Field Session Website for Dr. Painter-Wakefield

Team: CSM Painter-Wakefield

June 15, 2020

Contents

1 Introduction **3**

2 Requirements **4**
2.1 Functional Requirements . . . 4
2.1.1 Website Requirements . . . 4
2.1.2 Team Creation Requirements . . . 4
2.2 Non-Functional Requirements . . . 4
2.2.1 Website Requirements . . . 4
2.2.2 Team Creation Requirements . . . 5
2.2.3 Database Requirements . . . 5
2.2.4 Stretch Goals . . . 5

3 System Architecture **6**
3.1 Design Specifics . . . 6
3.1.1 Backend Architecture . . . 6
3.1.2 Database . . . 8
3.1.3 Team Selection Algorithm . . . 9
3.1.4 Frontend Architecture . . . 10
3.1.5 API . . . 11

4 Technical Design **12**
4.1 Algorithm Design and Approach . . . 12
4.2 Genetic Programming . . . 13
4.3 Nomenclature . . . 14
4.4 Implementation . . . 14
4.4.1 Seeding . . . 14
4.4.2 GP Implementation . . . 15

5 Quality Assurance **16**
5.1 Frontend-Backend Integration . . . 16
5.2 Backend-Database Integration . . . 16
5.3 Algorithm Considerations . . . 16
5.3.1 Constraints . . . 16
5.3.2 Avoiding Other Teammates . . . 17
5.3.3 Timing and Optimization . . . 17
5.4 Code Quality . . . 17

6 Results **18**
6.1 Summary of Testing . . . 18
6.1.1 Frontend . . . 18
6.1.2 Backend . . . 18
6.1.3 Database . . . 18
6.1.4 Algorithm . . . 18
6.2 Lessons Learned . . . 19
6.3 Future Work . . . 19
6.4 Features Not Implemented . . . 19

7 References **20**

8 Appendix **21**
8.1 Website Pages 21
8.2 Website Pages, Mock-ups 22

9 License **23**

1 Introduction

Currently, the Colorado School of Mines Computer Science Department (CS@Mines) is using a closed-source web application called Connect, deployed at `connect.mines.edu` to manage student teams and projects for CSCI370 (Field Session). Unfortunately, Connect contains bugs that its administrator cannot fix because it is closed-source. Every year, when it is time to assign teams for field session, a non-negligible amount of time is spent manually working around Connect's issues, primarily with automated team selection. This project, Reconnect, is a new, open-source web application handling the current functionality of Connect with modern practices in place.

The frontend of Reconnect is a website consisting of a login page, a team creation tab, a projects page, and administrator pages for each of these purposes. The administrator can add students and projects both in bulk from a file and one by one. Students whose information is in the database can log in to the site and enter their information for choosing a project and team.

The administrator then can create teams on the site by running a team selection algorithm and/or manually editing teams. The team selection algorithm creates teams based on user-input information, such as desired project, preferred teammates, and teammates to avoid. This algorithm shall be accompanied by a web application that shall serve as an access portal for students alongisde administrators.

Figure 1: Team creation page

2 Requirements

2.1 Functional Requirements

The three main components of the project are the website, the team selection algorithm, and the database.

The website shall provide information about Field Session to all users. When logged in, based on account permissions, users shall either have a "student view" or an "administrator view", from which they may access appropriate pages and interfaces. The team selection algorithm shall be usable by the administrator via a user interface on the website. The database shall contain the necessary information about teams and projects for the website to fulfill its requirements.

2.1.1 Website Requirements

- Students shall not have the ability to edit their profile information after the team creation algorithm has started running.
- Users shall be able to log in using Multipass via Shibboleth or Google OAuth.
- The administrator shall be able to add users by uploading a list of names and email addresses.
- The uploaded name and email list shall be properly formatted JSON.
- The administrator shall be able to add individual users by name and email address. He or she may do this through file upload, or one-by-one via text entry.

2.1.2 Team Creation Requirements

- The administrator shall be able to specify different minimum and maximum team sizes for each project.
- Students shall not receive teammates they have requested to not work with.
- The website shall have a page from which the administrator may edit teams using a graphical user interface.

2.2 Non-Functional Requirements

Given that this project aims to create an accessible web application, there are several considerations to be made for how the project will process and store user data. Moreover, given that this project aims to create an algorithm as well, that algorithm must be designed with certain constraints in mind, as specified below. Also noted is how the application behaves in contrast to how the user interacts with it.

2.2.1 Website Requirements

Some pages are required for minimal functionality. This includes a login page; a student profile page for project and team preferences; a Projects page for team assignments; and an administration page where team configuration may be made.

Key implementation details include:

- The website shall be created using JavaScript, HTML, and CSS.
- The website software shall have an open-source license.

- There shall be two types of users: administrator and student. Admins may edit projects and all user information, as well as use the team creation feature, while student users may only edit their own information.
- The Projects page shall be editable without deleting all projects being necessary. Currently, the administrator has to completely wipe the page and re-upload a file to edit projects and users.

2.2.2 Team Creation Requirements

- The website shall automatically generate student teams based on user-input criteria. This information shall be stored in the database and shall be viewable through a page on the website.
- Teams shall be restricted to a specified size.

2.2.3 Database Requirements

- The database software shall be PostgreSQL.
- The database shall be password-protected.
- The parts of the website interacting with the database shall be protected against SQL injection.
- The database shall store information sufficient to determine which teams have which members, and which project each team is assigned to.
- The database shall contain the names and bounds on team sizes of each project.

2.2.4 Stretch Goals

While not absolutely necessary, some stretch goals may be considered for this project. These may include making the website look “pretty” per good UI design, or allowing students to edit their profile pages after initially submitting them.

3 System Architecture

3.1 Design Specifics

3.1.1 Backend Architecture

This web application uses the standard client-server model. It consists of a PostgreSQL database, a NodeJS backend framework, and a VueJS frontend framework which uses ExpressJS to handle requests on top of NodeJS, and Vuetify to render the web pages on top of VueJS (see Figure 2). This combination of frameworks results in an extremely memory-efficient single-threaded application. This system aims to reduce memory transfer latency between devices at the cost of being single-threaded, meaning that the web server may experience delays in computationally intensive operations. (This excludes the team-generating algorithm, however, which runs in a separate thread).

