COLORADOSCHOOLOFMINES
EARTH • ENERGY • ENVIRONMENT

ADVANCED SOFTWARE ENGINEERING

CSCI370

CSM EDNS 2

Client: Dr. Chelsea Salinas, Mines EDNS

Authors
Tamara COUSINEAU
Paris FLOYD
John HENKE
Anna OTTERSTETTER

June 16, 2021

CS FIELD SESSION: THE STABLE MARRIAGE

Contents

1 Introduction

Each semester, the Colorado School of Mines EDNS Department has the task of matching senior design students of varying majors with projects. Each project requests a certain number of students from each major it desires. Furthermore, each student submits a list of n ranked projects, from most to least preferred. The department ideally assigns students with a highly ranked project on their preference list. Currently, the group process is performed manually. While it produces effective results, it is an extremely arduous and time-consuming process.

This project automates the capstone teaming task, as described above, and is geared toward maximizing the happiness of students and fullness of projects. However, due to the fact that student happiness is a priority, the requested number of students in a project is a slightly flexible requirement to ensure that students get placed in a project on their list. This program, therefore, aims to match all students fairly and maintain project major preferences where possible. The program also flags unpopular projects in order that these projects can be dropped for the current semester and potentially reevaluated for assignment in future semesters. Finally, it creates output files, including multiple CSVs and Excel files listing the students, the project they were matched with, and the student's ranking of their assigned project. Successful completion of this program will save precious time for the client that will be better spent on more academic and human-oriented tasks.

2 Requirements

2.1 Functional Requirements

The overall functional requirements for the capstone teaming project include the following:

- An automated software product for capstone teaming that efficiently provides an optimal solution (of which there are many).
- Compatible with different input file formats. This is solved with YAML files.
- A "cost" function for a given "state."
  - A state is an assignment of students to teams.
  - The cost function will calculate the overall "cost" of the given state. The cost is what we are trying to minimize.
- Fairness - want to make sure that students are not given special treatment by the algorithm based on characteristics like their major or the date that they answered the survey.
- Output a formatted CSV/excel file of the client's specifications.
- Handles special cases where students must be on a project.
- Identifies unwanted projects

2.2 Non-Functional Requirements

- Program must take in a CSV file created from an online survey.
- Program's output file must be readable and contain what choice number the students' assigned project was for all of them.
- Similarly, the input file must ask for reasonable information that both students and project heads can access easily.
- Program must be adaptable to changing surveys or potentially inconsistent input.
- Program must be clear and easy to understand for later improvements and/or changes.
- Program must be easy to run and install, and will be made using Python.
- Program must aim to optimize happiness of both students and clients.
- The grouping process must be done fairly.
- Documentation must be sufficient such that low-code literacy clients can run the code and not have issues.

