# TigerPath

Final Report

Richard Chu (PM)
Barak Nehoran
Daniel Leung
Adeniji Ogunlana

## A. Planning and Milestones

### 1. Timeline

When creating our design document, we mapped out a rough timeline of our goals and the features that we wanted to complete by the end of each week. This timeline was helpful in the beginning, but we strayed away from it as we continued working on our project. We thought of countless new features that we had not had in our original timeline, and we decided to track the status of all of these features in a new Excel spreadsheet. We began following the feature list and decided to assign priority levels to each feature rather than follow a strict timeline. Developing our app by feature was a lot more helpful because we could distribute tasks much more easily.

In the end, we were able to complete all of our goals for our MVP relatively early, so we spent the remaining time implementing the features that our users requested the most. We still have a lot of exciting features to implement, but we are very satisfied with what we've done so far.

### 2. Development Process

We decided to focus first on implementing course searching and dragging and dropping courses into individual semesters. We thought this was an important milestone to hit because if we implemented this feature, then at the very least our application would be a functional four-year planner, just without a requirements tracker. After we reached this point and added the ability to store user schedules in the database, we focused next on implementing the requirements tracker. We eventually finished the tracker, completing our MVP, and began implementing additional features afterwards.

### 3. Planning Practices

We evaluated each of our strengths and discussed what areas of the project we preferred to work on. We decided to split our group up to focus on different aspects of the app: creating the major/degree requirements data, frontend, and backend.

In addition to our weekly meeting with our TA, we also met as a group for a few hours twice every week. This was really helpful to update each other on our progress, but being together also made it easier to work. Although we could always work on the project on our own and communicate online, being together facilitated communication when multiple members of the group were working on features that linked with each other. We could call each other over to look over some code or get questions answered instantly about each other's code without having long comment threads on GitHub.

## B. Design Decisions

## 1. User Interface Design

Discussing with our TA and looking at similar past COS 333 projects, we realized that a lot of past projects don't have users because their user interface is not clean and easy to use or it took too long for the user to get set up with the app. Especially with a four-year planner, which has eight semesters worth of course data on one page, there's just so much going on. If the user interface is not intuitive, the user may become overwhelmed and decide not to use our app.

We decided to go with a one-page app so that the user wouldn't have to click through multiple pages in order to plan out their schedule. Having everything on one page makes it so that the user can see all of the information at once, such as course searching, their schedule, and the requirements that have been satisfied. However, given the numerous elements that need to be shown on the page, we also had to make sure that we didn't clutter the user interface too much, so we decided to make some information hidden until the user wants to see it. When the user hovers their cursor over a course in the schedule, there are tooltips which appear and show more information, such as the course title and whether or not the course is a duplicate course. When the user hovers their cursor over a course in the search results, links to the course evaluations and ratings appear.

Our application relayed a lot of information through certain color codings. We thought about having a legend to convey this information but decided to minimize this as well. This decision also guided our process in how we chose the color scheme for our application. For the requirements tree, we decided that green nodes could implicitly indicate a requirement is satisfied and red nodes could implicitly indicate that a requirement is not satisfied. We also used a neutral-shade of yellow for nodes that could satisfy a requirement but were optional. We decided not to use any of these colors in other areas of our application so we didn't confuse the user. For the courses, we had three types of courses: fall semester, spring semester, and both semesters. We color coded each of these courses using three different colors (purple, blue, orange) that were contrastive from the previous colors. To implicitly convey for example, purple to code for fall courses, we colored all of the fall semester headers to be purple as well so the user would, in the course of using the app, make the association that all purple courses should be under purple semester headers.

We also tried to make our app more simple and intuitive by implementing a fluid drag and drop interface, which allows the user to easily rearrange the courses they are planning on taking in each semester. Also, we decided to have the search box search for courses from all semesters by default rather than making the user specify which semester to search. This not only reduces the number of clicks for the user but it also allows the user to plan for multiple semesters at once. They can drag courses to multiple semesters without having to constantly switch semesters or plan out one semester entirely before moving to the next.

## 2. Requirements JSON Structure

At first, we were hoping to get help from the Registrar's office to get a list of the requirements for each major. When we realized that they did not have this data in a format they could give to us, we realized that we would have to compile it ourselves. Many of the requirements are spread out across a variety of different websites on the internet. We found that the undergraduate announcement website was very useful, but the information available was neither complete nor machine readable.

