$, and $T = \{5, 10, 20, 30\}$. For each considered combination of s, c, r, t, in order to reduce the influence of random, high variance input, we generated 10 random data sets with exactly same sizes of input, and then take average. As shown in Figure , the metrics we considered here are:

1. *Experiment*: This is the preference score refers to the number of classes students get assigned to eventually among all classes they preferred $\frac{experimental\ score}{Best}$ with Best $= 4 * s$.
2. *% Optimality*: This evaluates how good the preference score is compared with the number of students, which is calculated by $\frac{\text{Experiemental}}{4*s}$.
3. *Time(sec)*: This metric helps us to analyze the effects of the size of different datasets for our overall runtime, where we record by real runtime, instead of system and user runtime.

Notably, from analyzing the runtime for computing and their input datasets' various size, we discover the influence of the size of $|Students|$ and the size of $|Classes|$. We can check the combination no. 15

and no. 17 shown above, where the sizes of other input datasets, $|Classes|$, $|Times|$, and $|Rooms|$ stay consistent. But the size of $|Students|$ ahs increased from 1000 to 3000, which caused a noticible change for the run time from 0.0341 to 0.0497. A similar thing could also be found by comparing no. 13 and no. 15, where the size of $|Classes|$ changes from 100 to 200, increasing the runtime from 0.0327 to 0.0341. Other than those two input datasets, we don't find other dataset whose input size would heavily influence the runtime. This is similar to what our modified runtime analysis shows.

3 Handling Real Data Constraints

After observing real data of BMC and HC, we found that we need make modifications to handle the constraints and complexities of real-world data, ensuring the algorithm aligns with practical application scenarios.

3.1 classes with no professors assigned

In the dataset, we encountered a situation where not all classes have information about their assigned professors. Without the professor's details, it would be impossible to ensure that there are no conflicts. Thus we decided to exclude these classes from the scheduling process. By ignoring such incomplete data, we maintain the integrity and feasibility of the scheduling algorithm, ensuring it functions accurately for the remaining classes with sufficient information.

3.2 classes with no professors assigned

In the basic algorithm, we initially assumed that every student's preference list consisted of exactly four classes. However, upon analyzing the real data, we discovered that many students prefer either more or fewer than four classes. To accommodate this variability, we modified the process that handles students' preference lists. The updated algorithm now dynamically adapts to the actual number of preferred classes in each student's list, ensuring flexibility and compatibility with real data. This adjustment enhances the robustness of the algorithm and allows it to handle diverse and realistic scenarios effectively.

3.3 deal with overlapping time slots

In the basic algorithm, we used simplified, non-overlapping time slots labeled as numbers. However, in real data, time slots are defined by exact time periods, and some of them overlap. This introduced a new challenge: a classroom cannot be used in overlapping time slots simultaneously. To address this, we extended the algorithm to identify which of the time slots can be used without conflicts. The first part of our extensions focuses on maximizing the number of non-overlapping time slots utilized, ensuring that we can schedule as many classes as possible while adhering to real-world constraints.

Extensions based on Timeslots

Recall that our algorithm assign sorted classes into classrooms (capacities from large to small), assigning the next classrooms until fulfilling every time slot, where the conflict (not able to assign exactly based on the sorting) of assigning class might happen due to the professor's another class's time slot. Then, when we assigning classes from students' preference list to students, the conflict might also happen due to the time slot conflict among student's assigned (preferred) classes. Intuitively, we think the arrangement

of time slot is important and might be helpful for overall fit. Here we will propose three modifications based on the time sloys.

4 Timeslots: Greedy

4.1 Description

When we first worked with Bryn Mawr and Haverford real data, we noticed the overlap of time slots, which is the case that we did not cover when we worked on basic dataset. For this approach, our modification for time slot assignment is assigning the next non-conflict time slot starting from the first time slot. Then, we will use the same time slot sets for all classroom and follow the same algorithm.

4.2 Pseudocode

Algorithm 3 Time slot - greedy

```
Procedure find_timeslot(T)
    // This part demonstrate how the greedy approach works
59  T' = [t_0]
60  rank time slots by starting time
61  for t_i ∈ T but not t_0 do
62      if t_i not conflict with the last element in t_0 then
63          add t_i into T'
64      end
65  end
66  return T'
end
```

4.3 Performance and Time complexity

The greedy algorithm did not introduce a noticeable change of the runtime, since the checking of conflict or not is constant. The performance of greedy have a great balance between run time and score. It performance better than the dynamic program for most time, and worse than IS and annealing, but those often take much longer time.

