FitForge Allocation System v1.0 Report

T039

OLIVER PINEL – ALEXANDER KEARNEY – RAPHAEL FAHEY – CALUM INGRAM

Abstract

Over the past five weeks, our team has developed a React web application designed to assist the teaching team of IFB398 in optimizing the allocation of student teams to available projects. Building on the research conducted last semester, where we defined the goals and requirements of such a system, we made strategic decisions in UI layout, algorithm selection, and user flow design to create a tool that meets these requirements and ideally improves on the performance of previous systems.

This report provides a comprehensive overview of the system, including the goals and scope of the project, the architecture and key features of the web app, and the design rationale behind our choices. It also details the system's functionality, focusing on frontend-backend interactions, flow and state management, and the intended user process. Each section aims to document the system's development, justify the decisions made, and demonstrate how the project goals were achieved.

Table of Contents

Introduction

As we developed the system, it was essential to adhere to the considerations defined when beginning our research of the program, and how we would meet the requirements derived and attempt to meet the criteria in order to make a system that would be a good fit for the situation. As such, this report will begin by clearly defining the project goals and scope, the requirements and considerations behind the choices made, as well as the definitions incorporated into this report.

It is prudent to define some terms used throughout this report for simplicity. First, a “pairing” is one potential combination between a team and a project, characterised by the overall b value formula, based on the compatibility between team and project. An “allocation” is a confirmed pairing, one which will be finalised and used in the upcoming semester. A “removal” is the removal of a pairing from the allocations, in the event that it becomes undesirable, or less optimal. A “rejection” is a disqualified pairing that should not be considered any further in the process. Finally, an “allocation set” is a list of pairings generated by an algorithm as a potential finalised allocation list.

Three main considerations were the focus of the decisions made when developing the system. The system should ensure that the allocation process is quick and simple, reducing the manual efforts in manipulating data. This was the biggest focus for our development, and we aimed to address this by providing interfaces to streamline the process of comparing allocations as best as possible. Additionally, we considered the accuracy of the allocation, in terms of how well the teams had projects allocated. Whether they had a high b value coefficient in the pairing, whether team preferences were respected, and how this could be emphasised in the workflow.

Finally, the human element is the most important aspect. Not only must a human make the actual allocations, only using an algorithm as an assistive tool, they must do it one by one, in a naturally greedy-like allocation flow. While this is undesirable, as pairings have to be manually allocated it is inevitable that without significant effort the allocations will be inherently greedy. As such, a variety of UI elements and a connected flow are the best attempts for this system to allow for a wide comparison to each allocation, in order to improve the overall accuracy and satisfaction, and balance the fairness of the process to each group.

For this system, our team has access to the testing b values that were used in the previous semesters system. In addition, we are expecting to only work with these. While there are more factors involved in each pairing, these were condensed into the b values, which while limiting full understanding of each pairing provides enough details to characterise and distinguish them from each other. Due to the privacy of data, our team has developed a system focused only on providing UI and algorithmic ways to present different pairing combinations, in order to assist the user in making allocations.

System Design

Overview

As described in our report from last semester, our team has developed a system that combines an assistive algorithm with a variety of UI options to provide the user with choice and streamlined navigation. We have developed using the React framework a web application that can be accessed through a web browser, allows the user to upload the student data, and from there traverse through the different views to determine the best allocation for each team to each project. The data is stored locally, none of it being held on the server, to increase security. Through proper implementation of React's features, the backend of the system was connected to the front, allowing for data storage and manipulation, and the proper handling in the various representations. In addition, the system features the ability to save and load the current state, allowing for the user to pick up from where they left off.

React

The React framework has two main features and one consideration that make it an appropriate choice for development. The framework is designed to develop small "components" which are reusable and can be stacked easily within themselves, leading to a very modular style project. Due to the nature of many different UI elements, that only need to access data and not each other, the modular nature of the framework makes it simple to develop and test a wide range of UI elements independently of each other. In addition, React exposes hooks to variables and states in each component. This allows for each component to not only be highly interactive with the user, but to store and update data as it changes from others. The simple nature of the react hooks, makes it easy to connect components to the data, and with a combination of useState and useEffect, to reload data from the backend on changes. Finally, our team all has previous experience in React, making it the easiest choice to develop this prototype system for proper testing of our understanding.

