**ENG EC 327 Final Project Documentations:**

**BU Class Scheduler**

**Pranet Sharma, Noah Cherry, Rishav De, Aya Kassem**

**Introduction**

Every semester before registration starts, most BU students (and especially the ones with busier schedules like engineering students) struggle to find a schedule that satisfies them in terms of both having good professor ratings and having no time conflicts. The problem however is mostly caused by registering for lectures or classes that only have one section (like classes that are just composed of one lab for example) since classes composed of multiple sections usually have many discussions and/or lab sections that grant the students a lot of flexibility. We therefore decided to build a final project that will make choosing classes much easier: the program will prompt the user for the classes they need to take and then output the schedule with the highest average professor rating without any time conflicts. Our project is built off of two fundamental parts: writing a web scraper in order to get all the data our project would need, and then handling this data and writing the algorithm that would use it to give us the desired output.

**Obtaining input from User and generation of URL for Web scraper**

The program begins by prompting the user for the class code of the class that they want to take. Any class code which does not meet the length requirement or does not lead to a valid URL is discarded and the user prompted to enter a valid class code. If the class code is valid, the URL for the website is generated depending upon certain factors. This URL is then passed through the read\_html() function from the pandas library which reads the tables from the website and converts them into Dataframes, which is a type of data structure in the pandas library. Initially, Selenium web driver was researched, and the plan was to implement web scraping using it. This involved analyzing the HTML page of the website and learning about XPath as Selenium would have required the Xpath to obtain the information using the find\_elements\_by\_xpath() function. However, while analyzing the HTML code for the page it was noticed that the data to be scraped was stored in tables and thus after doing more research it was found that the read\_html() function automatically reads tables and converts them into Dataframes.

**Scrubbing of Data and Converting into Required Format**

The user is also prompted for the semester for which the schedule is required. Based on this the indexes for the classes with section A1 are obtained. Assuming that the new semester lecture starts with section A1, the index for when the spring semester classes start is obtained. After this the data is indexed from the Dataframe and put into the required list. Dataframes are indexed into based on two indices. One of them is the index and the other one is the field name. Based on that the required information is appended to the existing lists. A helper function called **time\_convert()** was implemented to convert the times from a 12 hour format to a 24 hour format.

**Pulling Professor information from RateMyProfessorAPI**

The API used is a python file that contains a main class and a Professor class. Almost all critical prewritten functions such as GetProfessorByLastName and WriteProfessorListToCSV, search\_professor, and WriteReviewsListToCSV, were dysfunctional due to references to nonexistent variables such as a reference to “self.indexnumber”. As a result only the ScrapeProfessors method was used, which itself has an innate call to get\_num\_of\_professors. Using the school ID in the url of Boston University’s rate my professor page (124 for BU), a RateMyProfApi object is instantiated. A call is then made to scrape professors which executes an API pull. The pull returns a json file with every professor in the Rate My Professor database(for the specific school ID) with information for each professor such as first name, last name, department, and average rating. Specific attributes for each entry in the json are assigned to a Professor object which are then added to a dictionary named “professors” with the “tid” as the key. TID is the url id for each professor:



**Professor Objects**

The information for each professor was stored in a user defined object named Professor. Each professor has a first name, last name, id, department, name, overall\_rating, and num\_of\_ratings. For getting the information we need, only the last name, ID, department, and overallRating.

**Searching for Professors and Returning Average Ratings**

Using the dictionary of professors and the list of last names provided by the user input, the program iterates through the dictionary until a professor is found with the last name entered. The key for that entry is then saved as the professor ID. If multiple professors are found to have the searched name then the user is prompted to enter the professors department to determine the correct professor. Once the professor ID is set, the average rating can be found by returning professors[“ID”].overallRating which is an attribute returned by the API.

**Problems with User Input**

Problems occurred when the user entered a professor name which did not exist in the database, or when multiple professors with the same last name were found. This was a difficult problem to solve because the only input provided by the user was a list of professor last names which limits the capabilities of the initial dictionary search. The best solution we found was to prompt the user for the professors department name if multiple professors with the same name were returned. The department names of each professor returned are listed in the terminal for the user to select from. If there are multiple professors with the same last name in the same department, due to the limited terms in the search query, the average rating is set to 0. Otherwise, the dictionary is once again iterated through until a professor with the same last name and department is found. The Professor class did not initially have a department attribute so one had to be added as well as assigning the value returned in the json to that object for each professor.

