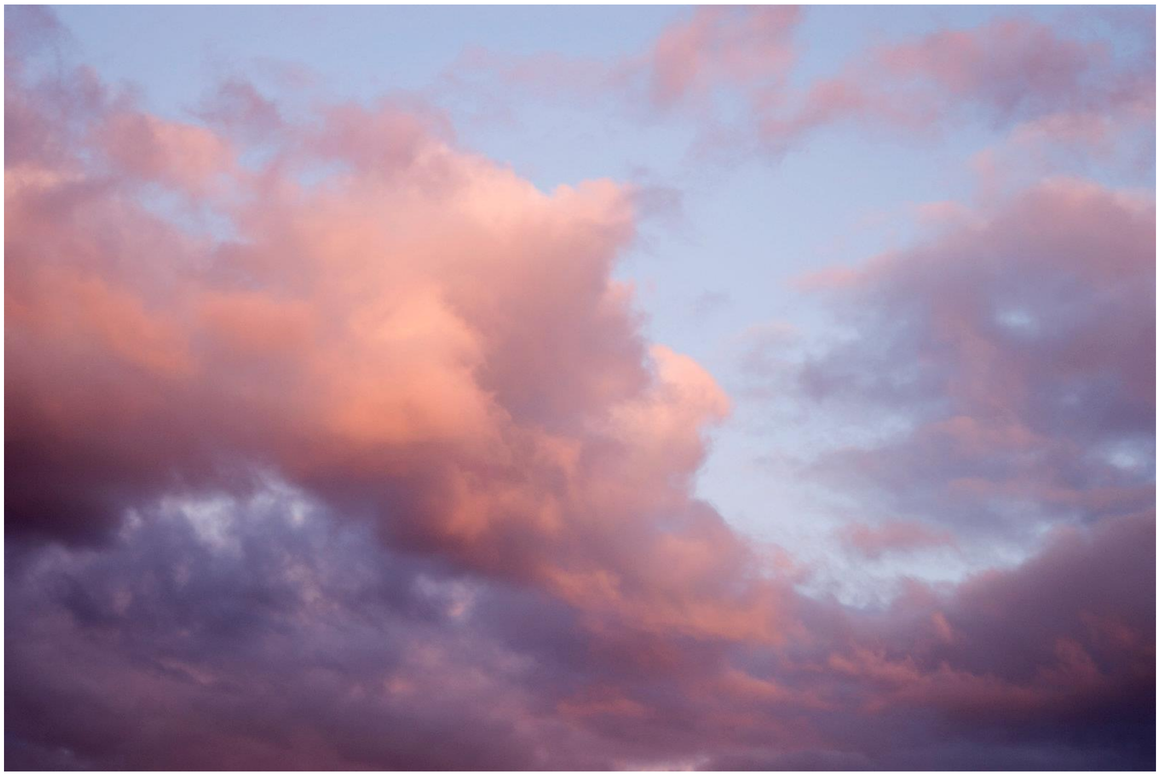


Final Report

Software Engineering Project 3

The Shady Bunch

Caroline Hyland - 13409662
Karl O'Brien - 13529873
Mark McKenna - 13504023
Niamh Mc Eniff - 13343846

Table of Contents

Specification of Project

The overall goal of this assignment was to design and implement Java classes to represent a table that maps students to preferences and that performs a search for the best mapping from one to the other.

Every year, fourth year students are competing for a finite number of final year projects. A list of proposed projects is produced by the School of Computer Science and students must decide which projects they would like to do. As it is not a perfect world we have many students wanting to do the same projects. As only one students can do a certain project the students must fill out a list for the top ten projects they would like to do. The projects are then allocated, trying to give as many people their most preferred options. Some students have arranged in advance to work with a project coordinator and draw up a specification for their own project. These projects are called preassigned projects. If a student has a preassigned project they only have to list that project and are guaranteed to get it. It is our job in this assignment to implement Java classes to represent the students and their preferences and search for a mapping from one to the other that keeps the most amount of people happy.