Currently, we have implemented the react app as a web application. This is beneficial as it allows for easy testing in any web browser, locally or on any device on the network. The lack of system requirements allows for the focus of development to remain majorly on the functionality and success of our implementation, rather than specific optimisations for a system. JavaScript as a language is widely accessible and runs on virtually every system, meaning that our tests will not need any specific requirements.

In addition to the availability of testing the system, in future iterations it is possible to combine our react web app with electron, a program designed for converting web applications into desktop applications. This could port our system into an OS neutral software, in the case which we wish to explore more subwindows or showing multiple views at once. Additionally, it is simple to port a react app into a webserver due to the ability to build the program into pure JavaScript, CSS and HTML. As such, by using React we can focus entirely on testing the success of our system's implementation, and later move to determining how the final product should be presented.

Backend

In order to store and handle the data operations of the program, a singular class was created to manage the state and operation of the system. CoreService, as its name makes clear, provides the core functionality of the program. This class provides the storage variables for the different data options and controls, as well as the functions for the different operations of the program.

The CoreService class contained several private variables. These variables stored the initially uploaded data, the modifies to the number of teams each project could be allocated to, as well as any calculated values through the process of working. In addition, the class managed the current state of the program, ensuring the user could not access functionality before it was intended to be used. The variables stored the allocations as they progressed, and any algorithms run, for them to be referenced later.

For the functionality of the data access and manipulation, there were four main categories of functions exposed for the components to make use of. Initially run will be the setup functions, which reset the data to the default conditions, read and store the input and calculate any basic requirements for operation of the system to proceed. These functions will set the state of the program as necessary and allow the user to reset the data should major mistakes be made. Finally, these functions also implemented the ability to save the state of the program and load it from a save file.

The second, and largest, class of function was the data access functions. These were made available for nearly every kind of property stored in the class, and provided functionality for returning not just raw data, but information about the state of the program, the overall score of the allocations, the background colour for a div depending on the b value being represented. These also included functions to update variables the user would be allowed to change.

The third class of function was the allocation handlers. These functions implemented the logic behind allocating a pairing; by ensuring they aren't already paired with something else. In addition, they implemented the removal logic, and other functions required to manage if something could be paired.

The final class of function was the algorithm connector. This only involved two functions, that uniquely defined the data types required by the two implemented algorithms, Integer Linear Programming and Gale Shapely matching, to pass the data to them, and return a proposed best fit that could be stored for later use.

In order to properly expose this data to the other components for use, an instance of the class had to be initialised in a react context. This allowed for one singular instance to persist across the lifetime of the program, ensuring that the same data could be stored and accessed by all the components that were children to the context. It subscribes any children props to this instance, keeping it from garbage collection and ensuring that the public functions can be universally accessed. This allowed for the individual modular nature of the program to not be impacted by API constraints.

The major flaw with this implementation, was that while it functioned extremely well in exposing a variety of options for individual components to be developed, as the system and

scope expanded the class became much larger than initially expected, making it more complicated than expected. And yet, due to the integreal nature of it, it could not easily be replaced, becoming one of the issues that had saught to be avoided by utilising react. As such, when it came time to implement the navigational functionality, a second context was created, instead of building on top of this class even further.