**Obtaining Rating data Using the Rate My Professor API and Outputting to CSV**

After using the RateMyProfessorAPI to obtain the ratings based on the Professor last names, the information is outputted using the csv.writer() function. Based on whether it is the first class to be written or any subsequent class, the information is either written to a new file or appended to the previously created file.

**Reading from the .csv file: the FileIn Class**

The FileIn class has a member function ReadFromFile() which will take a .csv file name as input and will then read from the file into the data members of the class, which are all vectors of the same length to represent the course ID (vector of strings) , name (vector of strings) , rating (vector of doubles), start time (vector of int) , end time (vector of int), and days taught (vector of character vectors) for multiple professors’ classes. For example, the first element of each of those vectors would correspond to a certain professor, the second elements of the vectors to another, etc.

The ReadFromFile() function first opens the file and checks for failures. If there aren’t any, it starts reading from the file using getline() into string variables that also correspond to the course ID, name, rating, start time, end time, and days taught. Communicating with the members responsible of the web scraping and the outputting to the .csv file allowed to know that the .csv file’s contents will be ordered like the following:

<course id>, <name>, <rating>, <start time>, <end time>, <days>

The delimiter of the getline() function was therefore a comma for all variables except for the days variable, for which the delimiter used was the newline character “\n”.

Since all the variables that contain the information we need are strings, the next step was simply to turn them into the corresponding type before pushing them back into the corresponding vector/data member. This process is repeated in a while loop until all the lines of the file have been read.

Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
Ink Drawings
id
name
profRating
startTime
endTime
days
Ink Drawings
Ink Drawings
ith index of each vector corresponds to the same professor
Data members

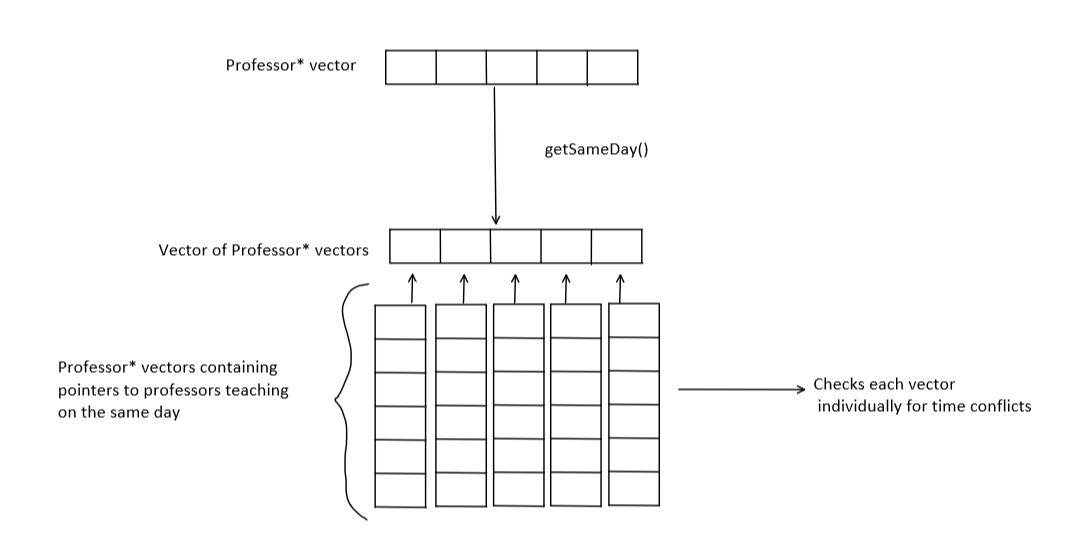

**Algorithm to find time conflicts:**

The program uses a function hasTimeConflicts() that takes a vector of pointers to Professors and returns 1 if the timings of the classes of those professors conflict with eachother, and 0 if not. The function first calls getSameDay() and passes it that same vector of Professor pointers, and getSameDay() then returns a vector where each element itself is a vector of pointers to Professors teaching on the same day. The getSameDay() function uses iterators to traverse the vector of Professor pointers and get all possible combinations of two professors within that vector. It then checks whether the two professors’ classes happen on the same day. If they do, the function first makes sure that the two professors are not already in one of the vectors that contain professors teaching on the same day, and if they aren’t it creates a new one and adds them to it.