For this assignment we were given a .tsv file called "Project Allocation Data", this file contained a list of student names and their project preferences.

Before we started the assignment we did a Work Breakdown Schedule (see below) where we split the project up into work packages. Work packages are kind of like mini projects, it is the effort required to produce a deliverable in a project. We split the project up into three main work packages. We determined that each package should require about the same amount of effort.

The first package was storage. We needed a way of storing the data we were given in the file. For this we created the *PreferenceTable* class. This class reads in a tsv file and stores the contents in a two dimensional vector. This was a good way to store the information about all the students together but we felt that this was not enough and to make the program more efficient we should store information about each individual students separately, and so the *StudentEntry* class was created.

The *StudentEntry* class takes a student name as a parameter, stores it, and then searches the two dimensional vector from *PreferenceTable* for that given name and then stores the preferences that student has listed in a vector. As not all students have listed ten preferences we have a variable that stores the number of preferences that the student originally listed. The StudentEntry class also contains a Boolean variable indicating whether or not that student has a preassigned project. *PreferenceTable* creates a *StudentEntry* object for each student in the file and stores these objects in a Hashtable for easy access.

We had now stored all the data given to us and, so, it was time to start thinking of how to allocate projects. When the allocation of projects was finished each student would have only one project associated with them. As *StudentEntry* stored the student with all of its preferences we decided we needed a new class to store the student with the one project they had been allocated and thus we made *CandidateAssignment*. The *CandidateAssignment* class stores the *StudentEntry* object and the project that student has been assigned in a string variable called *currentAssignment*. The *currentAssignment* is a project chosen at random from the student's preference list.

Now we had made our first step towards allocating projects by assigning a project to each student but as the task was to find a way to allocate projects to all students, we needed a way to look at the entire mapping as well as individual mappings. From this requirement we created the *CandidateSolution* class. *CandidateSolution* creates a *CandidateAssignment* object for each student in *PreferenceTable's* Hashtable. We now had a way to look at the information given for each individual student, the project assigned to each student and the entire mapping and so completed the task of storing the data given.

The next work package to tackle was finding the best mapping of students to projects, a mapping that would give students one of their more preferred projects while not assigning the same project to more than one student. Before we could do this we needed a way to quantify our allocations and overall mapping. First we looked at *CandidateAssignment,* we decided to give a number called an energy to each assignment. The energy of an assignment was determined by where the project assignment was located in the student's preference list. The more preferred the project the lower the energy e.g. the energy of the first preference is 1 and the energy of the tenth preference is 100.

Then we turned our attention to *CandidateSolution.* Again, we wanted to attach an energy to the mapping. As each *CandidateAssignment* already had an energy we thought that a good way to quantify a *CandidateSolution* would be to get the sum of the energies of all the assignments. This would give us an accumulative number that represented how happy students were with the project assigned to them, but this did not take into account that some projects could be assigned to more than one student.

To avoid this we decided to add a penalty to our energy every time a student was assigned a project that was already assigned to another student. In order to make a noticeable impact on the energy this number needed to be large. As the energy of a student assigned their tenth preference was 100 we knew the penalty would have to be much greater than this, and so we settled on a penalty value of 1000.

Now that we had a way of quantifying the quality of a mapping we could start working on finding the best mapping. In our quest for the best mapping we were required to use two different algorithms. The first algorithm was Simulated Annealing. It appeared that this algorithm would require less effort than the next. Simulated Annealing is a technique whereby a single mapping is made and slowly changed until the solution is optimal. This was done by first making a new *CandidateSolution*, then choosing at random a *CandidateAssignment* then making a change to the assignment. For this we needed to add functionality to *CandidateAssignment* to allow us make these changes. We added a method called *randomiseAssignment* that allowed us to choose a new random project for that student from their list of preferences. We needed an undo method to allow us to undo the change we made if we did not wish to keep it since Simulated Annealing is a greedy algorithm. Once we made a change to the solution we checked if the change we made was good or bad, i.e. if the energy of the solution has fallen or risen. If the energy has fallen we kept the change and if the energy has risen we must decide whether to keep the change or not. Sometimes we must keep a change that initially makes the solution worse so that further down the line the solution will actually improve. The Boltzmann function tells us whether to keep the change or not. We make many of these changes, over 100 thousand, in the hope that the final solution would have the lowest energy possible