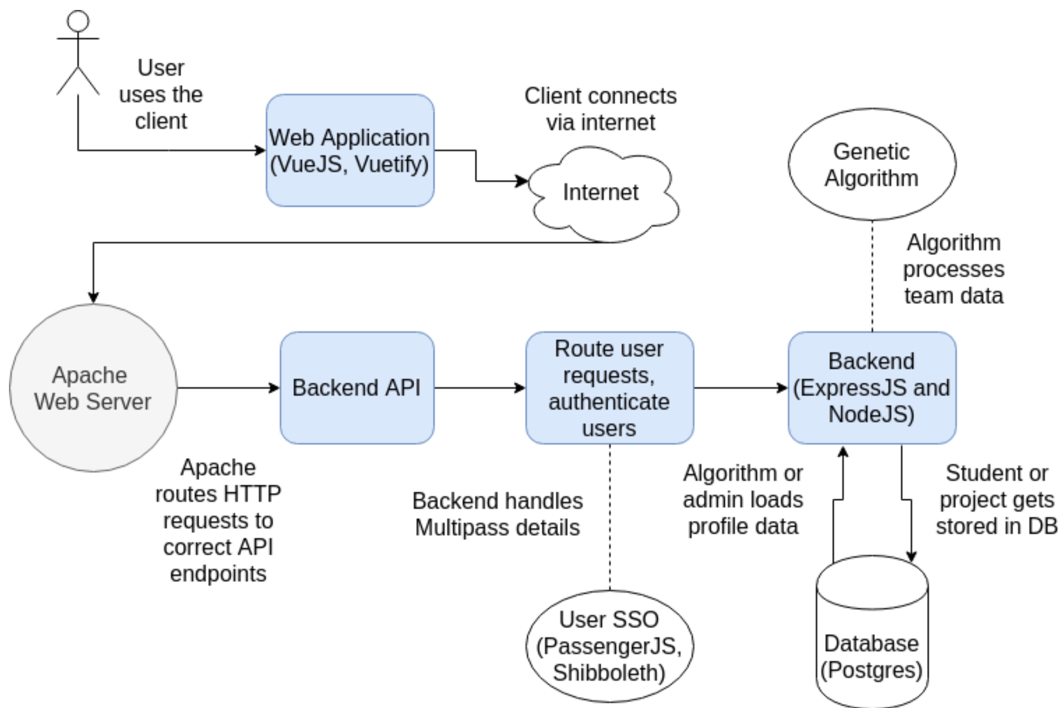

Figure 2: Project architecture

In Figure 2, the user accesses the client, which communicates with the backend by sending API calls to ExpressJS and NodeJS. Depending on the API call that the client makes, as well as the authorization that the client has, different responses may be given from the server.

As needed for the behavior in question, the backend may access the database, utilize the created algorithm, or handle authentication and authorization per user (illustrated in Figure 3).

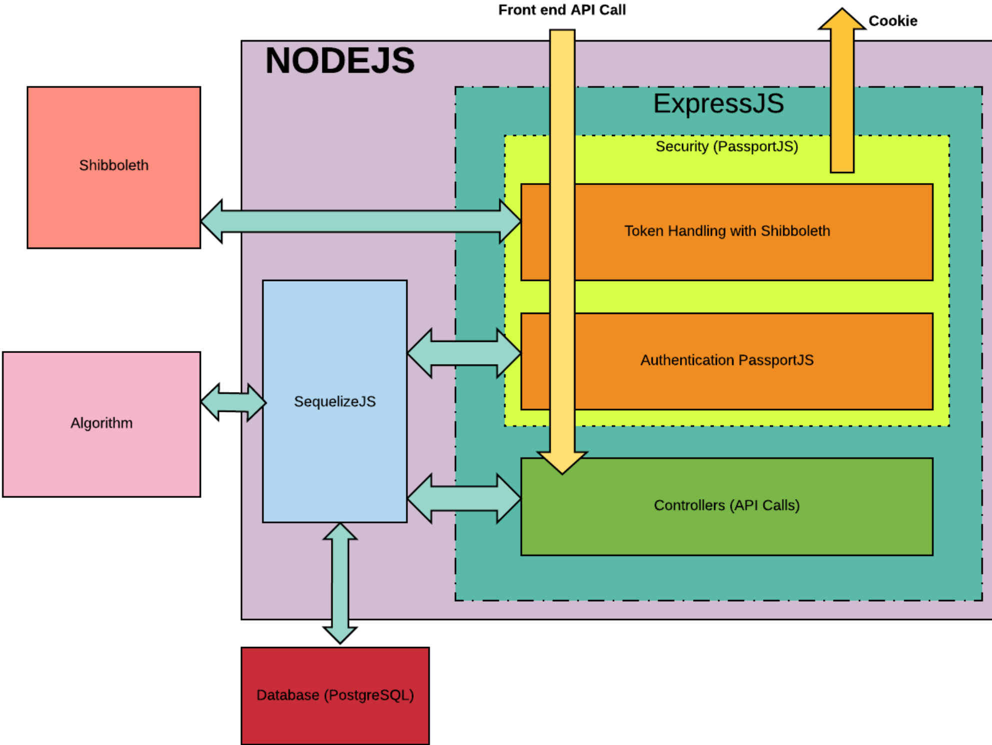

Figure 3: Components of the backend.

Other things to note is that, while the "backend" is indicated as the big purple square in the diagram above, the backend of the team's application actually includes the algorithm as part of the backend, as the algorithm runs directly on NodeJS. However, it is separate here for the sake of making it easier to identify. Similarly, it is also in a separate directory of the backend, providing a mental difference akin to the Database or Shibboleth components.

More on the database and algorithm in specific will be provided in diagrams on the following pages.