5 Timeslots: Dynamic Programming

5.1 Description

We only modified the time slots set and all the other things stayed the same as the greedy algorithm. We performed dynamic programming on all the time slots that are given and tried to achieve the largest set of non-overlapping time slots. Since it's hard to compare time slots on different days, we splitted them into MWF and TTH categories. MWF category contains time slots on MWF, MW and F, and M/W/F. TTH category contains time slots on TTH and T/TH. We performed dynamic programming on each set inside MWF and TTH categories and picked the largest set inside each and combined them together. Our dynamic programming is a modified version of weighted interval scheduling. In each set, we sorted all the time slots by end time and set each time slot's weight to 1. We also had pred[] for each time slot to trace back and used gen-schedule() to access the largest set is composed of which time slots. After processing time slots, we followed the same algorithm as greedy and designed schedule for all the

classes and assigned classes to each student based on their preference lists.

5.2 Performance and Time complexity

After implementing dynamic programming, we compared DP with greedy and found that in terms of performance, though DP got a higher score than greedy for a few semesters, generally greedy had higher scores than greedy for most semesters. In terms of time complexity, DP is slower than greedy because DP has an extra weighted-interval scheduling part, which took longer to process than greedy.

6 Timeslots: Independent Set

6.1 Description

To solve the problem of determining the maximum number of non-overlapping time slots, we transformed it into a graph problem. In this graph, each time slot is represented as a vertex, and edges are added between vertices if their corresponding time slots conflict. To construct the graph, we traversed through all pairs of time slots, checking for conflicts; if a conflict existed, we connected the two corresponding vertices with an edge. Once the graph was created, we applied an algorithm to find the large independent set within the graph. This independent set represents a group of vertices (time slots) that are not connected by any edges, meaning the corresponding time slots do not overlap. The size of this independent set corresponds to the maximum number of non-overlapping time slots that can be scheduled, providing a solution to the problem.

6.2 Performance and Time Complexity

This approach achieves a significantly higher score compared to the previous algorithms, even when provided with only a small subset of all available time slots as input. Despite the limited input, it consistently outperforms other algorithms in terms of scheduling effectiveness. However, the time complexity of this method is very high, as finding the largest independent set is an NP-complete problem. This computational difficulty is also the reason why it achieves such a high score, as it explores a more optimal solution space compared to simpler methods. Since the course scheduling problem itself is NP-complete, relying on greedy algorithms or dynamic programming cannot guarantee an optimal solution, whereas this approach leverages the inherent complexity to achieve superior results.

Extensions based on Classes

7 Popularity/Conflict Based

Our basic algorithm is popularity-based but we found that the most popular class might not be the one has most conflicts with other classes, so we decided to also do conflict-based algorithm.

7.1 Description

We built a matrix with row being each class and column also being each class. One cell [c_i, c_j] represents the times of class c_i and class c_j are in one student's preference list at the same time, i.e. the frequency of c_i and c_j being conflicted with each other. Then we went through each student's preference list and updated the counts in each cell of this matrix. We had a conflict list and kept adding two classes in the cell [c_i, c_j] with max value in the matrix and set the max value to a negative value. Then we designed

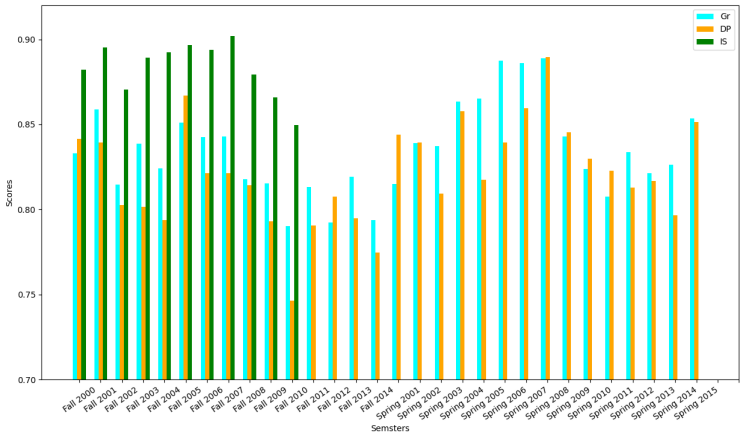

Figure 6: performance compare - bar chart

schedule for all the classes in the order of the conflict list and all the other things stayed the same as our basic algorithm.

