FALL 2023
Independent Study

CourseHelper Final Report

Jacob Poeschel, Meredith Naylor, Alyssa Yakey, Michael Rheintgen
December 7, 2023
Professor Yuting Wu Chen
Code Repository

Abstract

Throughout the college experience, spanning hundreds of years and thousands of universities, course registration has been a source of frustration. While the University of Illinois Urbana-Champaign and other universities like it have created many resources to alleviate some of the struggles of registration, they are scattered across the internet and word-to-mouth information prevails. The purpose of this independent study was to combine various resources available to students into an easy to use iPhone application to reduce common struggles that come with course registration. Through intensive web scraping, we were able to gather course information to be used in C++ algorithms designed to provide critical course information. These functions were then integrated with Swift programs to create an app interface for the user to interact with each of the operations and add in necessary information to fit their needs.

Introduction/Motivation

Regardless of major, courseload, degree type, or any other variables, the words “I wish” seem to accompany any academic conversation. The sentences “I wish I knew to take that class earlier”, “I wish I knew that professor was so hard”, “I wish I knew that class matched my interests so closely”, or even “I wish I knew my current grade” occur so frequently it is easy to become accustomed to them. Just because we are used to them, however, does not mean that we should be. In reality, the majority of college students pursuing their higher education are impacted by the lack of available resources and information regarding course schedules, professors, grades, or any number of other factors, each of which could be critically important to their learning or even their career. Even for universities which have all of this information available, it is generally dispersed among several websites making it incredibly difficult to process in an efficient manner. To compound the issue, many students only learn about these sources through word of mouth, meaning many incoming freshman or underclassmen suffer even more intensely from the lack of resources despite needing to make critical choices about coursework.

Our mission is to design a system which makes all of this critical information readily accessible and usable for students of all ages. The UIUC Course Helper App combines various functions such as a schedule generator, grade calculator, and course recommendation system into one easy to use and easily accessible application. This functionality was achieved through the combination of algorithm programming in C++, application development and linking in Swift, and data retrieval done through Bash and Python.

Implementation Details

Overview:

The implementation of this project can be broken down into five separate areas of development: web scraping/data retrieval, app development, algorithm development, bridging, and scripting, each of which are described in more detail in their respective sections. Here, we will discuss the flow of each section and how they combine to create our end product.

The first critical step to create the course helper was to perform all necessary web scraping and data retrieval. This step involves reading from the UIUC course explorer, looping through all of the available information for each course, and copying that information into a CSV file. Once this is done, data retrieval is complete. The created scripts select the set of information to be retrieved based on arguments sent as input by the user, allowing a large degree of flexibility for information collection while also maintaining the ability for an overall sweep of all UIUC courses.

The next step is developing various algorithms to handle the bulk computations used to generate schedules and calculate grades. This phase relies on all of the data compiled in the web scraping phase.

Now that all of the calculations are complete, they need to be connected to the app. This is done by bridging the C++ and Swift code in Swift.

Finally, the results must be displayed. The design and functionality of the app itself are handled in Swift. The connection of the algorithms and Swift in the bridging phase grant the app the ability to present a usable version of each designed function. At this point, the app implementation is complete.

Web Scraping/Data Retrieval:

The first major step in any web scraping process is choosing the correct web driver to allow you to optimally scrape the data you need without requiring bulky and confusing code. We decided it would be ideal to use a Selenium webdriver as it can quickly and accurately pull data from webpages by utilizing its extensive library of functions. Since there are several different resources available to students to access course information, we chose to pull from the Course Explorer as it contained all of the information necessary to generate an accurate class schedule for the average student. For example, the Course Explorer contains information such as CRNs, section times, section types, credit hours, etc., which is all required when generating possible schedules.

Prior to scraping the data for each individual course section, the web driver must first be able to access its course page. We accomplished this by creating a Python script that utilizes BeautifulSoup, another web driver, to gather the course name and URL for every course in the university and compiles the data into two separate .csv files. Since every page in the Course Explorer has the same base URL, the scraping program is able to grab the subject and course number from the .csv file and concatenate it with the base URL to access every page.