3 System Architecture

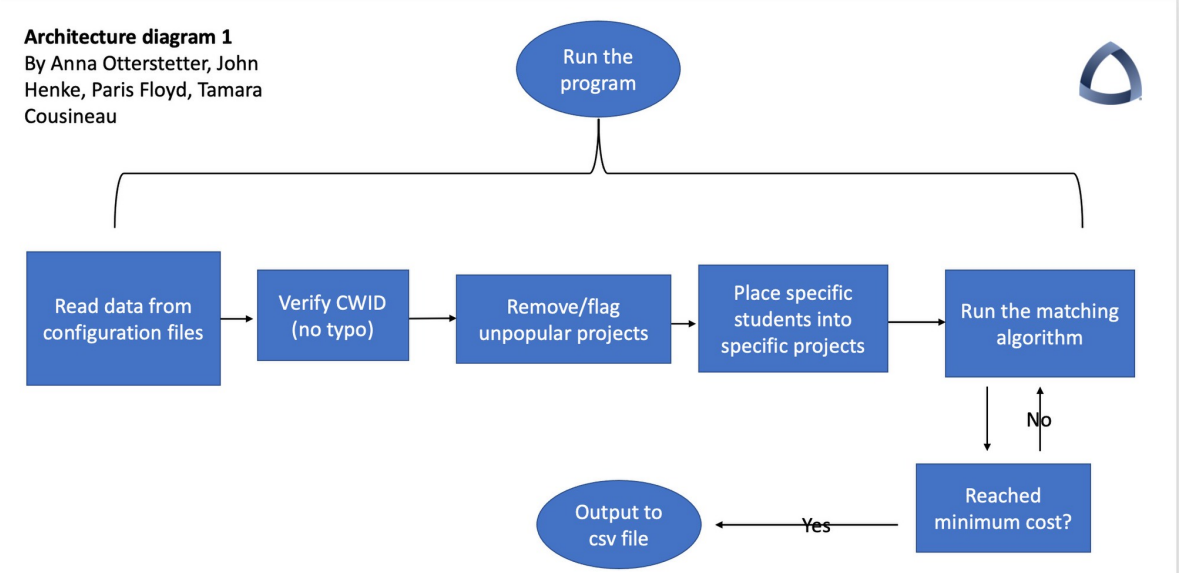

Figure 1: Architecture Diagram I

Displayed in Figure 1 is a visual overview of how the program executes from start to finish. First, we read in the student and project data from given .csv files. It utilizes configuration files to find the location of specific information within the data files. Next, we verify that each student has entered their campus-wide ID properly. This is important in ensuring there is no confusion between students. Following this, we use the ranking algorithm to flag the unwanted/unpopular projects, and then we allocate the students that are required to be placed onto a certain project. The matching algorithm stage, and the most time-consuming part of the program, consists of assigning students to projects. The algorithm runs repeatedly, and uses the cost function to determine which of the many possible assignments has the lowest cost. This final, most ideal assignment, is then output to an excel file.

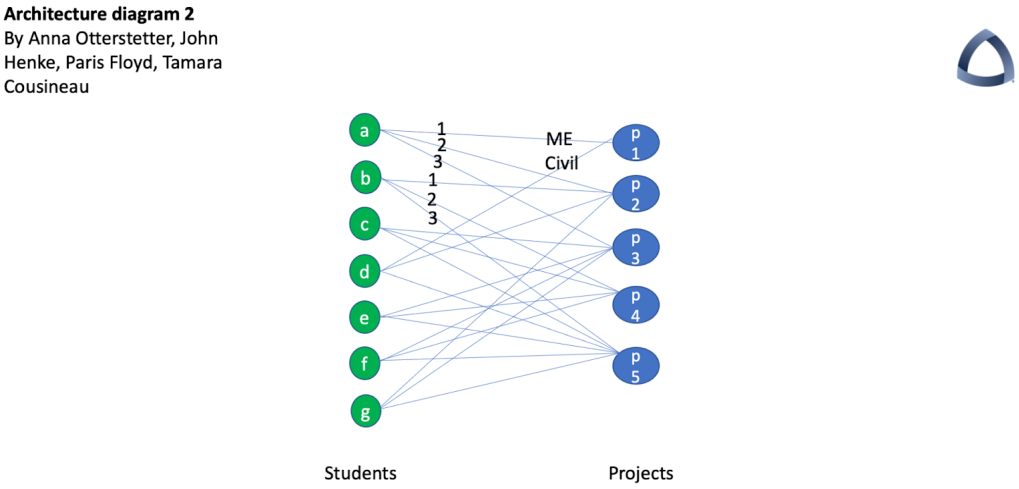

Fig. 2: Stable marriage problem. The number on each line representing the ranking of each student. On the right hand side, the labels on each line represent the majors requested for each project. Not every number are shown on the figure, for simplicity purposes.