We wanted to design a standard format to express the major requirements that was flexible enough to be able to encode all of the idiosyncrasies of each major and at the same time be machine readable and standardized enough that a single piece of code could check them without any hardcoding.

Our initial idea was that most requirements can be simply expressed as an AND or an OR of some number of sub-requirements. Using a tree with courses at the leaves and an AND or an OR at each node would theoretically work since this is a universal way to represent Boolean functions, but we found that we would need additional features to make it feasible for a human to create and understand. Among them, we decided to have lists of courses as the leaves of the tree, and requirement lists as the internal nodes, which reduced the depth of the trees. We also decided to allow encoding functions that interpolate between AND and OR by encoding the minimum number of sub-requirements needed in order to satisfy a parent requirement, and the maximum number of courses or sub-requirements that will be passed on further up the tree if it is satisfied. The last addition was to specify requirements for which courses can double count. That is, a course can count for that requirement even while also counting for other requirements. We decided that the default behavior would be that courses only count for one requirement at a time, because this seemed to be the case for the majority of major requirements.

Because we were expressing the requirements as a tree, we decided to encode them in JSON format, with nested layers of lists and dictionaries representing the tree. Once we had decided on a set format, we wrote a JSON schema to automatically check and enforce this structure, and we also wrote a script in HTML and Javascript to display a flexibly defined form that once filled, automatically creates the JSON file. This streamlined the process of encoding the requirements and reduced the human error involved.

## 3. Verifier Design

Once we had created some of the requirement JSONs, we used python libraries to load the requirements trees as python dictionaries. We implemented a modified and simplified version of the Ford-Fulkerson algorithm to find the largest flow from the leaves of the tree (the courses) to the root (the full major requirement).

One challenge we ran into was deciding which requirement to have a course count for. Our first attempt involved choosing the assignment that would satisfy the largest number of requirements. We soon realized that while this technically worked, this was often annoying for the user. It became really slow when the number of courses to assign grew large, and more importantly, the choices it would make would not always match what the user wanted.

To address this issue, we decided to reformulate the verifier to allow the user as much control as possible over how the courses are assigned to requirements without adding too much extra complexity for the user. To do this, we decided that when a user adds a course, the verifier would find all of the requirements that the course could satisfy. If there was only one such requirement, the course would be assigned and counted for that requirement. If the course could count for multiple requirements that required exclusivity, it would display the course to the user under each such requirement as a provisional assignment awaiting user input. If the user clicked on one of these provisional assignments, it would make that assignment official and remove all the other provisional assignments for that course, recalculating the satisfied requirements accordingly.

## C. Languages and Systems

### 1. Frontend

For the basic template of our application (such as the semester columns and header toolbar), we decided to stick with basic HTML, CSS, and Javascript. We did make use of the Bootstrap library because of how easy it is to setup and use and how mobile responsive it is. The library also provided nice default styles so we could get off the ground faster and focus on the core functionalities of our app. For the more interactive parts of our application; such as the search pane, dragging courses to semester columns, and the requirements tracker; we decided to use ReactJS. Because of how frequent the search pane and requirements pane would be updating, ReactJS makes it really efficient with its component based design (React only re-renders certain components when it is necessary). It was also really easy to integrate with the basic HTML skeleton we had built at first because React allows us to select any container on the page to render our items in. One team member was able to work on the frontend without using React at all and another member was able to merge his React code seamlessly—overall ReactJS made our code very modular.

Because we were already familiar with AJAX and our requirements data structure was best formatted as a JSON file, we decided to use AJAX to perform GET and POST requests. JSON objects are frequently transferred between the frontend and backend whenever a user's schedule updates or when their requirements pane updates.

We tried a number of drag and drop libraries before finding the perfect one. Not only does Dragula provide simple and fluid drag and drop functionalities for the user (One way drag from course search to semesters, dragging between semesters, translucent previews of where the course will be dropped), but it was also extremely easy to code with. With dragula, we could simply select containers where we wanted the children elements to be draggable and that was it! We did have to write a bit more code for more precise functionalities like one-way dropping but overall the library was fantastic to work with.