On each course page, the web driver is searching for specific elements on the page that it will scrape information from. An important feature in the Python library is the try-except block. This allows the web driver to search on the page for an element, in this case, a table holding the course information, and if it's not able to locate it in a specified amount of time, stop searching and move on to the next instructions. This procedure is invaluable for our web scraping as it allows us to recognize very quickly whether or not the course page is valid and it can determine whether it has looped through every row of the table and should move on to the processing of the data.

As the program loops through each section of a course, it gathers required data for each variable and stores it in a two dimensional array, with each row corresponding to each section of every class offered.

A major issue that arises with the process of web scraping is the formatting of the data that's being compiled. If the data is not formatted correctly upon its retrieval, the algorithms will fail. Thankfully, Python makes it easy to introduce strict formatting guidelines. Before a field is placed into the CSV, it is trimmed to delete leading and trailing spaces. This is enough to format most fields correctly. However, there are a few critical fields where more steps need to be taken. For example, in the instructor field, the comma between the instructor name must be replaced to ensure both the first and last name are grouped as one field and not as two. Additionally, when the instruction times are first scraped, they are formatted in one string including both the start and end times. In order to be viable data for our schedule generator functions, this string needed to be split and reformatted into two separate variables. And, since credit hours can be listed in a range (1 TO 4) or as an option (3 OR 4) these both must be read and translated to an integer value. Finally, we must loop through the CSV and remove all empty lines. After the data is properly scraped and formatted, it is written into a .csv file line by line so it can be read from and used in the algorithms.

Scripting:

To make the data retrieval process more efficient and understandable, we decided to design a simple bash script to manage both python scripts from a centralized location. Bash is a convenient language to manage arguments and provide usage information for users who wish to utilize the full project functionality without needing to understand a precise call order or argument structure. The first phase of the script is the URL pulling phase. If data is being retrieved from a constant set of URLs, there is no reason to repeat this phase, so it can be skipped by passing in the GenSkip argument. The second phase of the script is choosing from which URLs the data should be taken. The user can control this by passing in a major (ie "ECE"), a course number (ie "ECE 397"), or both. The bash script passes these arguments into the data retrieval script which then appends them onto the respective base URL, allowing access to the website specified by bash. At this point, the script has achieved its goal of providing a more convenient way to manipulate the given python scripts, signifying its completion.

App Development:

To create the front end of the app we used Swift. The first step for our app development was a storyboard. In Swift, a storyboard is used to connect different app view controllers. With the storyboard we created segues between each of the pages. For example, if one were to click the continue button at the start of the app it will bring the user to the main page. Once all the segues and the initial app layout was in place we had to start connecting the buttons to the Swift code. In Swift there is a class for every view controller. We connected both buttons and labels to their respective view controllers. In Swift labels are text on the screen that can be updated by the code. In most of the view controllers we created a background function which relies on a dictionary created to represent the user's selected school. With this dictionary, we can pull the colors for each school upon every page creation, allowing for dynamic screen coloring. This marked completion of the basic app shell.

The integration of the C++ proved to be very difficult. We spent two weeks researching and implementing many different ways to try and get a basic C++ function working. Eventually we made a separate app from scratch and were able to get a C++ function working on that app. The main step one must take to implement this integration is to create a bridging header, change the compiling to C++, and change the build order so the C++ compiles before Swift. With this method of integration initially we were able to get the grade calculator function working on the click of a button. For this to happen, we created a bridge C++ function that was called cppFunc which contains simple logic to create homework and an exam vector of grades. We then passed a vector of weights to the grade calculator C++ function and returned the result back to the Swift. In Swift we set labels to the result of C++ functions so it displays on the app.

Once we had simple C++ integration it was time to connect user inputs to the C++ functions. The first functionality we built out completely was the grade calculator. As discussed above, the grade calculator takes in a vector of doubles containing the weights of the grades and a vector of double vectors containing all grades for each weight. When integrating this logic with Swift there are two main connections we have to make. In Swift we made global variables to hold a title, weight and a vector of grades. Once the user confirms their input we begin interacting with the bridging C++ functions. The main purpose of the bridging C++ functions is to take the Swift input and connect or prepare it for the main C++ code. Their secondary purpose is to take the results back from the main C++ functions and return them to Swift.