Figure 2: Architecture Diagram II

Figure 2 is a bipartite graph illustrating the stable marriage algorithm. The graph consists of two columns: one representing the students, and the other one representing all the projects. The lines originating from the students and ending at projects represent the projects that the students wish to be assigned to. The number on the line is the ranking entered by each student. For example, student 'a' would like to be part of 'p1' as its number one choice, 'p2' is the second choice and 'p3' the third choice. From the view of the projects, each project has a certain number of students it can accept and also requests that these students be of certain majors. For example, Figure 2 shows that 'p1' needs one student in mechanical engineering (ME) and one student in civil engineering (Civil). Since Figure 2 is a simplified version of the problem, it does not show all the lines going from 'p1' to all the students with 'ME' as a major. In reality, any mechanical engineering student would be a candidate for the project (from the project's perspective).

4 Technical Design

4.1 Ranking Algorithm

In order to determine which projects should be dropped due to their unpopularity, a ranking algorithm was designed to determine a project's popularity.

$$\text{score} = \frac{1}{(\text{num diff majors needed})} \left(\sum_{i=0}^{n} (n-i) \left(\sum_{j=1}^{\text{num majors}} \frac{(\text{num people favorited})_j}{(n \times (\text{num of major}_j \text{ needed})_i)} \right) \right)$$

Let $n =$ the number of top choices received by the students. In order to ensure fairness across all projects, several different weights were added to this formula. First, it was necessary to ensure that projects requiring more people would not have an unfair advantage over projects requiring fewer. For example if a project requested five mechanical engineers and exactly five listed it as their top choice, that project should definitely not be dropped. However, if each vote for a top choice for a project was simply one point, then a project requiring 50 mechanical engineers and getting six top choice listings would score higher than the perfect scenario described earlier. To avoid this project scoring higher, the project points were divided by the number of people required. This ensures no unfair advantage to the projects requiring more points.

Next, consider the situation where a project requires a handful of students from two different majors but only one type of major heavily likes it. The ranking algorithm needs to ensure that all majors are equally weighted. This is taken care of through the second summation, where each each of the m many different majors required holds $(1/m)$ much of the total score weight.

Finally, the projects must be weighted differently according to how the students rated them. For example a students top project should receive more points than their 8th project. This was accounted for by multiplying the denominator by the variable n and multiplying the numerator by a decreasing metric. For example, if a project requires five mechanical engineers and a student lists their top eight projects, then if it is listed as the students top choice, it will receive $8/40$ more points to their score. Similarly, if it is second, then it will receive $7/40$ and so on until listed as eighth at $1/40$.

After this scoring has taken place, the projects are then sorted in score order. Next, the number of students the projects can hold are counted until a project is reached that exceeds this limit. Every unpopular project following this last one is then flagged as a potential dropped project.

4.2 Matching Algorithm

As stated previously, this problem is fundamentally a variant of the stable marriage problem. The stable marriage problem is a general problem in which there are an equal number of men and women that provide a ranked list of the partners they prefer. The algorithm then creates an optimal matching between the men and women (i.e. creates couples) [1].

An optimal solution within this problem is named "stable". A matching of men and women is considered stable if there is no couple that is not currently matched (i.e. they are

currently with other partners) that would leave their current partners to be with each other. In other words, there is no rogue couple (y, x) such that y prefers x to his current partner and x prefers y to her current partner. These two would abandon their current partners to be with each other (make a side deal), which results in instability and undermines the integrity of the algorithm.

Within the student-project assignment problem, the students are the "men" proposing (i.e. applying) to the "women" (which are the projects). There are two main variations from the base stable marriage problem with this context:

1. The women (projects) can be married to multiple men (students)
2. The projects are indifferent between students of the same major

The first variation is handled within the Hospitals-Residents problem (HRI), in which medical students are assigned to hospitals for their residency. The graduating medical students provide a ranked list of hospitals, and the hospitals provide a ranked list of students. The matching algorithm uses these rankings to produce a student-optimal stable matching of students to hospitals, where student-optimal implies that students receive the best matching they could receive in any stable matching [1].