3.1.2 Database

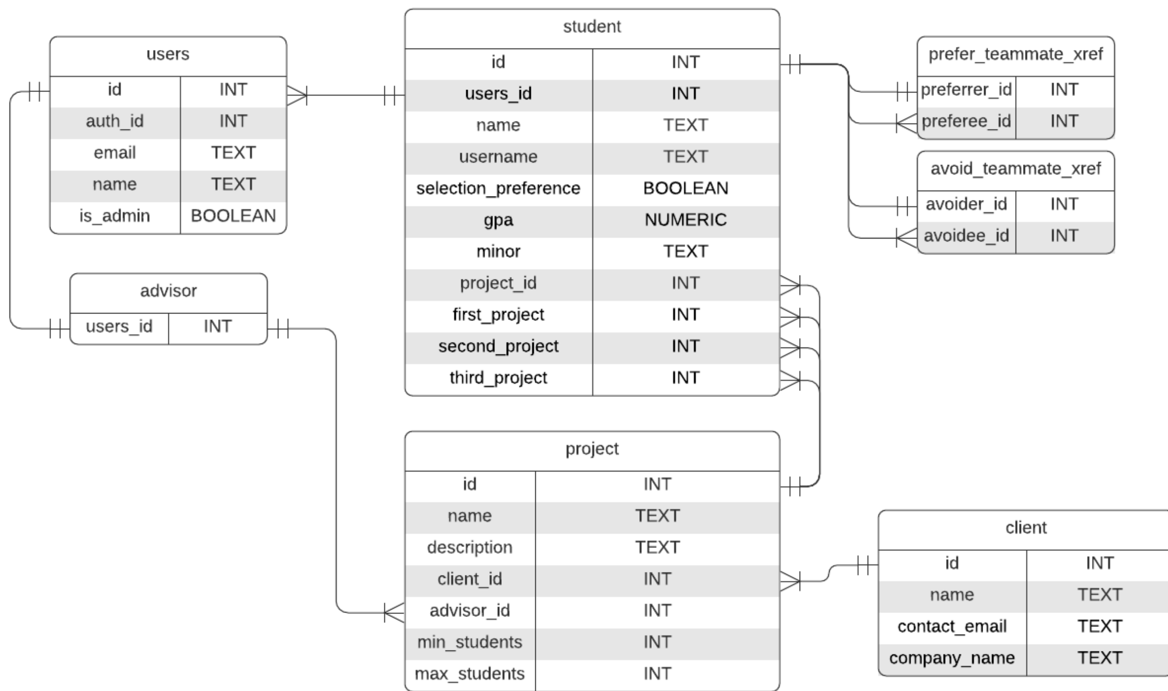

Figure 4: Relationships between tables in the database. (Diagram uses Crow's Foot notation.)

The database uses the relational model and contains the basic information about each project, the client it belongs to, its advisor, and its team members. Figure 4 illustrates the schema. Student preferences for the team selection algorithm are also stored. Each row of the `prefer_teammate_xref` and `avoid_teammate_xref` tables corresponds to a pair of students. The first of the pair (`preferrer` or `avoider`) is the student whose profile the entry was retrieved from. The second of the pair (`preferee` or `avoidee`) is a student the first of the pair requested to prefer/avoid having as a teammate.

3.1.3 Team Selection Algorithm

The team selection algorithm is written in JavaScript. A visual representation of the algorithms steps is shown in Figure 5. More details of its design are in the **Technical Design** section.

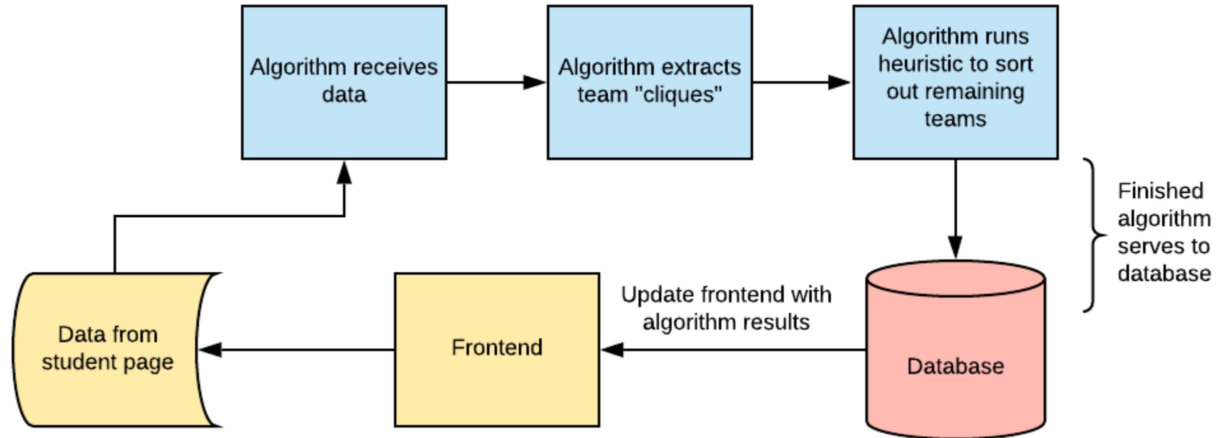

Figure 5: Steps taken by the algorithm.

Originally, we were considering using an integer programming (IP) algorithm to generate teams. However, it initially appeared more realistic to incorporate a simulated annealing (SA) algorithm. SA is well designed for discrete problems where a global maximum is necessary. Ideally, we want to maximize satisfaction of students and teams with this algorithm, so SA appeared ideal.

Of note, however, is the fact that this team sorting problem is $\mathcal{NP}$-Hard. After a series of decisions outlined in the **Technical Design**, we settled on a genetic programming (GP) strategy for sorting teams alongside a greedy algorithm to aid generational seeding (more on that in **Technical Design** as well).

Following the diagram above, the entrypoint is on the square marked "frontend," which is where a student may interact with the service. Their data is aggregated, and the algorithm utilizes the data of every student to generate teams. There are two primary algorithms at work: a greedy algorithm which generates teams initially with possibly suboptimal sorting; and a GP algorithm which receives the result of the greedy algorithm to produce a more adequate result. This result is then stored in the Database.