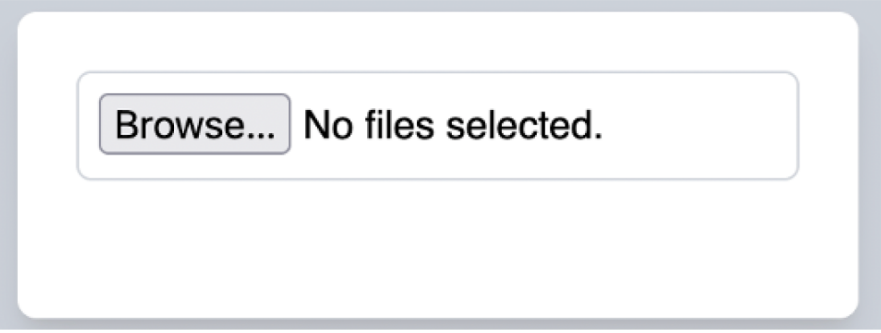
For navigation, the App.tsx file was responsible for rendering the major UI elements of the program. This was done by checking the state of the program to ensure the user would not progress beyond where they should be, and then reading the current page from the navigation context to be rendered. The navigation context exposed one function and three properties to the components for use. The navigate function was able to be called by a component that existed as a child of the navigation context, and when provided with a page and data, would cause App.tsx to switch which UI component was rendered to that. The three properties, history, currentPage and currentIndex, allowed for a component to both identify which page was currently showing, and build a history of the pages visited, for the user to more easily traverse between them.

Frontend

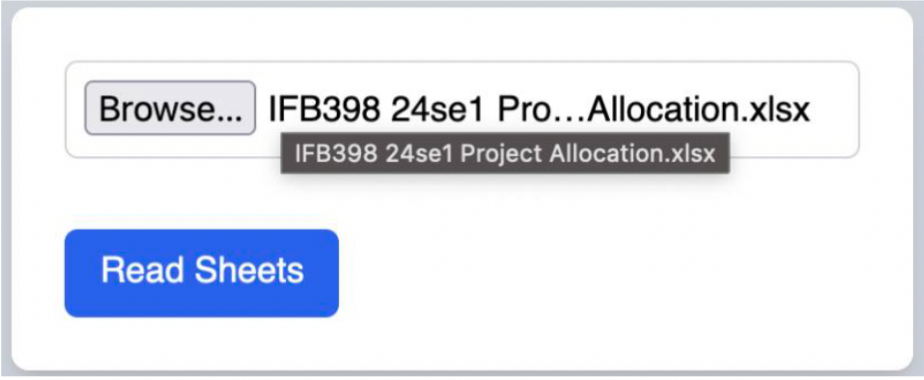
As has been explained, our system involves several, mostly unrelated major UI implementations. Each is meant to handle a specific part of the system, be it processing or rendering data, in different ways for the user to parse it. The goals of these elements are to increase the simplicity for the user to view and compare the pairings available, and we have designed them as such. Each view is designed to be simple, lacking deeper complexity, but it is hoped that the combination of them will produce a system that allows for a wide range of comparisons between data, and to reduce the need for complete greedy allocation.

Input Component

The user initially has all of their data within an excel sheet, or multiple excel sheets, thus an efficient method to collect this data and process it into a format the matching algorithm is able to use is essential. The Input Component was created to streamline the uploading and processing of data from multiple Excel Sheets into a single dataset that is able to be used by the matching algorithm in the application. The user is initially displayed with a simple user interface that prompts them to upload an excel sheet by pressing a button “Browse”. A button was chosen as the user may or may not be technically inclined, thus a minimal UI with a clear action that provides immediate feedback was chosen as to minimize confusion for any potential user.