To display the requirements pane, we used the react-treeview library. It was a simple display of a tree with no bells and whistles which was what we wanted for our UI design. The library was a little annoying to work with because of how inflexible it is. It took a some time and a decent amount of code just to make the entire node of a tree clickable rather than just its arrow.

### 2. Backend

We knew we wanted to use a web framework in order to help us build the web app, because it meant that we could get started more easily and we could focus on writing the business logic of our app instead of worrying about the low-level details. We were deciding between a couple of web frameworks, including Flask and Django, but we decided to use Django because it is well-documented, easy-to-use, and has a wide range of out-of-the-box features (database ORM, security). Django also uses Python, which we were all familiar with already and which has a robust ecosystem of packages.

We decided to use both ReCal's scraper and the Assignment 4 scraper to scrape for course offerings. ReCal was built on a similar stack to our own (Django and Postgres), which meant that its scraper could scrape for courses and automatically populate our database tables with them. ReCal's scraper is also used in production to serve thousands of students, which meant that it was robust enough to handle our use case as well. In addition, we integrated the Assignment 4 scraper in order to populate the courses with the distribution requirements that they satisfy, which is a piece of information that the ReCal scraper did not provide. One thing we could have done differently was to modify the Assignment 4 scraper rather than use ReCal's scraper as a base. We noticed very late in our development process that the webfeeds that ReCal scrapes from does not provide the distribution areas for the courses. In order to obtain that information, we had to run the Assignment 4 scraper which made the process took twice as long because we run two scrapers for each semester. This is not really a big concern however since we only rescrape for courses every semester.

## 3. Database

We were initially looking at using the MongoDB NoSQL database, but we quickly realized that it did not fit well with the Django web framework, which only supported relational databases. We felt that Django was a more essential technology to keep than MongoDB, so we ended up switching to the Postgres relational database instead. This worked out very well for us, since we were able to more easily integrate ReCal's course offerings scraper into our app, as they also used Postgres.

At first, we used Heroku Postgres to host our database, but we easily exceeded the free tier limit of only 10K rows. Since we were reluctant to pay monthly for a database during development, we did some searching around and landed upon the GitHub student pack, which provided $50 worth of credits on DigitalOcean, a cloud computing platform. While it wasn't as easy to get a Postgres database up and running on DigitalOcean as it was on Heroku, we finally set it up properly and was able to host our database on DigitalOcean without any row limits.

## 4. Version Control

We used Git for our version control system and GitHub to host our code. We implemented a feature branch workflow, which meant that each feature of our app was implemented in a separate branch from the main branch. After someone completes a feature, he would make a pull request to merge the feature into the main branch, and then someone else would do a code review to make sure that the code is readable, bug-free, and actually does implement the feature correctly. Having such a rigorous development workflow helped us optimize for maintainability by preventing hacky code from getting merged into the app just to solve something for the short-term.

### 5. Hosting

We knew from the beginning that we wanted TigerPath to persist beyond the end of the course, so one of the first things we did was buy a good domain name on Namecheap (http://tigerpath.io). We decided to use Heroku to host our website because it had a free tier, it was easy to set up, and it was used by other successful COS 333 projects. We set up two separate Heroku websites: one for a development website, which had the latest code from development, and a production website, which would be the actual website that our users would visit. We decided to have these two environments so that we could test our latest changes and catch bugs early on the development website before they became live on the production website. We also set up automatic deployment to Heroku from our code on GitHub, which enabled us to iterate more quickly. The “dev” branch on GitHub was automatically deployed to our Heroku development website, and the “master” branch on GitHub was automatically deployed to our Heroku production website.

### 6. Third-Party Services

We used a few third-party services in order to enhance the user experience. We decided to integrate the TigerBook API into the onboarding page so that the major and class year of the user would be automatically filled into the dropdowns based on their NetID after logging in with CAS, which saves them time from picking out the correct options. We still ask the user to confirm this information just in case the TigerBook API provides incorrect information, the user does not have his/her information in TigerBook, or the user has not declared a major.