Abraham, Irving, and Manlove provide two algorithms for the more general student-project allocation problem which can be directly applied to HRI and the student-project allocation problem. The first algorithm is student-optimal, which implies that a student is assigned to the best possible project that s/he could obtain in any stable matching. The pseudocode that Abrahram, Irving, and Manlove provide was used as the skeleton of the pseudocode written to solve the capstone teaming problem.

The main difference between HRI, stable marriage, and the capstone teaming problem is indifference. Within capstone teaming, projects ask for a certain quantity of each desired major; however, the projects are "indifferent" to students of the requested major. In other words, all mechanical engineering students are equally desired for a project that requests mechanical engineers. Indifference introduces three notions of stability, which Irving introduced [2]. See Irving's paper (cited) for an explanation of the three notions of stability. Any solution to the capstone teaming problem can only be weakly stable because there will always be indifference on the side of the project; there is no matching that does not have indifference.

The pseudocode is very similar to the base matching algorithm. Disassociate all students and projects. While there is an unassigned student with preferences it has not yet applied to, that students applies and becomes associated with their highest currently rated project. Two cases then arise: the project can be oversubscribed, full, both or neither. If the project is oversubscribed, then the project determines its least determined student and disassociates that student. Ties are broken arbitrarily.

```
assign_students(I) {
    assign each student to be free
    assign each project to be completely unsubscribed
    while (some student s_i is free) and (s_i  has a non-empty list of preferences) {
        p_j = first project on s_i preference list
        /* s_i applies to p_j */
        provisionally assign s_i to p_j
        if (p_j is oversubscribed) {
            s_r = least preferred student assigned to p_j
            break provisional assignment between s_r and p_j
        }
        if (p_j is full) {
            s_r = worst student assigned to p_j
            for (each student s_j that contains p_j in their preferences) {
                if (p_j prefers s_r to s_j) {
                    delete p_j from preferences of s_j
                }
            }
        }
    }
}
```

Figure 3: Stable Matching Pseudocode

This algorithm results in a student-optimal matching; however, there is some variability due to the indifference, which is where the cost function comes into play.

4.3 Cost Function

When the stable marriage algorithm assigns students to projects, it attempts to do so in the best way possible; that is, we try to minimize the cost of the assignment. We find that a certain assignment has a cost associated with it whenever it is imperfect, which is to say, just about every time the algorithm is run. Imperfections arise when:

1. Students do not get on their top ranked project
2. Projects have more or fewer students than they requested of each major
3. Students are left unassigned after the assignment algorithm has completed

The program attempts to approach perfection, while accepting that imperfections will always exist. For example, not everybody can get their top ranked project. Therefore, it needs a way to programmatically find the assignment which has the least cost. To accomplish this task, a cost function was designed to determine the cost of an assignment.

Because an assignment is represented by a set of projects which contain unique students assigned to it, we can determine the maximum cost of an assignment by summing the costs associated with each project. This is because each project might have more or fewer students of each major requested than was hoped for or might contain students

which did not rank it highly. To calculate a project's cost, a member function of the project class is created.

In order to make the project's cost function, all the factors of the cost must be accounted for. With some attention to detail given, the considerations are as follows:

1. Students which ended up on a project that they ranked relatively highly should be given a scaling cost based on the rank they gave. In the case that a student ended up on a project that they ranked very low, the scaling factor should be increased.

2. If the project is over or under capacity by +/- one student, it should have a very low cost associated with it. Any difference above/below that one student should be weighted much more heavily and should scale based on the difference.

3. The difference between the number of students representing each of the requested majors assigned to the project and the number requested by the project should also represent a cost. Of course, if a student of a major that was not requested ends up on the project, there should also be a cost.