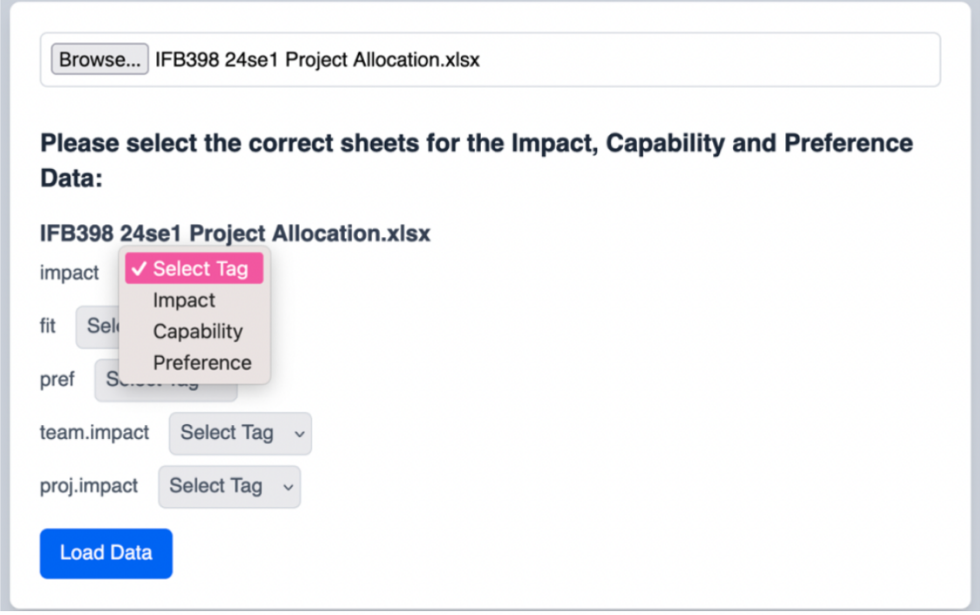


Once the button is pressed and the user selects their excel spreadsheet/s, the handleFileChange checks if files are selected using ‘e.target.files’. If a file is found to be selected, the ‘FileList’ object is converted to a Javascript Array with ‘Array.from(e.target.files)’ which will allow for more fluid manipulation of the data within Javascript. ‘setFiles(selectedFiles)’ updates the component state, which in turn will trigger a re-render to immediately show the user the selected files in the UI. This automatic updating and re-rending of the UI to the user enables a smoother, more intuitive user experience by dynamically updating the interface and providing immediate feedback after each action the user makes, and this was the primary reason for storing uploaded files in the component state, along with the encapsulation of data management within the Input Component.



When the user selects the ‘Read Sheets’ button on the User Interface, the loadSheetNames function is called, creating a promise for each file using ‘readSheetNames(file)’, retrieving all the names of the sheets in the file. The objects ‘newSheetNames’ and ‘newSheetTags’ are initialized store the sheetNames for each file and to initialize empty tags for each sheet. Using ‘setSheetNames’ and ‘setSheetTags’, these objects are stored in the component state. Objects were chosen for this functionality as they are an efficient way to map the file names to their respective sheet names and tags, allowing for simple management and organization of the data.

The user is prompted to select the correct sheets for the Impact, Capability and Preference Data, with a dropdown menu being immediately displayed to the user on the user interface, detailing a list of all the excel files uploaded, their corresponding sheets, and the ability for the user to assign the appropriate tags to each sheet (‘Impact’, ‘Capability’ and ‘Preference) they are labelled with clear titles, keep the user interface organized and uncluttered and prevent user error by restricting options to a predefined set.



The user then interacts with the ‘Load Data’ button, highlighted in a blue colour at the very bottom of the Interface to ensure users fill in the dropdown menu before attempting to load their data. Pressing this button immediately calls the loadData function, which will begin by creating an array of promises, with each promise reading data from a sheet with tag ‘Impact’, ‘Capability’ or ‘Preference’ utilising the ‘readFromExcelSheet’ function which is further explained below. The results of these promises are processed into arrays corresponding to their respective tags. Arrays were chosen as the data structure due to maintaining the order of elements, which is crucial for this application as the data must be processed in a specific sequence due to each data point corresponding to both a specific team and project. Following this, the loadData function collects the ‘teamNames’ and ‘projectNames’ from the ‘Impact’ data. A Setup object is then created that initializes all relevant data in the core service, updates the core service’s data stage and navigates to the next page.

```
  const setupParams: Setup = {
    impact_vals: impact,
    capability_vals: capability,
    preference_vals: preference,
    team_names: allTeamNames,
    project_names: allProjectNames,
  };

  coreService.initialise_values(setupParams);
  coreService.dataStage = "Stage2";
  navigate({ page: "TeamsToProjects", data: null });
});
```

The ‘readFromExcelSheet’ function extracts the data from the specific sheet it is being called to run on, along with team names and projects names, based on the provided tag (‘Impact’, ‘Capability, ‘Preference’), the function then stores this data into arrays. The function returns a

promise that resolves with an object containing a 2D array of numerical values extracted from the sheet, the tag ('Impact', 'Capability, 'Preference'), an array of team names and an array of project names. All of this data can then be used by the 'loadData' function to correctly prepare the data for future processing by the algorithm and passing it to core service.

```
resolve({ data: dataArray, tag, teamNames, projectNames });
```

List view

The ListView component accepts a title and an array of pairings as props, where each pairing is represented by a [team, project] tuple. The component uses the coreService to fetch additional data for each pairing, which is then stored in the pairingData state.