After the hasTimeConflicts() function calls the getSameDay() function, it iterates through each element returned by the function (aka each vector of pointers to professors having their class on the same day), and checks whether or not the professors teaching on the same day also have conflicting times. Similarly to how the getSameDay() function works, the HasTimeConflicts() function will get all possible combinations of two professors teaching on the same day, and for each combination, it will check for time conflicts.

Since checking for time conflicts basically comes back to checking whether one of the classes starts after the other class starts and it ends, the hasTimeConflicts() function will check for time conflicts using two helper functions, isStartAfterStart(), which takes the start times of two classes and returns 1 if the first starts after the second, and isStartBeforeEnd(), which takes the start time of a class and the end time of another and returns 1 if the first class starts before the second ends (since our program uses a 24 hour format for the timings, getting these helper functions to work was fairly simple since it could just be done by comparing the hours of the two timings, and if they are the same, compare the minutes).

The HasTimeConflicts() function will therefore return 1 as soon as it encounters a case where one of the professors’ class starts after another’s professor’s class starts and before it ends (for classes happening on the same day). We would not have to check for other cases since we want this function to return 1 if the given schedule has any time conflicts.



**Creating the data structure:**

The first step was to create a Professor class. The class contained information about the professor, which included the course they taught, name, rating on RateMyProfessor, the start and end time for that particular class we are taking into account, and the days that class was taught by them. This information is stored in variables. Additionally, the class has getter functions for each of these data members and a parameterized constructor to initialize the data members based on the arguments passed. There is also the AddNode(<args>) function that is used to create a vector of Professor pointers to professor objects that have the same course id. It creates a Professor object in heap and pushes a pointer to it onto a vector of Professor pointers..

Back in main, a few for loops and some logic is used to create a vector of vectors of Professor pointers (the data structure needed). It uses the vectors that the FileIn class returns and creates a vector of Professors pointers that teach the same class using the AddNode(<args>) function. We have to pass a vector by reference to this function, and that vector contains the professors that teach the same class after the for loops are over. This vector is then pushed onto a vector of vectors of Professor pointers. Then the same logic iterates for the next course/class. In the end, the vector of vectors of Professor pointers is created, where each index contains a vector of Professor pointers that points to professors with the same course.

**Algorithm to create the vector required to find best possible schedule:**

The Course class contains the functions CalcAverage(<args>) and MakeCombination(<args>) functions, as well as two structs, one helper struct called loop\_calc which is used in the algorithm to iterate through the data structure and another called rating\_prof. The CalcAverage function returns a vector of rating\_prof’s, where rating\_prof contains a vector of professors from different courses and their average rating (In other words, a possible schedule). MakeCombination simply creates that vector of professors from different courses, which is used by the hasTimeConflicts(<args>) function in FindConflicts class. That vector is also used in the return vector of rating\_prof’s if the professors in that vector have no time conflicts.

CalcAverage function contains the main code for the algorithm. Essentially, the code with all its variables and helper functions cycles through the vector of vector of Professor pointers that was created beforehand. It chooses a set of professors who all teach different classes and find the average rating for that set. It does this same process for all possible combinations. It also uses the hasTimeConflicts(<args) function to check if a possible combination (or schedule) has time conflicts. If not, the information is pushed to a vector of rating\_prof which is returned. Thus, the returned vector will have the information about all possible schedules with no time conflicts.

**Displaying the output:**

Now by sorting this vector of structs based on average rating then traversing through, we can find the best case schedule. The output GUI (by friedmud on GitHub) is used to display the output in the form of an easy to read table. There is also the option to see all the possible combinations which have no time conflicts depending on user input.