CS340 - Project Description and Analysis

Yutong Li, Tianming Xu, Jiaping Wang

October 5, 2017

1 Description

1.1 Notation

Let S be the set of students, with $|S| = n$. Each student will have a list of class requests, so let l_{s_i} to represent the i^{th} student's list of class requests. We denote the set of teachers by P and its order by m. Let C be the set of classes, and so we have $|C| = 2m$ classes. Denote the set of classrooms by R, with $|R| = y$, and the set of time slots as T, with $|T| = x$.

1.2 Algorithm Description

Given S, P, C, T, we determine a class schedule with following rule: schedule as many students as possible into each class in decreasing order of the class' popularity.

Start off by traversing the request lists for each $s \in S$ and creating a list K storing the number of students who signed up for each class, indexed by classes. Sort K by decreasing order of popularity. Then, sort the classrooms in decreasing order of capacity. For each teacher $p \in P$, construct a list to store the time period during which he or she is unavailable (none at this point). Initialize the variable, *skippedSlot*, to none.

Then, we can begin pairing classes with classrooms and time periods. Starting with the most popular class c_1, we try scheduling this class in the biggest classroom r_1 during the first time slot t_1. If r_1 is occupied during t_1, we move on to the next available time slot. When all time slots for a classroom has class assignments, we move on to the next largest classroom. Once we arrive at an empty classroom-time pair, find the teacher p_1 who teaches c_1 and check for time he/she is unavailable. If p_1 has no conflict during this time, put the class-classroom-time combination into the final schedule. Otherwise, set *skippedSlot* to the assignment (r_1, t_1) and move on to the next time slot. Once this class is scheduled, mark both the teacher and classroom as unavailable during this time period. Repeat this process

for the next most popular class, except that once *skippedSlot* is non-empty, we can assign the class to *skippedSlot* directly and set it to empty again. Otherwise, try scheduling the class to the next largest classroom at its earliest time slot, checking for conflicts with teacher and classroom. Keep doing so until either all classrooms are filled at all times, or all $2m$ classes has been scheduled.

2 Time Analysis

2.1 Data Structure

For each student, the class requests can be stored as a linked list, so accessing a student's class takes $O(1)$. Hence, it takes $O(n)$ to traverse all class requests and initialize the array K. Sorting K with the standard quicksort requires $O(n \log n)$.

To store information about whether a classroom is taken at a certain time, we can create an array of size x for each $r_i \in R$, where the j-th value in the array indicates whether r_i is available during time t_j. Constructing the array takes $O(xy)$ and access time for each value is $O(1)$.

3 Proof of Correctness

Proof of Termination.

Proof of Validity.

4 Discussion

Well, it's not that intuitive to come up with this algorithm. First, we brainstormed some algorithms, dynamic programming, graph algorithm, recursion, naive algorithm, and greedy algorithm. It's pretty clear that dynamic programming and recursion are not suitable for this problem, since there's nothing to conquer and divide. Even if it does have something we can recursively solve, for this particular problem, we cannot afford the cost of recursion since the number of students is fairly large and is predicable our computer will run out of stack if we insist to do some recursions. Also, it's hard to construct for this problem. Suppose we let students become vertices, then there're still a bunch of limits we need to consider, and thus the edges are hard to defined. Then we come up with greedy algorithm

which contains no backtracking which implies that there might exist a $O(nlogn)$ or less greedy algorithm that can solve the problem.

One of the most important complications we met in designing algorithm is to be "greedy" on which aspect. According to the description of the problem, the performance of the algorithm depends on how many classes students can take considering their preference lists. Thus, the intuition is to compute out for each class how many students to take, then this become the first part of our algorithm, looking into students' preference lists and calculate the popularity of each class.

Then we decided to first guarantee classes with highest popularity, thus there comes the sorting part of the algorithm. Naturally, most popular class(with the most students wanting to take) should occupy the largest room. And in order to avoid collisions, the second popular class pairs up with the second largest room, so on so forth. Then there came our first corner case, what if the number of students wanting to take the most popular class exceeds the size of the largest room? Luckily, in the FAQ part we get clarifications that no class should have a second section.

After dealing with rooms, we should look into time slots. The main idea is to reduce time collisions between those popular classes. Then we modified our previous statement about room: rather than pair the largest room with the most popular, the second largest with the second most popular, we can pair up the largest room with all those popular classes, and arrange them in different time slots. Under such a circumstance, we can put each room into full use.

Considering limit on teachers then. What if in some cases, in the pairing process we find that a teacher might have time conflicts? In other words, a teacher p_i teaches two classes c_i, c_j and c_j is less popular. Though taught in different rooms and have different popularity, c_i, c_j somehow will be arranged into the same time slots x. One of the choice is to skip c_j, but this is not optimal. Instead, we simply move one time slots downward, and set c_j, to that slot, and keep track of the time slots we skipped. Consequently, the room c_j was in won't be empty at time slot x, since we will set the next class to x.

For this problem, the most obvious difficulty is to deal with all these restrictions and solve collisions. It is also predictable that there're lots of corner cases we need to deal with when implementing the algorithm.