CS340 - Project Description and Analysis

Yutong Li, Tianming Xu, Jiaping Wang

October 10, 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, R, 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 sorting the classes in decreasing order of popularity, which is defined by the number of students who signed up for each class. Sort the classrooms in decreasing order of capacity.

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. Find the teacher p_1 who teaches c_1. If p_1 has no conflict during this time, put the class-room-time combination into the final schedule. Otherwise, add the assignment (r_1, t_1) as a skipped slot 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 in the next time slot, except that if there are previously skipped slots, we try scheduling the class to the first skipped slot first. Otherwise, try scheduling the class in the next largest classroom during its earliest time slot, checking for conflicts with teacher. When all time slots for a classroom has class assignments, we move on to the next largest classroom. Keep doing so until either all

classrooms are filled at all times, or all $2m$ classes has been scheduled.

2 Pseudocode

```
1  Function Schedule(S, P, C, R, T)
2      for all c ∈ C do
3          Count the number of students who signed up for c
4      end
5      Sort C in decreasing order of popularity.
6      Sort R in decreasing order of capacity.
7      k, j ← 0
8      skippedSlots ← empty
9      while j < y and k < 2m do
10         Choose classroom R_j
11         i ← 0
12         while i < x and k < 2m do
13             Choose time T_i
14             if skippedSlots is empty then
15                 if teacher P_{C_k} who teaches C_k is available in T_i then
16                     Schedule[R_j][T_i] ← C_k
17                     i ← i + 1, k ← k + 1
18                     append T_i to occupied[P_{C_k}]
19                 else
20                     append (R_j, T_i) to skippedSlots
21                     i ← i + 1
22             else
23                 pop the first element in skippedSlots
24                 if teacher P_{C_k} has a conflict with the skipped slot then
25                     push the slot back to skippedSlots
26                     repeat lines 18 − 24
27                 else
28                     Schedule[skippedSlot] ← C_k
29                     k ← k + 1
30         end
31         j ← j + 1
32     end
33     return Schedule
```

3 Time Analysis

3.1 Data Structure

The class requests of each student can be stored as a linked list. Constructing the lists in S takes $O(n)$ and accessing a student's class choices takes $O(1)$. Furthermore, it takes $O(n)$ to traverse all class requests and initialize the array K. Sorting K with the standard quicksort requires $O(n \log n)$.

The instructor information P for classes can be stored in an array indexed by the classes. Thus initialization of P takes $O(m)$, and retrieving the teacher of a given class C_k takes $O(1)$. Similarly, classroom capacities can be stored in an array. Construction of R takes $O(y)$ and sorting R takes $O(y \log y)$.

Information about teachers' assignments can be stored in a dictionary indexed by the teachers. Each time a class is scheduled, append the time t_i to the teacher's assignments. Because each teacher teaches at most two classes, the size of a teacher's assignments cannot exceed 1. Thus, checking whether P_{C_k} has conflict during time T_i takes $O(1)$. Construction of the dictionary takes $O(m)$.

The skipped slots due to teachers' time conflicts can be stored in a FIFO queue. Therefore, pushing a slot into the queue, checking if it's empty, as well as popping its first element can be done in constant time. Since each teacher teaches at most two classes, each teacher can have at most 1 conflict and so construction requires $O(m)$.

To store the final schedule, we can create a dictionary of dictionaries. The outer dictionary has $r_j \in R$ as keys, and for each r_j we construct a dictionary of size x, where the i-th value in the array indicates whether r_j is available during time t_i. Constructing the nested dictionary takes $O(xy)$. Accessing or changing each value can be done in $O(1)$.

3.2 Time Complexity

Using the data structures described above, we know that, in total, initialization requires

$$n + n + n \log n + y + y \log y + m + m + xy = O(n \log n + y \log y + m + xy).$$

By construction of the algorithm, the *while* loop terminates when all classrooms are $filled$ during all time periods or $2m$ classes are scheduled. Therefore, the nested *while* loops runs $\min(xy, 2m)$ times. Therefore, the overall time complexity of the algorithm is

$$\begin{aligned} T(n) &= O(n \log n + y \log y + m + xy) + O(\min(xy, 2m)) \\ &= O(n \log n + y \log y + m + xy) \end{aligned}$$

Since n dominates, we can say the time complexity of the algorithm is $O(n \log n)$.

