Problem Background

There are several areas of work in the non-trivial task of course scheduling for the CSSE department. Currently my advisor Dr. Stephen Beard holds the responsibility for this task and I worked on a couple areas of this task as part of my senior project.

Overall there are 2 major constraints to consider in course scheduling:

1. Professor preferences:
    a. Availability: what days/times are professors available to teach?
    b. Other preferences:
        i. MWF vs TR schedule
        ii. Back to back courses vs break in between
        iii. Room preference
2. Room availability. In addition to the classrooms in Frank E. Pilling, the CSSE department is granted a set number of rooms in other buildings as well.

The bulk of the work can be summed up by the following 2 goals:

1. Collect professor preferences in a manner that is convenient to them and possibly increase accuracy/number of responses
    a. Before my project, this data was collected via a Microsoft Forms survey (see example). While this more or less got the job done there were some areas of improvement here, both from the user and “developer’s” perspective.
        i. Namely, from the user side, the web app allows professors to:
            1. provide their preferences for the year (rather than repeating the survey at the middle of each quarter)
            2. modify their preferences (rather than filling out a one-time survey for the quarter)
            3. save and reuse preferences (rather than re-entering similar data for the survey each quarter)
        ii. From our perspective, this web app allows us to:
            1. Validate professor input easily (see example)
            2.
2. Determine an optimal (or near-optimal) schedule for each professor
    a. This aspect of the project had to do with the actual task of course scheduling based on the preferences received from professors. Namely, determining rooms + times for professors to teach.

For my senior project I worked on both these goals. Initially I worked on goal #2 for most of the first quarter before transitioning to goal #1 for the remaining duration of the project.

Part 1: Determining Professor Schedules

Background

Before I began working on this project there were already existing scripts that ran a scheduling algorithm to determine ideal professor schedules. The OptaPy library, an “AI constraint solver" in Python, was leveraged for this process.

Determining schedules consists of the following steps:

1. Data preprocessing+preparation
    - This step consists of filtering out or marking specific data points - for example, some classes need to be manually scheduled due to special constraints (ex. CSC 445 which can be taught on a MTRF schedule). Additionally the data must be converted into numerical categories and representations that can be understood by Optapy
2. Scheduling algorithm
    - This step consists of defining professor preferences and room availability constraints. Optapy then generates all possible professor schedule combinations based on these constraints and assigns each schedule a score. Then it picks the schedule combination with the highest numerical score.
    - A schedule combination’s score starts at 0 and only *decreases* from there. Specifically, a penalty decreases a solution’s score. There are 2 types of constraints that determine the severity of the penalty:
        1. Hard constraint: if a solution violates a hard constraint then it is immediately eliminated, regardless of its numerical score
        2. Soft constraint: if a solution violates a soft constraint it faces a penalty deduction to its score

Overall, I worked on aspects of both steps 1 and 2: enhancing the existing scripts with data validation, as well as a reward function to prioritize schedules that aligned with professor’s preferences with regards to a MWF or TR schedule.

Implementation

Data Validation:

1. Added validation that each professor made a submission
    a. This was fairly straightforward. I simply fetched the list of all submissions and checked that each professor from a fixed list of faculty was included.
2. Validate that sufficient availability was provided
    a. A professor can choose between 3 responses for each timeslot: preferred, acceptable, or unacceptable. Thus at a minimum if a professor is instructing n hours of class in total, then they must have provided at a minimum n hours of preferred and/or acceptable availability. I added this check as well.

New reward function:

- Dr. Beard was considering requesting both a MWF availability as well as a TR availability from each professor which I agreed with. Previously the Microsoft Forms survey just asked for hourly availability and then there were separate questions with regards to whether the professor preferred a MWF or TR schedule (see survey example here). However we decided it'd be better to request dxxxxxxxxxxxxxxxxxxxxxx availability for both MWF and TR and then ask about preferences between MWF and TR as separate questions.
- Thus I worked on implementing a penalty function that would prioritize schedules in which the professor's MWF/TR preferences were met. Specifically my penalty function added a soft constraint for professor schedules to match their MWF/TR preferences. For instance if a professor was teaching 2 classes and indicated that they want to teach 1 class on MWF and 1 class on TR, then a schedule that assigned them 2 classes on MWF would face a soft penalty deduction whereas a schedule that assigned 1 class on MWF and 1 class on TR would not face this soft penalty deduction.