Key features of the ListView component include:

- Sorting Functionality: Users can sort the list based on various properties such as team, project, and several scalar values. The sorting order can be toggled between ascending and descending.
- Expandable Items: Each item in the list can be expanded to show more details. The state of expansion is managed by the expandedIndex state.
- The use of hooks ensures that the component is reactive and updates whenever the pairings prop changes. The sorting mechanism is straightforward, using JavaScript's array sorting methods, and the expandable items are implemented with conditional rendering.

Why a List View?
The decision to use a list view was driven by the need for a clear and organized presentation of pairing data. Given that users might need to browse through numerous pairings, a list view allows for a scalable and easily navigable structure. The grid layout adapts to different screen sizes, enhancing the user experience on various devices.

Sorting and Filtering
Users often need to analyze data based on different criteria. The sorting functionality provides flexibility, allowing users to view the data in the order that best suits their needs. By enabling sorting on multiple properties, the component caters to diverse use cases, from quick overviews to detailed analyses.

Expandable Items
The expandable item feature serves two purposes:

- Space Management: Only essential information is displayed initially, keeping the interface

uncluttered. Additional details are revealed on demand.

- User Engagement: This interactive feature invites users to explore the data more deeply,

enhancing their engagement with the application.

Reactivity and State Management
The use of hooks such as useState and useEffect ensures that the component is responsive to changes. This design choice aligns with modern React practices, promoting a clean and maintainable codebase. The asynchronous data fetching inside useEffect allows the

component to handle dynamic data gracefully.

Goals and User Interaction
The primary goal of the ListView component is to present complex pairing data in an accessible and user-friendly manner. The component is designed to:

- Facilitate Analysis: By allowing users to sort and expand items, the component helps users analyze data more effectively.
- Enhance Usability: The grid layout, combined with sorting and expandability, ensures that the component is both intuitive and efficient.
- Adapt to User Needs: Whether the user wants a quick glance at the data or a deep dive, the ListView component is flexible enough to accommodate both scenarios.

Users are expected to interact with the component primarily through sorting and expanding items. The design encourages exploration, with sorting providing a tailored view of the data and expandable items offering additional insights without overwhelming the user.

Spreadsheet

The Spreadsheet View is core component of the FitForge web application. This is designed to present data in a structured tabulated manner like an Excel Sheet. The Spreadsheet View is implemented with various hooks and context services to manage state and interactions. Some key features include data display, cell interaction, and dynamic styling.

Key Components:

- useState Hooks:
  - displayType: Manages the current display mode (“Numbers”, “Colours”, “Both”, “Assigned”)
  - isModalShown: Tracks whether a modal window (for pairing details) is visible
  - modalTeam and modalProject: store the team and project IDs for the currently selected cell
- useCoreService and useNavigation Hooks
  - coreService: Provides access to core data (number of teams, projecta and cell specific data)
- Even Handlers:
  - handleCellClick: triggered when a cell is clicked, displaying pairing data in a modal window
  - setIsModalShown: Toggles the visibility of the modal
- Rendering Logic:
  - The table is dynamically generated based on the number of teams and projects, with each cell displaying data or colours fetched from coreService

How it Works

The SpreadsheetView component functions as a dynamic table where each row links to a team, and each column represents an available. Inside each cell, a best fit number is displayed, which is colour coded from best to worst (green to red). When a user clicks on a cell, a window appears which identifies what team and project were selected. This window provides more in-depth information about that pairing, offering users a closer look at the data without cluttering the main view, this improves user efficiency and experience. The design

also includes a fixed header and various UI features such as a changing colour when hovering over cells, so as you scroll through the table, the project labels remain visible at the top, making navigation through large datasets more manageable.