7.2 Comparison between Popularity and Conflict-based Algorithm

We tested popularity and conflict-based algorithm on Fall2004. The popularity-based list(contained all the classes sorted in the decreasing order of popularity) had 265 classes while the conflict-based list(contained classes sorted in the decreasing order of conflict frequency) had 264 classes. Our guess was that a class only appeared in one student's preference list and this student only picked this class. Popularity-based-only algorithm got score 0.82419 and conflict-based-only algorithm got score 0.82150. Then we appended the left class to conflict list, i.e. the list now had all the 265 classes, and the score was 0.82150 as well.
Then we combined the popularity list with the conflict list and produced a new list and designed schedule based on the new list. We tested different proportions of rankings in the popularity list and the conflict list and added up the two rankings in the two lists and produced a new ranking for each class and sorted them in decreasing order in the new list. The following formulas are in the format of proportion1 * ranking in popularity list + proportion2 * ranking in conflict list = performance with proportion1 + proportion2 = 1:
Fall2004
1.0pop + 0.0con = 0.82419 (same as popularity-based only)
0.99pop + 0.01con = 0.82365
0.95pop + 0.05con = 0.82553
0.93pop + 0.07con = 0.82607
0.90pop + 0.10con = 0.82957
0.89pop + 0.11con = 0.83602
0.88pop + 0.12con = 0.82392
0.85pop + 0.15con = 0.82930
0.75pop + 0.25con = 0.82796
0.70pop + 0.30con = 0.82607
0.60pop + 0.40con = 0.84408
0.50pop + 0.50con = 0.82150 (same as conflict-based only and append the last class to conflict list)
0.40pop + 0.60con = 0.83037

0.30pop + 0.70con = 0.83387
0.25pop + 0.75con = 0.84408
0.20pop + 0.80con = 0.82741
0.10pop + 0.90con = 0.82661
We observed that there are some trends in scores of Fall2004 as the proportion changes and we also tested the proportion idea on Spring2004:
Spring2004
0.9pop + 0.1con = 0.83356
0.8pop + 0.2con = 0.82297
0.7pop + 0.3con = 0.83611
0.6pop + 0.4con = 0.82042
0.5pop + 0.5con = same as conflict-based
0.4pop + 0.6con = 0.83120
0.3pop + 0.7con = 0.83258
0.25pop + 0.72con = 0.82630
0.2pop + 0.8con = 0.83003
0.1pop + 0.9con = 0.83258
We can see that trends in the scores of different datasets are different and we are not able to reach a general conclusion on trends for all datasets, i.e. when the highest score would be produced. But when generating schedule for one semester, we can use different proportions to test the highest score, which might be higher than both popularity-based only and conflict-based only.