The second algorithm that we had to implement was the Genetic Algorithm. Genetic Algorithms work by making a number of solutions and combining with each other in the hopes that the solutions they make will be better than the solutions that made them. We made a *GeneticAlgSolver* class in which to implement the algorithm. We started off by making our population, 1000 CandidateSolution objects, and storing it in a vector. We then needed to rank these solutions based on energy, the best/fittest solutions were stored at the start of the vector. We added a *compareTo* method to *CandidateAssignment* that takes a *CandidateAssignment* as argument and compare that assignment with itself based on the energy of the assignments. As we needed to keep the population size the same we needed to cull the solutions with the highest energy. Then we had to decide what members of the population were going to be allowed breed and who they would be allowed breed with in order to make a new generation of solutions. We decided that we would cull the bottom 10% and let the top 10% of solutions mate with the worst, the new bottom 10% of the list. But now we realised we needed to figure out the best way for two solutions to create a new solution. So, we turned back to *CandidateSolution* to add a method called combine that would take a *CandidateSolution* as a parameter and combine that solution with its self to create a child. We did this by creating a new solution with no *CandidateAssignments* in its vector, we then added the first assignment from *CandidateSolution*A and tested the energy of the solution. We then removed that assignment and did the same for *CandidateSolution*B. We then added back in the assignment which gave the solution with the lowest energy. We did this until there were no assignments left to add. We then mated each solution in the top ten percent with a random solution from the bottom ten percent. We then repeated these two steps, sort then combine, 1000 times. We then sort one last time to ensure the best solution is at the top of the vector.

We were then satisfied that we had correctly stored all the data and had used this information to create the best mapping of students to projects we could. It was now time to start the third and final work package. This was to create a Graphical User Interface (GUI) to allow the user to navigate the various functions of our program with ease. We sat and discussed all the buttons we thought our GUI should have. The first of these, of course, was a button to allocate projects but seeing as we had two different ways of allocating projects we had to find a way to choose which one to use. We ran both algorithms multiple times and recorded the average energy of each algorithm. Simulated Annealing was out performing the Genetic Algorithm and so I was chosen to allocate projects.

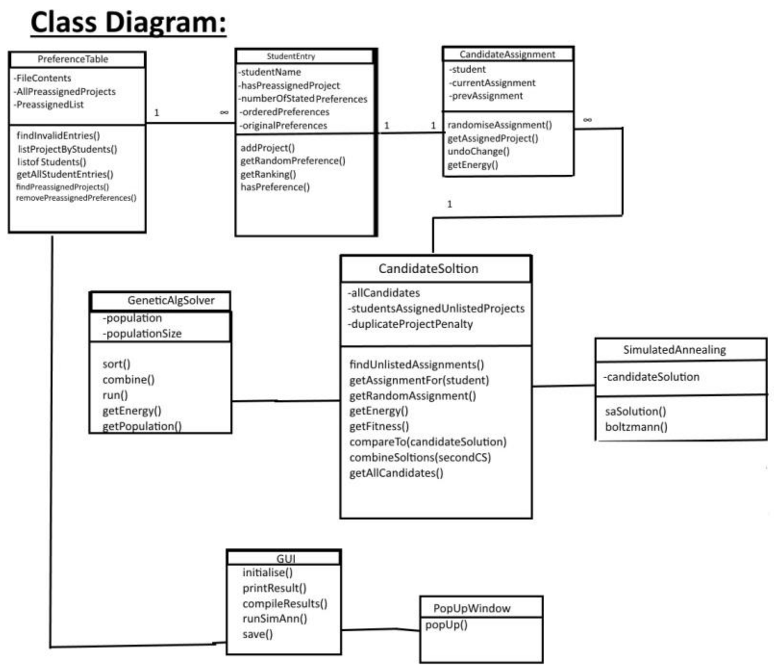
As the user is unaware of how the projects are allocated by the “Allocate projects” button, we decided to also give them the choice of which algorithm to run by adding two more buttons, “Simulated Annealing” and “Genetic Algorithm”. As the user may not have knowledge of the inner workings of our code displaying the energy of a solution will be meaningless so when either algorithm is finished running a new window will pop up declaring what percentage of students got each preference e.g. 50% of students go their first preference, 0% of students got their tenth preference. This window will also give the user the option to save the mapping to a file or discard it. The user can run the algorithms as many times as they wish but if they choose to save more than once the original file will be overwritten. This is to avoid the confusion of having multiple files if the algorithm is run more than once and not knowing which one contains the best mapping.