3.1.4 Frontend Architecture

The front end of this web application was built using the Vue.js framework in conjunction with vuetify for aesthetic purposes. The front end consists of multiple different pages for each user, which differ based upon whether the user is a student or an administrator. Some mockup drawings of the application are in the **Appendix** below. The basic flow of user interaction with the application is in Figure 6.

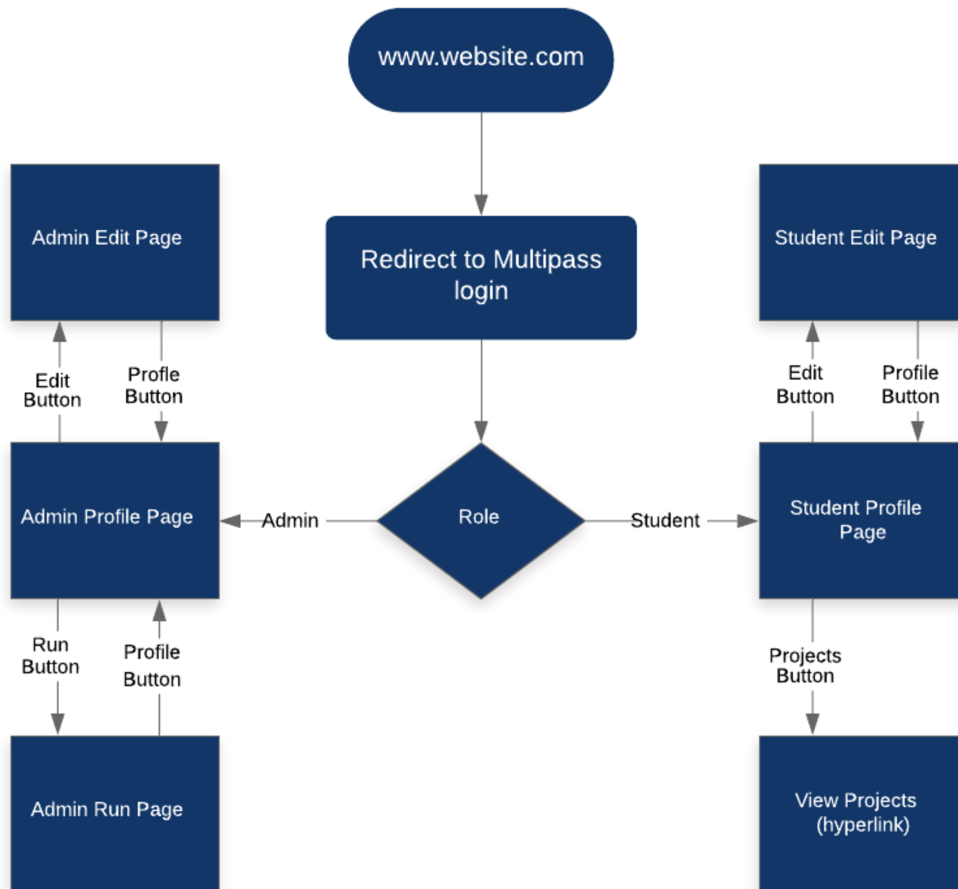

Figure 6: A flowchart demonstrating movement between web pages using the application's UI.

- Every User
  - Login Page which redirects user to multipass authentication.
  - Home Page which gives a description of the web application and informs users as to what will be used.
- Student
  - Profile Page to display users profile (Figure 11 in **Appendix**)
    * Displays status of team selection
    * Displays users preferences of project vs. team priority
    * Displays users project preferences

    * Displays users teammate preferences
    * Displays users teammate avoidances
  - Edit Page to edit users profile (Figure 12 in **Appendix**)
    * Edits any of the components displayed in the profile page
- Administrator
  - Profile Page to display student profiles (Figure 9 in **Appendix**)
    * Allows administrator to select a student and view their information including all information that a student can see in their profile tab.
  - Run Page to run optimization algorithm (Figure 10 in **Appendix**)
    * Allows Administrator to run optimization algorithm to group students to teams based upon their profiles.
    * Allows administrator to manually make changes to group students.
  - Edit page to allow administrator to add projects (Figure 13 in **Appendix**).
    * Allows administrator to input a .csv file with a project to be added.

3.1.5 API

To make the frontend and backend communicate, API endpoints used by the application are defined. The paths below are relative to the project root.

Method	Path	Description	Authorization
GET	/	Gets the main page for the application.	None
GET	/api/:profile	Gets the profile page for the requested user.	Student/Admin
PUT	/api/:profile	Updates the profile page for the requested user.	Student/Admin
GET	/api/login	Gets the login page for the application.	None
GET	/admin	Gets the admin page for the application.	Admin
GET	/api/:runner	Gets the running page for the algorithm.	Admin
POST	/api/:runner	Runs the algorithm.	Admin
GET	/api/:edit-projects	Gets the admin project editor page.	Admin
POST	/api/:edit-projects	Posts JSON of student profiles to load.	Admin
PUT	/api/:edit-projects	Allows the admin to add a project or user.	Admin
DELETE	/api/:edit-projects	Allows the admin to delete a project or user.	Admin

Table 1: API endpoints

It is important to adhere to the create, read, update, delete (CRUD) paradigm in defining these methods. By way of the internet today, everything will be done over HTTPS as well.

Since this is a single-page application (SPA), redirection of the user to multiple different pages should be avoided. Instead, everything is updated through API calls.

4 Technical Design

4.1 Algorithm Design and Approach

For this project, the team considered several different approaches for writing a team-sorting algorithm, which was one of the primary requirements for the project. Due to the presence of both "project" and "teammate" preferences/avoidances between students, this is fundamentally an operations research (OR) problem, as there is a large solution space with particular constraints to be met.

To elaborate, the client decided that sorted teams must obey the following rules:

- Students must not be on a team with someone whom they chose to avoid.
- Students must be on a project they prefer, or they must be with a teammate whom they prefer.
- Team sizes must be within the client's specified size.