Justification for the Design
The design of the SpreadsheetView component was carefully thought out to ensure it could handle large amounts of data efficiently while still being user-friendly. Flexibility is at the core of this design where the table automatically adjusts to the size of the dataset, whether you're dealing with a few teams and projects or many. This adaptability is crucial for users who need to work with varying amounts of data without fuss. The spreadsheet also resembles that of a Google or Excel Sheet which enhances user familiarity allowing for an ease-of-use. User interaction is also a key focus. By allowing users to click on a specific cell to open a window with more detailed information, the design keeps the main interface clean, while offering depth where needed. The component's ability to switch between different display types, like numbers or colours, gives users the freedom to view the data in the way that best suits their needs.

Layout of the Components
The table is the centrepiece, organized with project headers at the top and team names along the side, making it easy to find and compare data. When you click on a cell, a modal pops up over the table, giving you more details about that team and project pairing without disrupting the main view. This structure table ensures that everything you need is within reach, making the component not just functional, but also intuitive to use.

Goals for Spreadsheet View
The SpreadsheetView component is designed with several goals in mind. First and foremost, it’s meant to make managing and interacting with large datasets as easy as possible by adjusting the size depending what data is input. Whether dealing with complex team-project assignments or a simpler data set, the application is built to handle it efficiently. The design intentionally mirrors familiar spreadsheet tools, making it easier for users to get up to speed and work effectively. Additionally, the interactive elements, like the clickable cells that bring up modals, are designed to support quick decision-making by giving users immediate access to the details they need, right when they need them. All in all, the SpreadsheetView system aims to be a powerful, user-friendly tool that helps users manage their data with ease

Pairing details

The overall goal of the system is to provide the user with options to sort through possible pairings and allocate as good a set as possible. To this end, the ability to compare a specific pairing is the most important aspect of this. It must be quick, visually obvious and easily distinguished. To that end, after several iterations of design, the current layout was chosen. It was determined that it should be essential that the team and project be identified easily and quickly, being prominent in the layout. Additionally, the view needs to be able to share the preference data, the capability data and the b values for the pairing. The pairing should have navigation options to view the other pairings for that team, and other pairings for that project, to ensure that they can be properly compared. Finally, the pairing needs a way to confirm the allocation of it or reject it from the pool. As such, the following design is the final version.

In order to provide a visual signifier, the background colour for each pairing is a gradient scaled from the minimum b value to the maximum converted into a ratio of red and green, giving a quick way to visually discern multiple pairings. The relevant comparison data is outlined in the centre row, with the options for navigation and allocation are clearly visible on the bottom.

This view will be the central focus of the functionality of the program, being propagated through most of the other UI elements, and its functionality will be of high consideration in the testing phases, as we look to further refine it.

User Process

The following is an outline of how we expect the system to be used. First, if not already obtained, compile the system using the ‘npm run build’ command. This will require an installation of NodeJS and npm, along with the packages used in the system, installed with `npm install`. This produces in the dist directory a compiled HTML, Javascript and CSS file, which can be served with any webserver hosting software. For our instance we are using the npm program ‘serve’. Once the web app is hosted, it can be loaded in a web browser using the IP address of the host device. In addition to the built program, the user will require an excel spreadsheet in the expected format, containing three sheets. Each sheet must be m+1 x n +2, where m is the number of teams and n is the number of projects. The titles of each team must be down the first column from row two onwards, and the titles of each project must be along the first row from the third column onwards. The data must be stored in the section defined by the teams and projects.

On the first screen upon loading the program, the user can choose a single or multiple excel file to upload. The system will read the names of the sheets, and provide a select form, to identify which sheets correspond to each dataset. Once they are selected, a button will appear to load the data into the system backend, storing it and allowing the user to move onto the next stage of input.

One of the features developed that has not been mentioned yet, is a UI page to determine the number of teams which can work on each project. This form has no requirements to be filled out, and is simply an integer input for each project, defaulting to one. If this is acceptable, then the user can progress to the other options in the navigation bar.

Next, the user can optionally choose to scale some of the data, either the capability or preference data. This can allow for more control, and less concern over the incoming data, whether the preferencing is in another format to the capability.