We also decided to add some useful buttons that would give the user information about the data in the file they are using. The first of these buttons is entitled “Student Lists”, pressing this button will make a popup window appear with a drop down menu. The user then has a choice whether they would like a list of all students, only students with preassigned projects or only students without preassigned projects. Another button we added was “Projects Lists”. This button does the same as the previous except it will print projects names as opposed to student names. Next we added a button called “Projects by popularity” and, as the name suggests, this will print out a list of all the projects in descending order of popularity. After each project name there is a number, this is the number of students that listed that project in their preference list. The next piece of functionality we added to the GUI was the ability to type in a project name and have a list of the students who listed that project returned to them. The final thing we added to our GUI was allowing the user to choose a file that the program will get its data from by searching their compute for a new tab separated file. This allows the user to reuse our software year after year and they never have to edit the code itself.

At this stage we felt that we had done what was required of us by the assignment: To take a file with information about final year students and their final year project choices and search for a way to allocate a single project to each student that would strive to make the satisfaction of each student as high as possible.

Implementation Description

Our project contains a total of eight classes: PreferenceTable, Student Entry, CandidateAssignment, CandidateSolution, SimulatedAnnealing, GeneticAlgSolver, PopUpWindow and lastly the GUI class.



The program starts by creating a *PreferenceTable* of a tab separated file chosen by the user. This entails splitting the file, line by line, and storing each line in a vector.

Each vector was then passed to the *StudentEntry* class, in which a *StudentEntry* object was created. This object included a string containing the student's name, a boolean stating whether or not the student had a preassigned project, a vector with an ordered list of the student's preferences (which could be altered depending on a project's validity as well as how many projects the student listed) and another vector containing a student's original preference list which would remain unaltered for the duration of the program for comparison purposes. Lastly, the *StudentEntry* object would contain an integer value of the number of preferences stated by the student.

Once each StudentEntry object had been created it was time to add methods to each class to manage our newly created *StudentEntries*.

The first thing to check for was for any invalid *StudentEntries*. We already accounted for duplicate preference entries when creating the student's entries as well as a student with a preassigned project listing more than one preference. The only other validity test that we could see was to ensure that nobody attempted to list a preassigned project that wasn't assigned to them.

To counter this we created a list of all the preassigned projects and we then iterated through all the *StudentEntries* and checked each of their preferences to make sure it wasn't contained in our list of preassigned preferences. If it was present in this list, and it wasn't their own preassigned project, we would remove it from their list, bumping every other preference one slot higher.

Once we had assured every *StudentEntry* was valid, it was then time to fill any remaining slots in a student's preference list with projects. We decided to optimise this process by making sure to only add projects that were otherwise unpopular in order to avoid clashes as much as possible. To do this we created a method that would return a list of projects sorted numerically by how many students had it listed. We would then assign the projects at the bottom of the list, provided they weren't preassigned.

This now left us with a list of fully valid student entries, each one filled to capacity with preferences. We could now create the *CandidateAssignment* class that takes a *StudentEntry* and randomly selects one of the preferences that student has listed and assigns it to that student. This class contains a method to get the energy of an assignment, the energy being dependant on the ranking of the assigned project to the student in their preference list. We also included the previously assigned preference in the class to allow us to easily undo a change if it didn't improve the energy.