Mathematically, these constraints may be itemized via some kind of "score," e.g. a numeric score calculated per how many constraints are met/not met after the algorithm is run. This will be discussed more in section **Genetic Programming** below.

On a small scale, one may be able to brute-force this task. However, towards the start of the project, the team quickly found that a brute-force solution would likely yield an algorithm that would run in $\mathcal{O}(n!)$ time, where n is the amount of students being assigned. Demonstrably, this is unfavorable. Since the CS@Mines Field Session class typically has over one hundred students (the Summer 2020 semester did, at least), any brute-force algorithm would be highly impractical.

On the other hand, the OR field has several approaches to this kind of problem. First, the team considered an integer programming (IP) algorithm for this task. With IP, one has several variables $\mathbf{x} = (x_0, x_1, ..., x_{n-1})$ and several weights $\mathbf{w} = (w_0, w_1, ..., w_{n-1})$, and a certain constraint C which must be met, given the following rules [1].

$$\text{maximize} \sum_{i=0}^{n-1} x_i \qquad \text{given} \sum_{i=0}^{n-1} w_i x_i <= C \tag{1}$$

Sadly, while straightforward, the team found this process too difficult to adapt to the notion of teams. If students were simply being put in *one* big project, and had to meet certain weights, this would be an easier problem. This was not the case, however.

Going back to the drawing board, the team then considered simulated annealing (SA) for the algorithm. SA was initially appealing due to its ability to find global maximums (of student satisfaction, contextually) instead of being stuck at local maximums [2]. However, SA suffered a similar issue to IP, where difficulty was found adapting the team-sorting problem to its approach.

To that end, the team considered one more option: the multiple knapsack problem. The knapsack problem involves trying to fit as many items into a knapsack at once, with weight being a constraint. Each item has a certain weight, and cost, and one wants to maximize cost while minimizing weight. Formally, this is similar to equation (1) above. The *multiple* knapsack problem is an extension of the knapsack problem, now allowing for multiple knapsacks with different constraints [3].

This would have been useful for team selection, as each team could be thought of as a "knapsack," while each student could be thought of as an "item" to place in the knapsack. The issue with this approach, however, is that the multiple knapsack problem *assumes* that the weight limit on each knapsack is known and is a hard upper bound; for the team's case, they had a hard lower bound which may be variable depending on the project's constraints.

Finally, the team looked at the nurses scheduling problem (NSP), which is a very similar problem in nature. Nurses request certain work schedules from the hospital, which requires there to be at least X staff at any given time. Nurse schedules are considered “soft” constraints, where the hospitals requirements come first (“hard” constraints) [4]. This is analogous to the team’s problem, where student preferences were soft constraints, and project sizes and student avoidances were hard constraints.

In a paper [4] by Lizzy Augustine et al., Augustine et al. provide an overview of the work that has been done on the NSP. This is when the team encountered a “genetic approach” to team sorting. The problem described by Augustine et al. was extremely similar to the team sorting problem, so the Reconnect team adapted it to their uses, utilizing other sources along the way (such as [5]).

4.2 Genetic Programming

At a high level, genetic programming (GP) involves codifying relationships between pieces of data as genetic traits in generations. More favorable relationships–such as more students who prefer a project being assigned to said project–manifest traits that are more likely to be passed onto later generations [4] [6]. This is similar to Darwin’s theory of evolution, where individuals who are deemed more “fit” will pass favorable traits onto later generations.

Before we proceed, it is important to clear some nomenclature regarding genetic programming, especially in context of the team’s project. GP, having “genetic” in the name, appropriates several terms from genetics. A *generation* is a collection of individuals, which is a collection of chromosomes, which is a collection of genes. Succinctly, this may be represented with set notation:

$$gene \subset chromosome \subset individual \subset generation$$

In genetic programming, each individual is scored on “fitness,” similar to real genetics, and the most “fit” individual is “selected” to populate further generations. This can be an expensive process, but it can be good for quickly finding high-fitness individuals.

The process of evolving generations, in this case, may be outlined in a few steps [7]:

1. Seed an initial generation, giving it individuals with at least some fitness.
2. Select the fittest individual from the generation.
3. Take that fittest individual and, from it, create a new generation.
4. In that new generation, mutate a few of the individuals.
5. Continue the previous four steps until little or no increase in fitness is demonstrated in the individual with the highest fitness.

This is an adapted version of John Koza’s outline for GP [7], which is more general than needed for this project. Koza mentions “architecture-altering operations” for GP, which the team deemed impractical for this problem. Moreover, individuals are not “bred” with each other as there is no clear distinction between individuals of high fitness, making sexual reproduction ineffective for the task at hand.

The implementation of this process for the algorithm can be seen in **Implementation** below.

4.3 Nomenclature

Some of the nomenclature for GP, as it applies to the algorithm created, may be confusing due to the conflation of "individual" and "student" (which is a gene in the algorithm), or "individual" and a list of projects. The diagram below aims to alleviate this confusion.