From here, the main allocation process begins, as the user will utilise the four different kinds of list view, as well as the spreadsheet, the allocations tab and the rejections tab. Each page will show a different method of considering and comparing the data and using either the buttons on the navigation bar or the navigation buttons on each pairing the user can carefully consider each team one by one, compare their best options and quickly determine whether these options will impact the allocation, before selecting the allocation options to confirm it.

As the user progresses through, due to the greedy nature of the allocation method, inevitably the options will reduce to being less optimal and harder to discern which ones will impact the remaining pool the most. In this case, it is intended for the algorithms implemented to be used as assistive tools. The user can, based on the current allocations and rejections, run an algorithm to generate a suggested allocation set based on the current choices. This algorithm will be the numerically optimal mathematically perfect calculation based on the b values of the remaining choices. This is intended to provide a baseline for the user to work off, as a way to reduce the bad options to the best possible that they can then consider and justify further.

Once all the allocations have been made, the process is over, and the allocations can be viewed and copied out of the Allocations tab in the system. This concludes the expected process the user would take when using the program. Admittedly, no testing has been done on anyone uninvolved with the development, and these preconceived notions will likely impact our understanding of the UI. For this reason, external testing by a real user is an opportunity that will likely have a large impact on the plans for this project and our understanding of the goals overall.

Testing

During development, this system had two main testing phases. Initially, as we developed the system, we did not have access to the official data. As such, using python, randomly generated datasets in the expected format were generated. While this data was not from a real sample, as we knew the expected data structure it was possible to do accurate testing on the functionality of the system. This ensured that there were no unexpected problems, and full run throughs on each component could be properly performed.

Despite this, due to time constraints, the system was not designed to be foolproof, and as such, many errors and other bugs were handled in simple manners, either simply displaying the error for the user, or logging it to the web console. This allowed us to identify and fix many bugs, but the focus during development was not on this aspect. In future development, depending how the next testing phase progresses, this will be a more significant focus, to produce a robust system that can function without errors that require in-person handling.

The second testing phase is a “real” test, using the data for this semester’s actual IFB398 team allocations, to get a better sense of how well the system meets the use-case requirements for IFB398. While we have attempted to define requirements and considerations and built upon those for the main focuses for each stage of development, our understanding will naturally differ from the intended users. To that end, a recording of the system being used properly will be made by our Industry Partner, and in studying that recording, we will aim to identify the key components of the process, redefine the main focuses for the purpose of the system, and refine our system overall to better meet the goals of this project.

Future Plans

Our team, prior to the results of the testing, has two main areas of focus we believe would be the most beneficial to explore in the future. While the development of the system has made it clear that our understanding of the project could certainly be expanded through more research on the wide topic of allocation, there were two focus areas that we intended to improve upon.

Notably, one of the aspects we intended to include in this version of the system was a more graphical experience. Whether this was a pie chart, a Gantt chart showing the burndown and time of the program, or some kind of project vs team comparison graph, the simplicity of a symbolic display is in line with the ideals behind the other UI choices. Due to time, a lack of understanding as to how it would be represented, and most significantly the greater difficulty in parsing large data in a chart format compared to our pairings and allocation sets, this feature was dropped. In the future, it would be a beneficial addition, to provide a comparison or contrast to the large number of numbers visible on screen, making comparisons simpler, even if it is harder to sort through the large sum of data in only one or two charts.

Additionally, one of the goals of this system was to reduce the inherent greedy nature of the allocation. While as of this time our understanding is that it is impossible to entirely remove the greedy nature by going one by one for allocations, an expansion as to the flow of comparisons and allocations would be worth exploring. Potentially by reworking how the system recommends pairings, or how the system flows through with allocations and rejections, the individualistic greedy nature of each allocation can be reduced by comparing the impact of possibilities somehow.

In summary, once the testing for this system is completed, and the comparison between this system, as well as the previous method can more accurately identify the major factors in how allocation should work, we can review the requirements and main metrics that should be targeted with a system and perform more research to properly define how a system to accomplish allocation best can be designed.