In order to manage the assignments for all our *StudentEntries*, we created a class called *CandidateSolution* that would store each student's assignment in a vector as well as calculating the total energy of all the assignments, collectively representing the energy of the solution. We included a penalty to the energy in the case of two students clashing and both being assigned the same project. We hoped that when it came to getting the lowest possible energy, this penalty would avoid doubly assigned projects.

CandidateSolution also included a method that would iterate through a solution and create a list of students that were assigned a project that they hadn't listed. This was done by checking if their assigned project was contained in the original preference list we created earlier in *StudentEntry*, if not they were added to the list returned by the method.

Another method included the ability to compare two *CandidateSolutions* based on their energies and returns an integer based on which solution had the better energy. This method is vital to running any algorithms to improve a solutions energy as our remaining classes will show.

It was now time to start optimising our solutions using two algorithms that would each lower a solutions energy significantly. The first of our two algorithms was Simulated Annealing. This algorithm worked by altering one student's assignment within the solution and comparing the new energy with the old. If the new energy was better than the old the new altered solution would be taken on and the process would be repeated.

The number of times the process was performed was determined by a temperature variable that was lowered in a for loop, approaching zero. This temperature was also used in a Boltzmann formula. The purpose of the Boltzmann formula was to occasionally accept a new solution even though it made the energy worse, despite seeming counter intuitive, this helped the energy levels to break out of local minimums and be further optimised.

It then became a case of simply tampering with our temperature and it's decrements in order to achieve the best energies while maintaining a manageable runtime for the user.

The second algorithm class used to improve a solution's energy was *GeneticAlgSolver*. The constructor for this class takes an instance of *PreferenceTable* as an argument and uses it to make new candidate solutions, 1000 to be exact. We then store these solutions in a vector called *population*. Next we have a sort method. We use a collections .sort to rank our solutions by energy. For this we added a *compareTo* method in *CandidateSolution.* This method compares two *CandidateSolution* objects based on their energy. The lower the energy of the solution the higher that solution would be placed in the list.

The next method in *GeneticAlgSolver* was the combine method. The combine method starts off by identifying the last 10% of the population and culling/removing them from the population. The basis on which Genetic Algorithms lie is allowing the fittest members of the population to breed with a select few of the rest of the population in order to create new offspring which will hopefully be better than their parents. It is on this premise we modelled our algorithm. We loop through the top 10% of the population for each of these "alpha solutions" we pick a random solution from the new bottom 10% of the population with which they are allowed to mate. Once we have identified which solutions are going to breed we need a way to mate them.

For this we created a *combineSolutions* method in *CandidateSolution. combineSolutions* takes a *CandidateSolution* as a parameter and creates a new *CandidateSolution* composed of aspects of itself and the other solution. This method starts by creating a new *CandidateSolution* and clearing its *allCandidates* list, making it a blank solution. We then use a for loop to fill the *allCandidates* list again. We start by adding in an assignment taken from the first *CandidateSolution.* We then check the energy of the new solution and store that in a variable called *energyOne*. We then remove this assignment and do the same with an assignment form the second *CandidateSolution,* this time storing the energy of the new solution in *energyTwo*. We then compare these two energy variables, if the first energy was less, then we remove the assignment taken form the second solution and add the assignment taken from the first solution. Otherwise we move on to the next slot in the list. We do this for each assignment in the first solution. We then return a new solution. Once we have combined two solutions, we add that solution to the list and continue to breed solutions until each member in the "alpha solutions" has mated.

The next method in *GeneticAlgSolver* is the driving method of the algorithm. The *run* method will call the sort method followed by the combine method 1000 times, allowing the progress indicator to create a pop up window in 10% intervals. Finally we sort the population one last time and then return the best solution, which will be residing at the top of the population vector.