These considerations are integrated into a function programmatically by implementing the following pseudocode:

```
initialize total cost of assignment to 0;
for (each project in the assignment) {
        initialize the cost of this project to 0;
        if (nobody is on the project) {
                set this project cost to 0, continue;
        }
        if (project is over/undersubscribed by more than 1 student) {
                project cost += difference in subscription;
        }
        for (each major in the project) {
                project cost += difference between number of this major
                requested and number enrolled;
        }
        for (each student assigned to the project) {
                project cost += rank the student gave this project - 1;
                // Note that a rank of 1 has cost 0 and scales from there
        }
        add cost of the project to total cost of assignment
        add cost associated with the number of unassigned students to the
        total cost of the assignment;
}
```

Figure 4: Cost Function Pseudocode

Once each project in the assignment can return a cost, the cost function for the entire assignment simply sums the costs of the projects and the unassigned students.

When the cost function is constructed, it allows the program to create a first assignment (of students onto projects), calculate the cost, do another assignment, compare the

cost to the previous one and save the assignment with the lowest cost, and repeat until the assignment with the minimum cost is found. Given the intended randomness of the algorithm, finding this minimum cost takes a very long time with our hardware. Therefore, assignments are made within a certain tolerance of time and the lowest cost is saved to be output to data files.

5 Quality Assurance

In order to guarantee the product is of high quality throughout the production process, the following must be true:

1. Every person on the team in creating the product must be working towards maintaining its quality with every addition to the code.
2. The dedication to high quality software should be reaffirmed often, not just at the very end of the production process
3. Quality is largely defined by the client; as such, the client must see the product as high quality when it is showcased to them.

In order to guarantee the quality of the product to the client, it is best to be continuously involved in the process of quality assurance. This way, it is guaranteed that both in shorter intervals (such as at the end of a sprint) and in the end (when the product is finalized), we can always have a high-quality piece of software ready to showcase to our client.

The practices and standards that have been maintained while designing and implementing the software are listed here:

1. Pseudocode
    - Before a solution to a problem is implemented or tests for that solution are designed, we must understand what is going to be involved in the solution - the required inputs, expected outputs, and general flow of a solution. In order to design the solution according to these needs, pseudocode is first written. The pseudocode is of sufficient detail, but still abstracts some of the more intricate and yet undeveloped functionality of the solution. This contributes to quality by enforcing the proper organization of the implementation before it is actually implemented, as well as simplifying the communication of ideas and concepts between the writer of the code and anyone looking to understand it after it's written.
2. Unit testing
    - Unit tests will be developed before a solution is implemented in order to explicitly lay out the expected output and use cases of the solution. These tests should be designed for every important function and should ideally examine the expected/average use cases as well as the edge cases that might be presented. This contributes to quality by checking for errors and failures of implementation before the product is given to the client at any stage in development as well as forcing the software that we deliver to be complete in its implementation at any stage.
3. User acceptance testing

- As mentioned previously, the fact that the client largely defines 'quality' for the software that we develop for them means that we must continuously check in with the client throughout the development process to make sure that the software is meeting their expectations of quality and usability. While it is not expected that the client should be able to comprehend the code that we have written, we should be able to explain what we have done to solve their problems and they should give us feedback on whether or not it matches what they had in mind for the solution.

4. User interface testing (automated or otherwise)

    - The client must be able to run the software conveniently. In order to check this, we must make sure at some point before the software is delivered that the client has had practice with and can successfully run our program on their devices. This is a big part of the client's definition of quality, and as such we must pay careful attention to it.

5. Documentation for the user

    - Simple yet thorough documentation will be provided to the user to help with configuration and use of the program. This will include instructions on how to set up and configure the environment necessary to run the program as well as install the required libraries and software, such as the yaml library and the Python 3 language.

6. Integration testing

    - Besides just ensuring the usability of individual components of the product, we must also ensure that the interactions between these components are working as intended. Integration testing can help to guarantee that the entire program works when all the moving pieces are put together. We plan to do this type of testing once most of the components have themselves been fully tested and implemented.