Challenges and Lessons Learned

Since much of the codebase was already in place before my project began, there were definitely some challenges that came up with regards to getting up to speed. Understanding the Optapy code was definitely the most challenging part. I had not worked in the area of constraint solving before so getting familiar with the library by reading documentation, playing around with the code, and asking questions was integral to gaining a better understanding. Overall playing around with the code was helpful but could also be time-consuming and unfruitful at times. Thus determining when and what questions to ask was a skill I improved on.

Part 2: Web portal for better UX

I followed a traditional model-view-controller (MVC) architecture for this application with the following tech stack:

1. Database: PostgreSQL
    a. Postgres is a pretty standard DB option and since I had experience with it before it made sense to work with it again.
2. Backend: Python (Flask)
    a. Flask is a lightweight web framework in Python; I chose Flask given my experience with Python and Flask itself as well as the small scale of this app.
3. Frontend: Next.js
    a. Next.js is a popular React.js-based framework that optimizes UI rendering and thus can greatly boost application performance

Background/Thought Process

Implementation+Contributions

Milestones

1. Auth
    - Using cookies (jwt) for session mgmt
    - Cal poly Saml is ideal but not been done yet
2. Quarter by quarter preferences:
    a. Choose availability + answer some preference questions by quarter
3. Profiles
4. File a bug page

DB Architecture

The DB design required considerable thought as there were several surprising constraints to manage. In the end I decided upon the following tables:

1. users
    - Schema:
        a. id : int (PK)
        b. email : string
        c. password : string -- note: this is the hashed password
2. quarters
    - Fixed list of quarters that we're fetching data for
    - Schema:
        i. quarter : string (PK)

There are 3 types of questions a professor is asked in this survey:

1. daily availability
2. agree/neutral/disagree "multiple choice" questions
3. written "free response" questions.

I created fixed tables to store the questions in each category. Furthermore I created corresponding tables to represent the answers for each category of questions.

3. written_questions -- stores all the free response questions
    a. id : int (PK)
    b. question : string
4. written_answers -- stores all the responses to the written questions for all users
    a. user_id : int (references Users.id)
    b. quarter : string (references quarters.quarter)
    c. question : int
    d. Response : string

    - PK: (user_id,quarter,question)
5. agreement_questions -- stores all the agree/disagree/neutral style questions
    a. id : int (PK)
    b. question : string
6. agreement_levels:
    - Basically just stores 3 categories: “Agree”, “Neutral”, “Disagree”. I wanted a separate table for this relatively trivial data still in order to enforce foreign key constraints in other tables as well as to give us the freedom to add more agreement options if needed (ex. In an earlier iteration of the survey the agreement levels included “strongly agree”, “strongly disagree”, etc.)
    a. category : string (PK)
7. agreement_answers -- stores all the responses to the agreement questions for all users
    a. user_id : int (references users.id)
    b. quarter: string (references quarters.quarter)
    c. question : int (references agreement_questions.id)
    d. category : string
    e. agreement : string (references agreement_levels.category)
    - PK: (user_id,quarter,question,category,agreement)
8. availability:
    a. user_id : int (references users.id)
    b. quarter : string (references quarters.quarter)
    c. day : enum string (“Monday” through “Friday”)
    d. One column for each 30 minute interval from 9 AM to 5 pm.
        i. Ex. the first column is “9 am” and represents a professor’s availability from 9-9:30 AM.
        ii. The professor can select between 3 levels of availability in the UI:
            1. Unacceptable
            2. Acceptable
            3. Preferred

Challenges

Lessons Learned

- Pain points:
    - Currently professors have to manually re-enter their preferences each quarter. It’d be better if they could re-use preferences between quarters and furthermore if they could create re-usable profiles

Backend

API Design

Here is a comprehensive overview of the API for the web app.