Lastly, there were a number of features to implement in our program to add clarity to the users such as lists of preassigned students, nonpreassigned, and all students. As well as all preferences, preassigned and nonpreassigned. We also decided to include a feature that would allow the user to enter in a project name and see the list of students who listed that project as a preference. We felt this would allow the user to see any trends in certain students only picking the more popular projects, leading to a lower satisfaction and a worse energy when creating a solution.

It was now time to create the interface for our program. We decided that Java’s built-in interface capabilities would be sufficient for what we wanted to create. When creating the GUI however we encountered some problems in our code so far, our initial plan was to have a default file “Project Allocation Data.tsv” and allow the user to select a new file, however this proved problematic so instead we allowed the user to choose the file initially and have no default file. We also originally had *StudentEntries* being created when *CandidateSolution* was called, however considering how often we called *CandidateSolution* in our algorithm it proved to waste a large amount of time, so the entry creation was moved to *PreferenceTable*.

When our GUI is first initialised the user is prompted to select a file to create the preference table from. After this selection they are brought to our home screen, containing nine options:

- *Choose new file*: Allows the user to choose a new file to create a PreferenceTable from.
- *Allocate projects*: Create a candidate solution using the default algorithm. (We chose Simulated Annealing as our default as it produced better energies in less runtime).
- *Student Lists*: Allows the user to view a list of preassigned, non-preassigned or all of the students.
- *Project Lists*: Allows the user to view a list of preassigned, non-preassigned or all projects.
- *Projects by Popularity*: Returns a list of the projects sorted by popularity in descending order.
- Search box to see which students chose a specific project.
- *Simulated Annealing*: Allows the user to create a solution using the Simulated Annealing algorithm.
- *Genetic Algorithm*: Allows the user to create a solution using the Genetic Algorithm.

Some buttons also feature a hover message that displays information about what the button does.

We wanted to include a progress bar that would show the user that the program was running and hadn't frozen. However, there were multiple difficulties encountered in the threading required in order to produce a functional progress bar, instead we settled for a simpler class that would display a popup on the screen in increments of ten percent as the program ran. We didn't want to clutter up the user's screen for too long so we decided to have the pop up window automatically close after displaying the percentage of the program complete.

Next came the issue of how to display the results of a solution to the user in a way that would be easily readable. We decided to display two pieces of information: The energy of the solution and a list of what percentage of students got each preference. (e.g 42% got their first preference, 21% got their second preference and so on.)

Lastly, we gave the user the option to save these results to a text file for later use or discard them if the user deemed them to not be of any use. After this the user is returned to the GUI's home screen and can run another command or exit the program.

Success and Failures

Throughout this project we experienced both success and failure. In the interim report we discussed all the features that we wanted to implement and, for the most part, all of them were a success. However, there were a few of these ideas that we thought were unnecessary so we decided against them.

The most successful features that we implemented were the two algorithms. The Genetic Algorithm was both a success and failure in a way. To begin with, the combination of the two *CandidateSolutions* was, in the end, successfully implemented although it had to be changed several times to help avoid duplications and also to improve the overall energy that the algorithm produced.

When we first began the algorithm, before the combine method was effectively implemented, the algorithm failed as it produced duplicate projects for some individuals. It would duplicate roughly seven or eight projects which was a serious problem and would give an energy of around seven or eight thousand. Another problem with the algorithm when it was first implemented was the run time. It was taking upwards of two minutes to compile the results of the algorithm. However, after editing the genetic algorithm several times we managed to originally reduce both the run time and the results before the base code was edited which reduced the run time and average energy further. This was a very successful part of the project and even though we tried different ways of combining the two solutions and the way that they “mated” we were unable to get this algorithm to produce lower run times and energies which was a bit disappointing. Overall, though, we believe this algorithm was a success.

The other algorithm that had to be completed was the Simulated Annealing algorithm. Unlike the Genetic Algorithm, we didn’t have any problems when coding this algorithm. As opposed to the Genetic Algorithm, which uses a population of *CandidateSolution*, this is a greedy algorithm that deals with just changing one *CandidateAssignment* at a time and if the energy is better than the previous energy the change is kept, otherwise we ask the Boltzmann function if the change should be kept.