The connection to the schedule generator C++ code proved to be more challenging than the grade calculator. Since the schedule generator returns a string, the result must be cast to a type that works with Swift. The type conversion that we use most is std::string to char*. The main difficulty in connecting the schedule generator was the file path. The schedule generator uses the CSV file created in the data retrieval phase so we included the CSV file in the project. However, we later discovered that its path is constantly changing. Eventually, we learned that you can use simple Swift logic to find the path to the file inside of the Swift code, rather than it being hard coded. This allowed us to guarantee an accurate connection between the schedule generator and the CSV file.

The creation of the course explorer involves mostly Swift code and a connection to a C++ map. The course explorer loops through eligible classes using vector logic, only displaying information on the course if the index is not negative one. When the index is negative the starting page of the course explorer will be displayed which gives the user instructions on how to use the course explorer. Additionally, storing an index as a global variable in Swift allows us to maintain page states, allowing the user to exit and re-enter the course explorer without losing their current page.

Similar global variable logic is used for the favorites page. Favoriting a class in the course explorer appends that class to a global vector in Swift. If that global vector is not empty then the favorites page will display the contents of that vector. The favorites page and the course explorer have the option to add courses to the schedule generator. This gives the user three separate places in the app to add courses to the schedule generator. The courses currently in the schedule generator are also a global variable. The course recommender lets the user add a course to the generator if that class is not already in the vector or remove the class if it already exists. However, the favorites page just offers functionality to add all of the favorited courses to the schedule generator.

Throughout most of the project, Swift has been a major challenge as we have no formal experience with the language. The process of learning a new language from scratch without oversight induced a situation where we had to make progress and learn solely based on available resources. This is a situation that we will all encounter in our careers, so this experience is certainly invaluable.

Algorithm Development:

The bulk of the functionality for our application is handled by algorithms created in C++. These functions make use of custom classes designed to represent critical sets of data, including Course, CourseSection, Schedule, and ScheduleGroup classes. The course object contains a name and a list of its corresponding CourseSection objects, each of which contain all necessary course information such as CRN, location, course type, etc. Each schedule object contains a vector list of CourseSections which represent a complete section. Finally, the ScheduleGroup object contains a list of schedules. The two main algorithms implemented with the help of these objects are the schedule generator and grade calculator. The schedule generator begins by pulling course information from the CSV created in the data retrieval phase while the bridging between Swift and C++ allows for the schedule generator to access the user input list of courses and check this list against the CSV. For every course that is found, a new CourseSection is created and added to its Course object. Once each Course is finalized, they are added to a CourseList representing the user input list of courses filled out with actual data fetched from the CSV. Then, this CourseList is passed along with a buffer ScheduleGroup object into the recursive schedule generator function. This function checks all possible combinations of CourseSections for the given courses and fills out the ScheduleGroup buffer with every Schedule that is valid. Before a CourseSection can be added, the function ensures that there are no time or day conflicts and marks a Schedule for needing an override if there are an improper number of credit hours. Upon the return of the recursive function, the ScheduleGroup buffer is full and can be printed in Swift to display possible schedules.

The grade calculator algorithm allows the user to enter in a list of received grades and their overall weighting. Then, simple calculations allow for a final result to be returned and once again printed in Swift.

Each of these algorithms, along with all of their helper functions, are extensively tested in the tests.cpp file.

Bridging:

The bridging files are written in C++ with the purpose of connecting the algorithms with the user input in Swift. At the beginning of the app we have a C++ function called CppInit that dynamically allocates many vectors that are used throughout the app. For the grade calculator we have an AddCategory function that takes in the weight of a class and adds that to a dynamically allocated vector. AddCategory also creates a new vector of doubles to hold each grade for the class. In order to add a grade to the category, we push the grade into the category's grade storage vector. Once the user hits confirm, the completeCategory function will be run which adds the grade list to the vector of lists to be entered into the grade calculator function in C++. The last step for the grade calculator is initiated when the user clicks "calculate grade" and the getGrade function is called. GetGrade utilizes the preexisting calculate grade function to determine the resulting grade. Then, it clears all the vectors that were used and returns the grade to Swift.