4 Proof of Correctness

Proof of Termination. By time analysis we know the *while* loop runs at most $\min(xy, 2m)$ times. Each time a class is scheduled, k, j are incremented by 1. Furthermore, each time j reaches x, i is also incremented by 1. Therefore, one of the loop preconditions will be violated as i and k keeps increasing. Hence, the algorithm terminates.

Proof of Validity. We claim that each class-room-time combination added to the final schedule is valid.

Lemma. *No two classes are scheduled in the same classroom at the same time.*

Proof. Suppose for a contradiction that C_k and C_l are both scheduled into the same slot, (R_j, T_i). Without loss of generality, suppose $l > k$ in the sorted list of classes. By execution of the algorithm, we know C_k gets added into the schedule before C_l. Then, when C_k is added, we increment i by 1 so that all classes after C_k don't get assigned to the same time. When i exceeds x, we set i back to 0 and increment j. Therefore, by the time C_l gets scheduled to (R'_j, T'_i) we have:

$$\begin{cases} j' \geq j \\ i' > i \quad \text{,if j=j'} \end{cases}$$

Therefore, it is impossible to have $j = j'$ and $i = i'$ at the same time. Hence, there does not exist two classes that conflict with each other. □

Lemma. *No teacher is teaching two classes at the same time.*

Proof. Suppose there is a teacher p who is scheduled to teach C_k and C_l at the same time. Without loss of generality, suppose C_k is more popular than C_l. Thus, when C_l gets scheduled, C_k has already been scheduled in room R_j at time T_i. Thus, *occupied*$[p]$ will contain T_i.

However, before the schedule of C_l is added, we check for *occupied*$[p]$. Since T_i is in *occupied*$[p]$, C_l will never be scheduled during the same time T_i. Therefore, we conclude this is impossible. □

Since the returned schedule satisfies the two lemmas above, it does not contain any conflicts and hence is a valid schedule.

5 Discussion

Possible solution might come from following categories: dynamic programming, graph algorithm, recursion, naive algorithm, and greedy algorithm.

First consider dynamic programming and recursion. Both algorithms involve ideas of dividing and conquering. However, in scheduling problem, since rooms can only be used by single class in each time slots, dividing students into different groups and arrange schedules respectively is trivial. Thus, dynamic programming and recursions cannot apply to this problem.

Then suppose graph algorithm can tackle this problem. According to the property of all graph algorithm, there must exist a way to transform given inputs into a graph data model, such as DAG. In any graph model, limits are interpreted as weights of edge, while objects constrained by limit are interpreted as vertices. Consider following: since the performance of algorithm depends on how many classes students can take, consequently, students are defined as vertices. Furthermore, the order of schedule has no effect on the performance, thus the graph is undirected. As stated above, limits are interpreted as edges, and according to the description of the problem, room size, teachers, time slots are all restrictions the algorithm needs to fulfill. Nevertheless, each edge can only pair up with a single restriction. Hence, an undirected graph cannot represent scheduling problem.

Naive algorithm aims to enumerate all possible schedules and compares their performance in order to get an optimal output. Yet, restrictions exist among given input, and thus requires large amount of collisions check. As a result, naive algorithm cannot given an acceptable time complexity.

Greedy algorithm guarantees optimal result for each iteration, in this case the number of class students can take. Accordingly, greedy algorithm always needs to ensure the most popular class is correctly arranged. In other words, the most popular class need to pair up with the largest room in order to get as most students as possible enrolled.

What is more, limits also exist on room size and time slots. Owing to the fact that a room can only be arranged to one time slot at a time, algorithm needs to schedule classes not only regarding room size but also guarantee that time conflict will not occur among popular classes.

Complications come mainly from restrictions on teachers and time slots. A simple corner case is that a teacher teaches two classes c_i, c_j, c_i is more popular in c_j, but the algorithm automatically arrange them into the same time slot. A natural way to solve this conflict is to simply skip that time slot and move downward. However, this will result in an empty room at that time slot. To make full use of each room, a skipped room marker is needed. Whenever the algorithm skipped a room, the marker will keep track of that room, and

arrange the next popular to that time slot.

To conclude, greedy algorithm will give an near-optimal result and can be done in an acceptable time complexity.