7. Code reviews

    - Code should be written according to a high standard of organization, commenting, and readability. This is ensured through the continuous practice of pair-programming, which allows one programmer to actively examine the code of another programmer as it's being developed. This improves both the efficiency of the coding process and the quality of the software, as two sets of eyes are always better than one for catching errors in both functionality and readability.

6 Results

6.1 Overview

This project came with a large learning curve. The most learning occurred with regards to teamwork, and specifically how to delegate the team's resources effectively and cooperate with busy and mostly inflexible schedules. In this regard, we have succeeded. Despite the difficulties in scheduling, we are on the precipice of delivering a high-quality product to our client that will meet almost all of her needs. Our final product is a well-tested, user-friendly, highly-configurable piece of software that we will be proud to show to our client. There is some room for improvement and polish, especially on the user interface side of things, but we are overall happy with our work.

6.2 Software testing summary, including accuracy and performance

Execution of our project has been tested utilizing the PyCharm IDE. In order to ensure the accuracy of our code, unit tests were written prior to writing the algorithms. The unit tests included edge cases and expected outcomes and were debugged until they ran without error. After this, the algorithms were written up and debugged until all unit tests passed. Once the unit tests passed, we could be certain that our algorithm achieved the goal it was intended, as unit tests were thorough in their search for edge cases. Furthermore, for testing, we created and used several different roster and project files. This allowed us to make sure that our project was not too specified on the edge cases present in our original sample file. Due to all unit tests passing on all given files, we considered the project well tested.

The accuracy of our program is satisfying, but we could work on improving the matching algorithm so as to put more students on projects that they ranked highly. For example, in the document provided to us to test our code, we have that 45% of the students get their first choice, 15% their second choice, 11% their third choice, 5% their 4th choice and 23% get their 5th to 8th choice. Ideally, we would want to reduce the last number of 23% to something closer to 10-15%. Another feature that we are currently working on improving is the number of unassigned students to each project. Reducing the number of unassigned students is helpful for the client as less students will have to be placed manually into specific projects, and the process being then closer to automated.

In regards to the performance of our main matching algorithm and its time complexity, we find that the general time complexity is O(n2). However, given that we execute our algorithm multiple times for each run of the program in order to optimize our accuracy, the actual time that the program takes to run is $O(cn^2)$, where c is some constant integer representing the number of times that the algorithm is run. In our integration tests, where we run the entire program and check the amount of time it takes as well as the results of the output, we find that the program completes within about two seconds when we choose not to repeat the matching algorithm. However, we find that when we begin repeating the matching algorithm, the execution time increases approximately linearly - if we decide to repeat the algorithm 100 times, we find that the execution time is typically more like fourteen seconds. These times were all based on tests that utilized data with more elements

than we expect the client to practically use. We will work on finding the balance between time complexity and the improvements in accuracy that increased iteration yields.

6.3 Plans for usability tests

For usability tests, we plan to have our client try out our product on a sample file she has. Due to sensitive student information, the data we were given had to be generated. Therefore, in order to ensure completeness, the client will test the product on actual data. In doing so, we will be able to help show her how to install all the required libraries and how to execute the program. In this, we shall be able to locate any possible confusing points in our documentation on how to utilize this product.

6.4 Missing features and possible future extensions

While our software meets almost all of the requirements and expectations set forth by our client, there are a few features that we intended to implement but which ended up being too difficult/time consuming to solve within the time allocated for the field session. Similarly, this is certainly room for further functionality and polish that future field session teams could take up.