Another main bridging function connects the schedule generator to Swift. In Swift the user will enter courses that go into a course list vector. Once the user hits generate schedules the course list vector gets sent into C++. To call the main schedule generator we send in the created vector of classes, the path to the CSV file, and a buffer. Once the string of schedules is returned it is converted into a char* and returned to Swift.

C++ is also used for the course search feature in the schedule generator and the course explorer. In C++ we have a map containing all of the data on each class. These functions are getName, getCreditHours, getAvgGPA, getPre, getGen, and getDescription. These each take in a char* of the course name and return a char* of the correct mapped data. The functions check to make sure that the data exists in the map and if not will return NA.

Possible Continuations

We believe that this project has a very promising future. The main features of our project currently are the course explorer, grade calculator, schedule generator, and the favorites page. We have a multitude of ideas for improvements in each individual feature, functionalities to add and improvements to the app as a whole.

Course Explorer:

The course explorer is a place that we believe has potential for major growth. Currently in the course explorer a user can favorite classes, add them to the schedule generator, and rotate through the courses to view the main characteristics of the class. At the beginning of this project our main goal was to combine information that a student would have to visit many websites for into one location. The course explorer combines the information one can find on the UIUC course explorer and the average gpa of the courses that are found at waf.cs.illinois.edu. We find information on both these sites fundamental when choosing a course.

The main goal for the course explorer in the future would be to provide the user with more information and collaboration with other students. One further implementation of this expansion would be to add videos that act as an ad for the course. These videos could include a general description of what you learn and some students' final projects. There are many courses that we have experienced in ECE where students make incredible final projects. These can be highlighted in the course explorer. To further the course explorer we also want to add a centralized place where students can write reviews on the course. These reviews would be displayed in the explorer for students to get a deeper understanding of a course.

In the future we see the course explorer having the ability to look for a specific general education course, a tech elective, or an advanced cs elective. The idea would be to incorporate a filter that can give the user the exact course to align with a search. This filter would have the ability to find “match” courses for students. The course explorer on the UIUC website can fit most of these filters although the idea would be to build it out to be major specific. For example, one could filter for a eight week ECE tech elective that has above a specific average GPA. A continuation of this filter would be to create an algorithm to match the student with courses. To create this ability we would have the student review or rate the classes they have taken in previous semesters or fill out a form of their interests. Then, if the student was looking for a general education course that fulfills specific requirements the algorithm could look at how they enjoyed other courses and match them with a general education best suited for their interests.

The intent behind the course explorer is to give students a full picture of a class before they sign up. Signing up for courses is a very stressful time for many students. The process that we find ourselves going through is reading the descriptions of the class on course explorer, checking the average GPA, searching for reddit reviews, talking with advisors, and asking other students about the course. This process makes it very hard to fully grasp the idea behind a course. We have especially found that choosing courses becomes increasingly difficult as you narrow down toward a speciality. The course explore aims to help guide students through the process a little smoother.

Grade Calculator:

The grade calculator has an opportunity for massive improvement. Currently, the grade calculator has the ability to calculate a student's full grade in the course. The grade is just a percentage and does not take into account any curving. The user can title different categories of a grade (ie. homework, machine problems, or exams). When entering a title the user will also enter the weighting of the grade. In this example, homework could correspond to 20% of the total grade and machine problems could account for 30% leaving the remaining 50% to exams. Then, once the user enters the title and weight for a category, they must enter all of their grades for that specific category. This design performs accurately yet is extremely limited. The main issue is that the user must know all of their grades in the course to receive an accurate result since the calculator mandates that all sections combined must have a weight of 100.

The main addition for the grade calculator would be to calculate grades for students throughout the semester. The idea behind this would be that students could enter the courses that they are taking into the app. Each category and its weight would be entered initially; then, anytime a grade is received throughout the course it can be added and an updated grade will be returned. The major changes behind the grade calculator to incorporate this feature would be removing the constraint that ensures the weights add to one hundred. The other main addition for this feature would be to add in a CSV file that we could locally write to and save the data that the user entered. The CSV file would give us the ability to hold the user data locally.

Another addition that we would add to the grade calculator would be a final grade calculator. In the final grade calculator the user would be able to enter their current grade in the class, their desired grade and the weighted percentage of the final. The output would represent the grade needed to attain the student's desired grade.