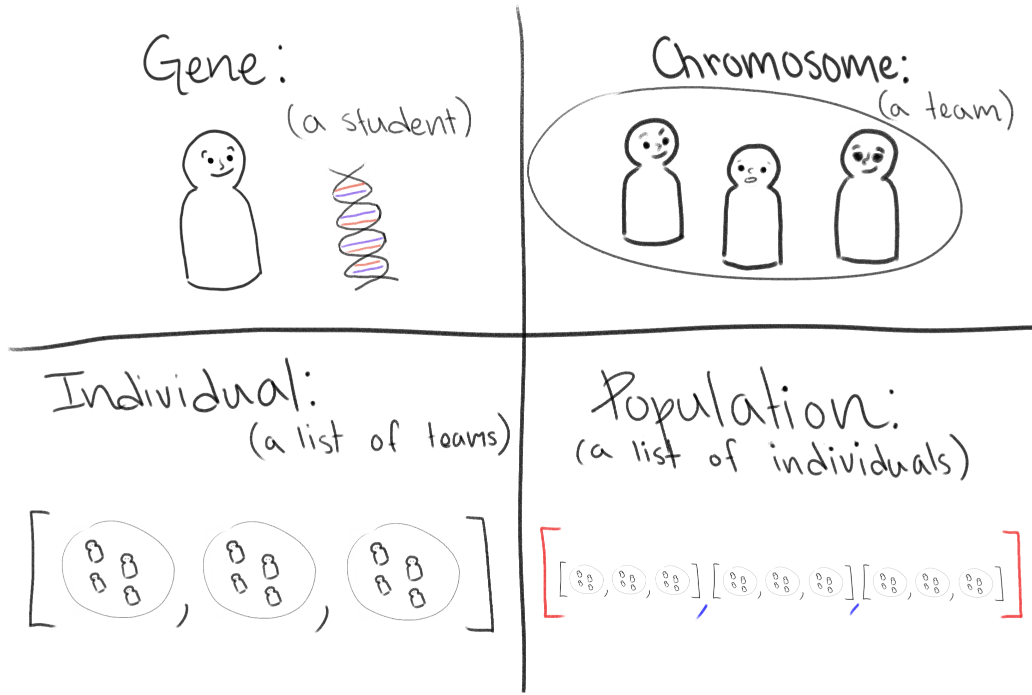

Figure 7: An image that maps GP terminology to the terminology of the algorithm used.

A gene is a student, and a team is composed of genes, or several students. The chromosome carries the information of every student as well as extra constraints, such as the amount of people that should be on said team.

The point of the algorithm is to find the most fit "individual," so in that case, each individual is a list of teams. Individuals may evolve by swapping students between teams (i.e. crossing chromosomes). A population (or generation) is a list of individuals. A population may be generated by swapping students between teams in a particularly fit individual, resulting in several mutated clones of that fit individual.

4.4 Implementation

The implementation of this algorithm relies on two primary components: A greedy seeding procedure and the GP implement.

4.4.1 Seeding

For seeding, the team chose to implement a greedy seeding algorithm for reasons outlined in **Genetic Programming**. If one simply starts with randomly assigned teams, it will be very difficult for the genetic component to actually evolve; at the very least, it will take a lot more time.

Thus, the seeding algorithm optimizes to put students in teams depending on whether they chose teammates over project preferences or not. If a student wants to be with teammates, the greedy component will prioritize putting students on teams together. If a student wants to be on a certain project, the component prioritizes putting said student on that team. Sadly, if a student has no preference, they will

likely be sorted last. Overall, the greedy component *primarily* prevents students being on teams with people whom they avoided.

To add variety between runs, the seeding algorithm also picks *random* students with each iteration. That way, ideally, no seed will end up the same. The Mulberry32 seeded PRNG procedure was used for randomization. Finally, the greedy part optimizes to make sure that no project has fewer students assigned than requested. It also prevents there from being more students assigned than required.

4.4.2 GP Implementation

The team's implementation of the GP algorithm selects for the fittest project assignments based on a scoring function. At a high level, this scoring function provides positive scores for every student that gets sorted with people whom they prefer or projects they prefer; it provides negative scores for team size violations, or teams with students whom avoided each other on assignments.

Once the fittest individual is selected from the generation, students are randomly swapped between projects to emulate the repopulation of a new generation. Finally, extra randomness is added in the form of "mutations," which exacerbate the randomness in only a few newly birthed individuals. Generations are repeated with 100 individuals each 100 times; at the 100th generation, repopulation ceases, and the fittest individual is served as the "solution." A diagram of this process may be seen below.

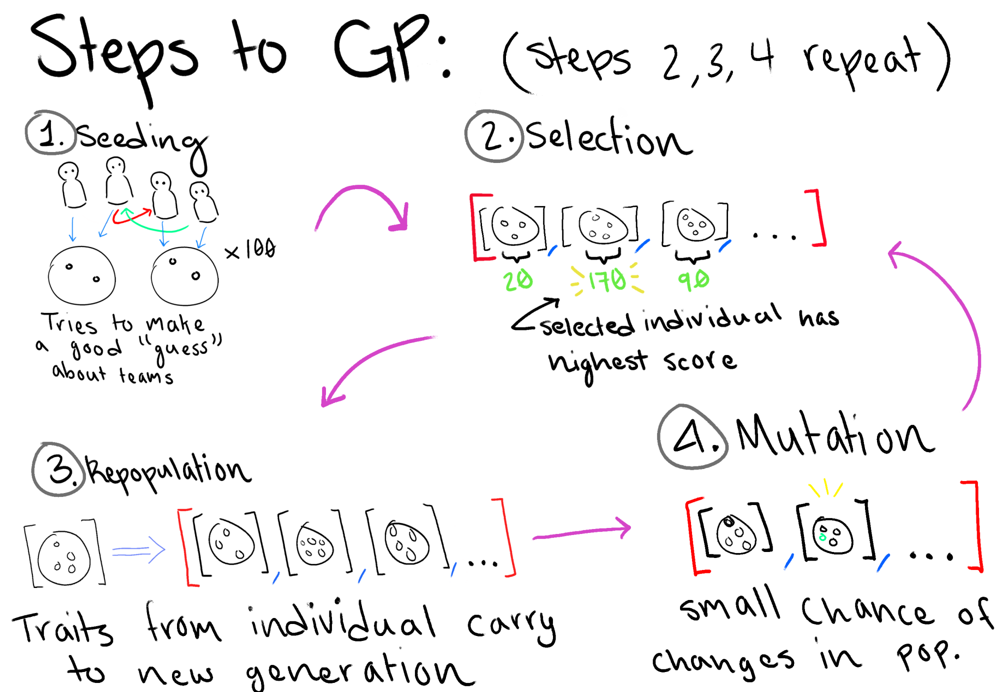

Figure 8: An image that demonstrates the GP process of the algorithm.