We also integrated the TigerApps Transcript API in the onboarding page so that the user wouldn’t have to manually add courses that they’ve already taken in past semesters to their schedule, which helps the user jump right into planning out their future semesters. This removes a source of friction that the user would have in trying out our app, which helps us acquire and retain more users.

Finally, we added tracking with Google Analytics in order to gather metadata about how our users are using TigerPath. We can see how many users are using the app and what they are doing on our app, which allows us to find potential pain points and ultimately improve the user experience.

## D. Testing

### 1. Self-Testing

After implementing every feature, we not only tested the new feature rigorously but also made sure to test all previously implemented features as well. With our peer review system, every member of the team made sure to checkout every pull request and test all features of the app before approving the pull request to merge with the main branch.

We also did some stress testing so we could guarantee a fluid and bug free UI for the normal user. We tested situations where we had over 15 courses in one semester, we dragged courses rapidly between semesters, we settled and unsettled requirements rapidly, we searched for queries that returned hundreds or thousands of courses.

We did some edge testing for both our search engine and our requirements tracker. We added duplicate classes, grad courses, searched for courses with unconventional course codes, settled duplicate courses to different requirements.

For performance testing, we made sure to test requirement tracking and course dragging with eight completely filled semesters. We also compared the speed of our search engine to popular, widely used apps like ReCal and Princeton Courses and made sure the speed of our engine was just as fast.

### 2. Beta Testing

During the early stages of our launch, we had a few of our friends test out our app. With every tester, we wanted to see if they could use our app with no guidance from any of us. Our priority was to see if our app was intuitive enough to use. After checking for this, we asked our users to continue using our app—and even try to break it—and report any feedback or bugs. Some of the improvements we made based on user feedback involved recoloring the semesters to make the color coding more intuitive and adding a grabby-hand cursor to speed up the self-discovery of the drag-and-drop functionality.

We decided to test with friends at first so we could do thorough testing and easily ask follow up questions. We could also ensure that their major was supported. After finalizing our app more, we began to publicize our app to certain groups with our target majors (since we do not support all majors yet, we have refrained from fully publicized our app yet). We implemented a feedback form to gather more feedback from these users.

### 3. Automated Testing

Since the verifier (which takes a user schedule as input and outputs a requirement tree with the courses under each requirement it satisfies) has to deal with a variety of different kinds of requirements and course schedules, and there is lots of room for error, we decided to fully automate its unit testing. We created numerous files containing test schedules with courses spanning all of our majors and created the expected output JSONs. We then wrote a script to run through each of them and compare the result to the expected output. We ran these tests whenever we made changes to the verifier to ensure that the updates we made did not break existing functionality, and we updated the tests both to catch more potential bugs as we discovered them, and to test new features as we added them. Before any modification to the verifier was approved the testing script would be run and it had to pass all the existing test cases.

## E. Next Steps & Advice

### 1. Next Steps

We think that TigerPath would be an invaluable tool for Princeton students to use to plan out their four-year course schedules. Thus, we plan on supporting TigerPath beyond the end of COS 333 and releasing to the wider Princeton community as soon as possible. In terms of features, we think that supporting more major requirements is one of the most important things to do next, since it would expand our target audience. However, we want to be careful not to compromise on quality and accuracy while adding new majors; it's vital that we try to provide information that is as accurate as possible, so that users are able to trust our application. Other features that have been highly requested by our beta testers include searching for courses by the requirement they satisfy; exporting the four-year course plan to PDF, which would help academic advisors; supporting certificates; and supporting the fulfillment of requirements through other means (such as by study abroad, AP credits, summer classes, and waivers).

## 2. Advice for Future COS 333 Groups

The importance of a good development and deployment workflow cannot be understated; it makes it so that you can concentrate on working on the business logic of your application instead of trying to fix merge conflicts all the time or wondering why your infrastructure isn't working. Use Git and GitHub with a feature branch workflow where each "feature" of your app is implemented in its own branch, and then merged into the main branch when it's completed. We set up a pull request and code review process such that every line of code that is added to our app has to be sent through a pull request and reviewed by one other person. This meant that silly bugs were caught early and ensured that the code was always readable. We also found it helpful to have separate development and production websites, and to have a "dev" branch and a "master" branch that respectively tracked each environment. This made it so that we could sometimes catch bugs before they were pushed to production and viewable by all of our users.