To further add to the grade calculator we would like to have an option for students to enter in grade cutoffs. In our experience some classes are curved and some will provide students with specific grade cutoffs at the beginning of the semester. This feature would provide the student with a letter grade as well. This would only be the case if grade cutoffs are set for the students course.

The grade calculator is an area where we see potential for major improvements. The idea behind the improvements is to make the grade calculator more versatile and meet more students' needs. The main goal of the grade calculator is to give the students a general idea of how they are doing in the course, and adding in any of these proposed functionalities will greatly increase its ability to do so

Web Scraping/Data Retrieval:

Currently, the data retrieval scripts take several shortcuts to deal with edge cases found in course information. For example, it is possible for a course to be multiple types (ie Lab - Lecture). These courses are currently ignored but should obviously be treated as legitimate courses. Additionally, courses which are designated as “Online” are not yet supported. Another area which lacks support are courses that only take place for a portion of the semester. Right now, there is no distinction between each type of course in a schedule, meaning that an 8 week course and a 16 week course are behaviorally equivalent. Furthermore, although ranges of credit hours are allowed in the CSV, they are not properly accounted for in the credit hour totals for each schedule. If a class provides a range of potential credit hours, a baseline sum is assumed, which could lead to improper schedules. These are solely the inconsistencies that we have encountered thus far. It is likely that as we continue the project, we will continue to discover other edge cases which must be accounted for.

Schedule Generator:

The first way the schedule generator should be improved is adding “breaks” for students to determine time periods where they do not wish to have any classes. This feature will allow a user to specify a time and day of the week where no classes should be scheduled. Before a CourseSection is added to a Schedule, this section can easily be checked for overlap with all Breaks, which would create the desired functionality.

The main improvement for the schedule generator is allowing it to handle all of the edge case course information we might encounter. While we also need to add support for these cases in python, we must also ensure that all of these courses can be added to Schedules and each can be placed appropriately independently of their irregularities.

A more creative improvement could be finding a smart way to apply the extra course information we currently have, but do not use, such as instructor or location. Perchance we could recommend schedules that place classes on a convenient walking path or schedules which prioritize sections with the most liked professors, etc. The abundance of course information makes the possibility for such changes limitless.

Favorites:

The favorites section is currently built out to display the favorites that were selected in the course explorer. The functionalities of the favorites page are to clear the favorites or add them to the schedule generator.

In the future we would like to add functionality to the favorites page that allows students to see all the details of the class by clicking on the class. This allows the app to be more efficient for the user.

Improved Scripting:

The bash script can be generalized to allow for more user control. For example, it could be expanded to allow a list of courses to scrape from or a list of majors to check. Most generally, it should be able to scrape courses across all majors if desired.

Graphic Design:

It is clear that our app is in need of major design improvements. For the design to be more professional, we would need to determine a theme, create animations, store user color preferences, and add functionality for any similar design elements.

New Functionalities to Add:

Course Recommender:

The course recommender's main functionality would be to find a course that fits the interests of the user. The student would input a few key word interests then the course recommender would come up with a list of courses that the student might be interested in. Once a course is selected the student can view a description of similar courses as well as favorite those courses.

Planner:

The planner would give students the ability to decide when they are going to take specific courses. This feature is to give the students a place to put the course that they have favorited. The planner will also have the ability to write into a CSV file and save the data similar to the future grade calculator.

Course Search:

The course search feature would be very similar to the search that is inside the schedule generator. The creation of a search page separate from the schedule generator would make the app more efficient for the user. In the search the student would be able to see all the information corresponding to the discussed improvements of the course explorer.

Expansion:

Instead of the app just being UIUC based we would add in other schools. With this we would have the ability for a user to choose their school the first time they open the app. These preferences could also be saved in a CSV, allowing us to maintain user settings and selections. They would have the ability to change this later in settings. We would also need to figure out how to add videos into the app for the tutorial and for the course explorer. Additionally, we would need to add the ability to write to local CSV files so individual user information can be saved on their phone when they close the app. This improvement is necessary for the future course explorer, grade calculator, planner, and favorites page.