The team has tested this algorithm on anonymized test data from previous CS@Mines Field Session classes. Fortunately, for one test, the team has found that over 99% of students are on a team with a teammate they prefer, 92% of students are assigned a project that they wanted, and 89% of students are assigned to their first project preference. While this is for only one test set, it demonstrates that the algorithm adequately completes its required tasks. It does this within about 30 seconds, which satisfies the client's time constraints.

5 Quality Assurance

5.1 Frontend-Backend Integration

The frontend shall primarily communicate with the backend via API calls. These API calls interact with the backend via the typical create, read, update, delete (CRUD) functions. Most frequently, a user will be creating content, which will be posted to the backend using POST requests, which will have a JSON body.

That said, there exist some concerns with submitting raw JSON to the backend. While all user submissions will be transmitted over HTTPS, the backend must guarantee that the JSON is valid, and will not cause problems for the rest of the architecture. Similarly, since this project utilizes a database, SQL injection must be prevented.

5.2 Backend-Database Integration

As said in the previous paragraph, it is the backend's responsibility to make sure that any actions pertaining to reading from or writing to the database are valid. SQL injections is the primary concern for this project. Of similar concern is leaking sensitive information; users may specify other users to "avoid" in team selection, and it is imperative that other students do not find who a particular student may choose to avoid. Furthermore, GPA data, which is subject to FERPA, is requested from students and stored in the database.

On the backend, user input is sanitized to prevent SQL injection, and making sure that only users with valid authorization may access whom they avoid. (For this project, this means whoever created the "avoid" stipulation, and any administrator). Sanitizing inputs may result in a small usability tradeoff, but it is important for preventing data breaches.

Small unit tests may be used to test whether inputs were sanitized or not. Such tests send a SQL query from the frontend and see if it gets rejected on the backend.

5.3 Algorithm Considerations

Since team sorting involves sensitive user data, ethical considerations must be made when designing the algorithm.

5.3.1 Constraints

The project also necessitates other constraints to the algorithm to satisfy users. The team has discussed these constraints with the client, and are in agreement that these constraints are satisfactory for this project.

- If someone prefers teammates over project preferences, then they should have at least one teammate they prefer on their team.
- Similarly, if someone prefers projects over teammate preferences, then they should be on at least one project they prefer.
- Someone who is ambivalent should have at least one preference on their team.
- Each team must have either a certain amount of students (between a min and max), or no students at all (the project then goes ignored).

While these constraints do not cause much harm if they are not satisfied, via Michael Davis' "reversibility test," the team anticipates that very few people would be happy, from a user's standpoint, to be sorted into teams that they did not prefer, with people that they did not know.

5.3.2 Avoiding Other Teammates

In team selection, the website allows students to choose other students to "avoid" in team selection as well. The user does not need to justify why they want to avoid other students. With that in mind, the team selection algorithm must avoid placing students on teams with students whom they requested to avoid.

This complicates the algorithm, however via Michael Davis' "harm test," a computational trade-off is worth it when compared to students being forced to work with other students that they want to avoid.

For all of the algorithm's constraints, unit tests can be written to guarantee that the algorithm's results meet each constraint with a reasonable degree of accuracy.

Since the algorithm is stochastic, it would probably be best to run the algorithm multiple times on the same data set with different seeds to see how often it comes up with an answer that meets the constraints. Since an admin may alter teams after they have been generated, team generation does not have to be absolutely *perfect*, but preferably it would be *at least* 95% accurate.

5.3.3 Timing and Optimization

Team sorting and selection problems are $\mathcal{NP}$-Hard, meaning that they typically take a long time to run.

The team and the client agree that the algorithm shall take no more than 30 minutes to run, given 10 threads and 10 GB of memory on Mines' server Bartik.

If the algorithm takes over 30 minutes, it "bottoms out" and yields one of its generated solutions. (Since the algorithm is genetic, it continuously generates solutions in "generations," solutions ideally getting better each generation). Benchmarks shall be taken.

5.4 Code Quality

- The code shall have consistent formatting and descriptive variable, function, and file names.
- This design document, approved by the client, shall serve as documentation.

6 Results

6.1 Summary of Testing

6.1.1 Frontend

The website frontend is confirmed to display and behave as intended when used with the following browsers:

- Firefox 73 and 76 on Linux
- Chrome 83 on MacOS
- Safari on MacOS

Chrome and Firefox are the only browsers officially supported. However, since this website is being created for use by Mines students and staff, basic compatibility for Safari was worth checking for.

6.1.2 Backend

The backend handles authentication and routing. As such, most of its tests focus on those aspects. The team spent many hours ensuring that API calls were routed correctly (with the correct behavior). Similarly, the team worked to make sure that students could not subvert authentication, which would pose security/privacy concerns to the user.

6.1.3 Database

Most interactions with the database are made via API calls, so to make sure that the database is functioning properly, it was tested with said API calls.

For security, the database is password-protected password on the database, and POST or GET requests to the database cannot be made without proper credentials.

6.1.4 Algorithm

Most of the project's algorithm code was written from scratch, amounting to over 500 lines of code (documentation and comments included). The JestJS framework was used for unit tests for parts of the algorithm. JestJS is a testing framework for Javascript, similar to JUnit for Java. The team made sure to maximize coverage of various parts of the algorithm to make sure that it operates as expected.

6.2 Lessons Learned

- Node.js and its package manager npm made developing a functioning application a quicker process than writing one from scratch would be. However, Node.js has issues including misleading error messages and out-of-date dependency names. Each module also has an enormous number of dependencies. Despite having only a few pages, the frontend has approximately a thousand dependencies. This made Vue compilation take a noticeable amount of time.
- Due to the team's lack of experience with web development, it was sometimes difficult to understand what the various modules were doing "under the hood." Most online web development tutorials tend to be conducive to writing code that has a given desired behavior, but not explain the fundamental concepts behind the code.
- The most challenging part of the project was integrating each component (frontend, backend, authentication, database), making sure that they function properly when connected.
- It is easy to implement an application of this nature without encryption and authentication, but poor security practice. Securing the application makes it somewhat harder to test; to that end, it would be useful to establish authentication and tokens for each developer earlier in the development cycle than the team did with this project.