All API requests require the following header fields:

1. "Content-Type": "application/json"
2. “Authorization”: “Bearer “ <jwt token>

1. Availability:
    a. /availability:
        i. GET (get availability for a user for a given quarter)
            a. Body params:
                1. quarter: string ex. “fall 2024”
        ii. POST (add/update availability for a user for a given quarter)
            a. Query params: None
            b. Body params:
                1. **Prefs**: List of preferences. Each preference is a json object with the following key/value pairs:
                    1. “day” → weekday as a string
                    2. “time” → any hourly or half hourly time in the range 9 AM - 5 PM
                    3. “preference” → preference level as a string
                    4. Example: {
                        “day”: “Monday”,
                        “time”: “9:30 am”,
                        “preference”: “acceptable”
                    }

2. Answers to written+agreement questions:
   Currently the questions themselves are hardcoded in the front end since they’re fairly static. However if needed they can be fetched dynamically from the backend. Obviously the answers that professors have for the written and agreement style questions will need to be read and written to the backend:
    a. /questions
        i. GET (returns answers for agreement+written answers from a user)
            1. Required query params:
                a. quarter
            2. Optional query param:
                a. scope: either “agreement” or “written” -- allows the user to receive answers for only 1 of these groups instead of both by default
        ii. POST (add/update responses for a user)
            a. Query params: None
            b. Body params:

i. quarter
ii. agreementAnswers: Json list of agreement question ids and agreement levels
ex. {
1: “agree”,
2: “disagree”,
3: “neutral”,
4: “agree”
}
iii. writtenAnswers: Json list of written questions ids and responses
ex. {
1: “sample response 1”,
2: “sample response 2”,
3: “sample response 3”,
}

3. Profiles
The profiles endpoints are identical to the /availability and /questions endpoints. The only difference is that rather than the quarter being passed in as a body parameter, the name of the profile will be passed instead. That being said here are the endpoints for profiles in the backend that correspond to the /availability and /questions endpoints above:
    a. /profile_availability:
        i. GET + POST supported (see /availability)
    b. /profile_questions
        i. GET + POST supported (see /profiles)

    Here are the new endpoints to be aware of for profiles:
    a. /profile_create - Create an empty profile (no data associated with it yet)
        i. Query params: None
        ii. Body params:
            1. “profile”: <profile name>
    b. /profile_clone - Create a copy of an existing profile (copies over availability+responses to questions to new profile)
        i. Query params: None
        ii. Body params:
            1. “New”:
    c. /profile_delete - Delete an existing profile and all data associated with it
        i. Query params: None
        ii. Body params:
            1. “profile”: <profile name>
4. Auth

These endpoints are temporary -- currently for all of them the user provided password is passed as plaintext to the backend, where it is then hashed. Once SAML auth is completed in the future this will be fixed.

a. /register
    i. Query params: None
    ii. Body params:
        1. “email”: <user email>
        2. “password”: <user password>
b. /login
    i. Query params: None
    ii. Body params:
        1. “email”: <user email>
        2. “password”: <user password>
c. /api/verify_user
    i. Body params:
        1. “email”: <user email>
        2. “password”: <user password>
    ii. Query params: None
    iii. Body params:
        1. “email”: <user email>
        2. “password”: <user password>

Frontend

I used several frontend components, taking advantage of their re-usability when possible. Each component is in its own separate file under `src/pages`. The most relevant pages are the /preferences and /profiles pages. Below I describe their implementation and underlying components:

1. /preferences
    a. This page is the meat of the app, where professors input their availability and answer the agreement and written questions. It uses the `Quarter_Availabiliity` component to display the availability table and the `Agreement_Questions` and `Written_Questions` to display the questions.
2. /profiles
    a. This page is where the user is presented with the 4 profile options: create, edit, clone, and delete. The clone and delete pages are very minimalistic, whereas the create and edit pages re-use the availability and question components to present an identical UI to the /preferences page.