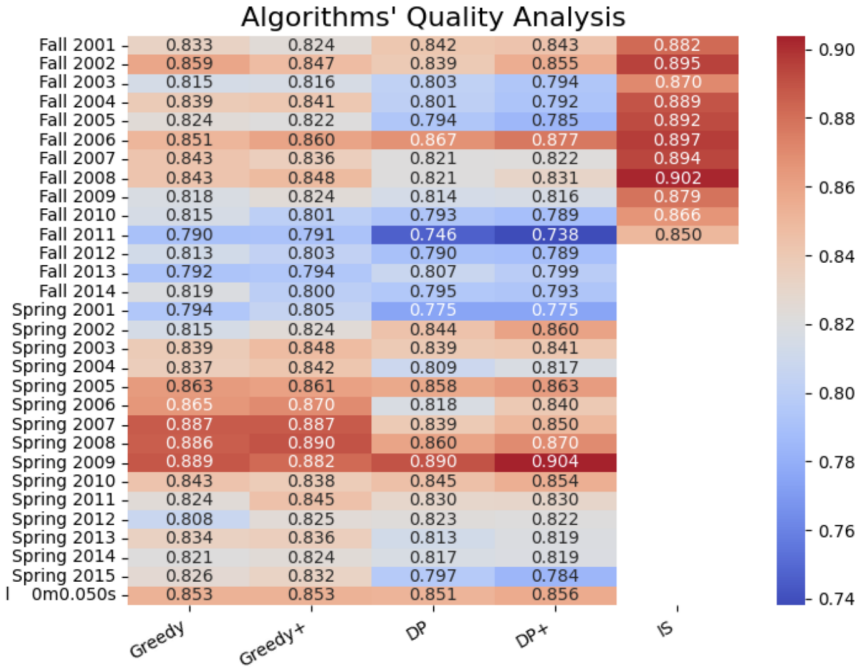

Figure 7: performance compare - heatmap

Extensions based on Classrooms

8 Departmental Classroom Allocation

8.1 Description

To handle the concept of departmental classrooms, we utilized the input data, which specifies the classrooms that each class can use. We created a class called "Department", which includes a field named "classrooms" to store all the classrooms available for use by that particular department. This information is read from the input and stored appropriately. Next, we modified the function "schedule_classes" to incorporate this constraint. In the updated implementation, we continue using a matrix to manage the scheduling process. However, before scheduling any class, we added an additional check to ensure that the selected classroom is in the department's list of allowed classrooms. If the classroom is not in the list, the class is not scheduled in that room. This modification enforces the departmental constraints while maintaining the integrity of the scheduling algorithm.

8.2 Performance and Time Complexity

Implementing the departmental classroom constraint actually led to a reduction in performance. Initially, the algorithm prioritized assigning the most popular classes to classrooms with the largest capacity, allowing more students to enroll in these high-demand courses. However, with the new constraint, each class is restricted to a fixed set of classrooms specific to its department. This can result in popular classes being assigned to classrooms with smaller capacities if the department's available classrooms lack large spaces. As a result, these high-demand classes are unable to accommodate as many students as before, leading to a decrease in overall scheduling efficiency and capacity utilization.

To validate this inference, we conducted a test comparing the initial algorithm with the updated departmental classroom allocation. In the initial algorithm, the five most popular classes were all assigned to classrooms with a capacity of 119 students. This resulted in a total availability of $5 \times 119 = 595$ seats for these classes. However, with the departmental classroom constraint in place, the top five classes were restricted to smaller classrooms, reducing their total availability to only 391 seats. This significant reduction in availability demonstrates how the departmental classroom allocation limits the scheduling flexibility and directly impacts the score.

The implementation of the departmental classroom allocation does not significantly affect the time complexity of the algorithm. To store the classrooms for each department, we only need to traverse the list of classes and store the associated classrooms in a list. Importantly, this operation is not repeated for every class, it is performed only once for each department. Since all classes within the same department share the same list of classrooms, we check whether the classrooms for a department have already been processed before performing the operation. As a result, the number of times we need to read and store the classrooms is proportional to the number of departments, making this step efficient and minimally impactful on the overall complexity.

Extensions based on assigning

9 Simulated Annealing

9.1 Description

As we discussed before, the conflict (non 100% classes matching) might also happen during the process of assigning classes to students, which gives us information about the efficiency of the class schedule. For this simulated annealing algorithm, we will make modification to the schedule after non modified version. We will first run a class assigning, then collecting all classes that running into the scenario that a student preferred but cannot be assigned to. Without loss of generality, we will take care the time slot conflict but not the case of exceeding the maximum availability.

Then, we will run simulated annealing within this set: we will randomly switch any two classes (switch both classrooms and time slots). Then, display the assigning process again, we will include this modification if this results in a higher score. We modify this based on the greedy approach.

9.2 Pseudocode

Algorithm 4 Simulated_annealing

```
Procedure Simulated_annealing(C, R, T, P, S)
    conflict, score = assign_courses(C, R, T, P, S)
    // conflict as the set of classes that has conflict to each other but the same
       student preferred
67  i = 0
68  while i ≤ 1000 do
        // with limited times of switching
69      a, b = random index in conflict
70      index_a, index_b = conflict[a], conflict[b]
        // This is the index in the C
71      C[index_a], C[index_b] = b, a
72      conflict_1, score_1 = assign_courses(C_1, R, T, P, S)
73      if score_1 > score then
74          C = C_1
75      end
76  end
77  return C
end
```

9.3 Performance and Time complexity

Since the way our algorithm design, the performance of this will not be worse than greedy. We have ran the experiment based on the dataset of Fall 2004, ranging from 83.494624% to 84.193548% with limited the number of switching to 1000, where the greedy has 82.419355%. The runtime is reasonable with the number of switching limited to 1000, but since this is a NPC algorithm, it's hard to analysis the time complexity.

Recommendations

1. According to the experiment of Departmental Classroom Allocation, we noticed a really low fit for students' preferences. This is caused by the limitation of the classroom departmental. Therefore, we might expand the departmental limitations to the whole building, i.e. all computer science courses can be taken not only those several classrooms in Park, but also all classrooms in Park. This would significantly increase the "score" (fit) of the schedule.
2. We recommend using IS and annealing algorithm for assigning classes to classrooms and time slots, which have a better performance overall.