6.3 Future Work

Had there been more time, the team would have liked to implement the following features.

- For deployment, a line of code has to be manually changed to switch the authentication from Google OAuth to Shibboleth. It would be better programming practice to have this be part of a configuration file read when the server starts instead.
- Similarly, it would be beneficial to write an installation script in Makefile or bash to improve ease of deployment and testing. An init script was written for testing, but it was not professional-quality. A proper init script would also aid with deployment and testing.
- The website frontend has only been tested on the web browsers listed in Section 1. Performing more extensive testing on mobile devices and other browsers, and editing the code as needed for proper behavior, would improve the user experience.
- The algorithm is tackling a $\mathcal{NP}-$Hard problem, and so time and space complexity is a necessary tradeoff for a closer-to-ideal solution. However, it could still be a little more efficient in terms of time and space complexity. It is sufficient for scoring teams with admin intervention. However, for maximum efficiency, it may help to overhaul the current codebase and re-write it in C++, utilizing a foreign-function interface (FFI) to NodeJS.

6.4 Features Not Implemented

All features specified in the **Requirements** and **Design** sections of this document were implemented.

The client intends to clear the database every year, making data from previous years unavailable. This poses little concern to the functionality of the application, but is worth noting.

Compatibility was one of the goals for this project, and while there is strong coverage of the browser market share (Chrome, Firefox, Safari), only the most recent versions of those browsers are supported. Microsoft Edge and the mobile versions of those browsers are also common, but the site has not been tested on them.

7 References

[1] Web.mit.edu. 2020. Integer Programming. [online] Available at:

⟨ web.mit.edu/15.053/www/AMP-Chapter-09.pdf ⟩ [Accessed 10 June 2020].

[2] Jacobson, L., 2020. Simulated Annealing For Beginners. [online] Theprojectspot.com. Available at: ⟨ theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6 ⟩ [Accessed 10 June 2020].

[3] Google Developers. 2020. Multiple Knapsacks OR–Tools Google Developers. [online] Available at: ⟨ developers.google.com/optimization/bin/multiple_knapsack ⟩ [Accessed 10 June 2020].

[4] Augustine, L., Faer, M., Kavountzis, A. and Patel, R., 2020. [online] Math.cmu.edu. Available at:

⟨ math.cmu.edu/~af1p/Teaching/OR2/Projects/P23/ORProject_Final_Copy.pdf ⟩ [Accessed 10 June 2020].

[5] Jacobson, L., 2020. Applying A Genetic Algorithm To The Traveling Salesman Problem. [online] Theprojectspot.com. Available at:

⟨theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem⟩ [Accessed 10 June 2020].

[6] Koza, J., 2020. Tutorial. [online] Genetic Programming. Available at:

⟨ geneticprogramming.com/Tutorial/ ⟩ [Accessed 10 June 2020].

8 Appendix

8.1 Website Pages

Figure 9: Admin Students Page

Figure 10: Admin Teams Page

8.2 Website Pages, Mock-ups

PROFILE PAGE

Reconnect Profile Edit Projects

Name: John Doe Email: email@email.com Minor: data science GPA: 4.0

Preference: My Chosen Teammates

Project1: example project
Project2: example project
Project3: example project

Preferred Teammates: Jane Doe, ...
Aviod Teammates: John Smith, ...

Experience/Rationale: Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Colorado School of Mines

Figure 11: Student Profile Page

EDIT PAGE

Reconnect Profile Edit Projects

Minor: GPA:

Preference:

Project 1
Project 2
Project 3

Prefer Teammates
Avoid Teammates

Experience/Rationale:

Colorado School of Mines

Figure 12: Student Edit Page

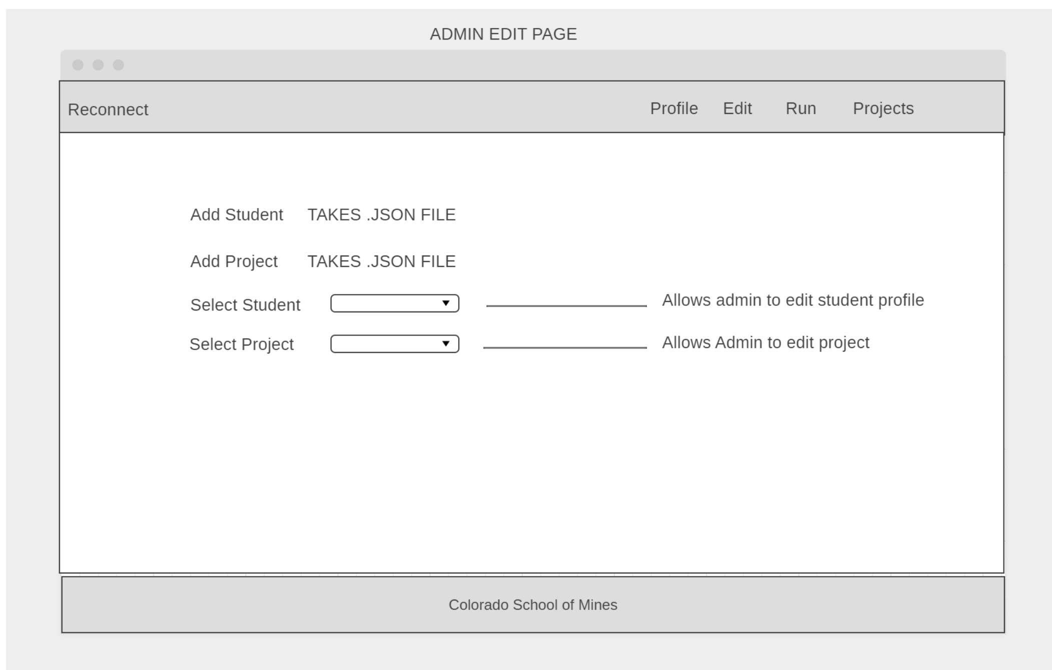

Figure 13: Admin Edit Page

9 License

The Reconnect source code is released under the Apache 2.0 License at `https://github.com/cpainterwakefield/fs-team-assembler`.