In terms of the features that our client asked for but we could not quite deliver, we only have a few. The first one worth mentioning is organization/readability of the final output file - the client wanted the final output file to be formatted with borders, good headings, and the project rankings color-coded, but we could not quite deliver this. While outputting to CSV is easy with the libraries that we were using, outputting to a well-formatted excel file involves a much more detailed understanding of the libraries required to do so. As such, we decided to forgo this formatting in favor of improving the effectiveness of our matching algorithm. The second worth mentioning is that our program is not very resilient to input files which differ in format from those that we were given as sample data. For example, our program demands that single columns represent the last name, first name, campus-wide ID, project id, etc. and that each of the input files have exactly one row of headers at the very top of the document. While the flexibility is greatly improved by our configuration files, and we expect our client will have no problems providing input files for the program that match the expected formatting, we are still falling short of our goal to make our program work with many file formats.

One of the aspects of our software that could certainly be improved by future developers is the user's experience of running the program. In our current iteration, our client must install an IDE in order to run the program, and then further install the libraries required to run our program using the IDE's console. While we are making sure that this process is well-documented for our client who has a lower code literacy, it would certainly be nice if the process was more simple and intuitive (and, to that effect, might have a user interface), such that next to no documentation would be required. We imagine that a future group could work on packaging our program into an installation wizard, such as is commonplace with more sophisticated programs. This would make the process of running the programming a much more convenient one for any user.

6.5 Lessons learned

1. Python is an effective language for running more script-like programs and can support more organization than you would expect (e.g. classes, unit tests). Additionally, it is very flexible with different operating systems since it is an interpreted language.

2. It is alright if some members of a group don't fully understand the minute details of a certain piece of the project's functionality. When this is the case, however, it is nice for the author to provide a black-box explanation of how the function can be used for those who lack full understanding of it.

3. Forcing everybody to wait until the next scrum meeting to merge good code into master can be inefficient. Therefore, efficiency can be increased by forcing those who would merge with master off-schedule to check that everything they have developed has been a new addition rather than a modification to existing code. This helps to prevent merge conflicts.

4. It's hard to keep scrum meetings short. We frequently found that we would get done with the 'scrum organization' portion of the meeting within two or three minutes, and it was only natural to move on to group problem-solving and pair programming. It's best not to stifle this productivity, even if it forces the scrum meeting to be long. However, it is helpful to let everyone know that if they have somewhere to be, they can feel free to leave once the more productive section of the meeting begins.

5. We generated a lot of questions to ask our client, and despite this we quickly found ourselves with even more after each client meeting! After some reflection, it makes sense that we should have sent more emails containing our more time-sensitive questions. The lesson learned is: don't be afraid to be in even more frequent communication with your client than you expected. It can only help.

7 Future Work

If given more time, there are several ways this project could be further developed and improved upon to match both the client's stretch goals for the project and other imagined features that might improve the quality and functionality of the program.

First, the client requested that the output document be color coded for ease of navigation. However, color coding with the output method used proved to be more of a challenge than expected. After a sufficient time trying, it was eventually concluded that this addition was not as important as the optimization of the algorithm, and so the development team's attention was instead focused elsewhere. However, with more time, this is a stretch goal that would have been nice to implement.

Second, the client requested that some projects be given priority in the filling process. However, this request came to us a bit late, and therefore did not allow us the time to implement this into our design. The matching algorithm we used ensures fairness for all projects and students, and therefore implementing such an addition would change our program significantly. Therefore, it was deemed that there was not enough time left to implement this and so it is left as a future goal.

The final future goal for this project is to implement an easier, non-computer science friendly way to interact with and run our program. If we had more time, a GUI would have been a nice, user-friendly way to interact with the program. Similarly, and installer program would have been a much more efficient and smooth way to provide the software files to the client.

Bibliography

[1] D. J. Abraham, D. F. Manlove, and R. W. Irving, "Two algorithms for the student-project allocation problem," *Carnegie Mellon University: School of Computer Science*. https://www.cs.cmu.edu/ dabraham/papers/aim04.pdf.

[2] R. W. Irving, "Stable marriage and indifference," *Discrete Applied Mathematics*, May 2002. https://www.sciencedirect.com/science/article/pii/0166218X9200179P.