This algorithm was the easier of the two algorithms and so was a complete success. We were very pleased with both the run time and the energies that were produced as, in our opinion, they were very low. There was never a problem with duplicate assignments, unlike with the Genetic Algorithm, and the highest energy that we ever got for Simulated Annealing was around three hundred. Overall, we were very happy with what we achieved with the algorithms in terms of implementing two algorithms that functioned as expected. However, we were slightly disappointed with the gap in results between the two algorithms due to the difference in difficulty between them. Despite this, we are very pleased with what we achieved with the algorithms.

The second core feature that had to be implemented was the GUI. The GUI would be the interface of our project and would allow the user to interact and choose a command. The first successful part of the GUI was to get the different buttons to appear. This wasn't too hard to implement and was completed fairly quickly. The next job was getting the buttons to function. The features that we wanted to implement in the GUI were all done so successfully. These features were: To run the algorithms, get various lists of student names based on whether they had a preassigned project or not, the ability to choose a new file to enter as long as it's a .tsv file, to save the results that the algorithms produced, get various lists of projects whether they are preassigned or not and, finally, to return a list of projects by popularity.

As I have already discussed both the algorithms and the results were a success. The implementation of the student and project lists were also a success although we did have a few problems when trying to implement a drop down menu to allow various choices. One such problem was if the user decided to cancel the operation and not chose any option. In the event of this the program would produce a null pointer error. However, these problems were solved and both menus were implemented successfully. This problem was also evident in the file selection but once again was fixed correctly.

One of the features that we believe was a big success when implemented was the ability to change the file that the program used. This means that our program can work out the mapping of students to projects with any file, for any year, in any college. We believe this was a very useful feature and was a huge success. We decided to implement the methods that dealt with the list of projects and list of students in order to give the user a chance to get various bits of data if they are ever needed.

When it came to the GUI another important feature that we implemented successfully were the exit and cancel buttons. Despite sounding simple these two were very important buttons in the GUI. These are very important buttons because originally when we pressed our cancel button an error would be thrown. This was an error that we encountered but managed to overcome and fix.

Before we could start coding the algorithms we had to implement a couple of methods to allow the algorithms to run properly. The purpose of these methods were to make sure that each of the preferences listed by each student were valid and that each student had listed ten projects. The first of these methods was successfully implemented and the main portion of it was to deal with students who had listed another student's preassigned project as one of their preferences.

Another part of the student validity was to make sure that students who listed a preassigned project only had a single project listed. The two methods to deal with these two features were successfully coded and worked correctly. The purpose of the filling student projects method was to make sure that every student had ten preferences listed. If the student didn't have ten preferences listed then a random preference would be inserted until they had the max number of preferences. This however wouldn't apply to those students who had a pre-assigned project. This method was successfully implemented and it did exactly what we expected of it.

Despite all of the success over the course of this assignment we did run into a failures. The biggest failure was the attempted implementation of the progress bar to inform the user how long was left until the allocation of projects would be completed. This was a complete failure because we couldn't understand how data threading worked in the time that we assigned for this task. Due to not understanding data threading it meant that we could only execute one method at a time instead of multiple methods.

However, we did managed to come up with a solution so the user would be able to see the progress of their algorithm. We decided that an automatic pop up window that would also automatically close would be an alternative solution. The window will pop up and tell the user what percentage of the algorithm is complete. This was a very big success and was a good alternative to a progress bar. With this window we did encounter problems such as the time it should be on screen for etc. We decided on a timing system that we believed was sufficient for the user to read the pop up but not long enough to delay the algorithms by too much(It adds about 5-10 seconds). One problem that occurred with the pop up window was when the user would press the "Okay" button the next time the window popped up a null pointer exception would be thrown as the program was trying to close a window that had already been closed. However, this was successfully fixed by just inserting a try catch statement where the null pointer exception was being thrown.

Another feature that we had originally planned to implement but decided against was the idea of allowing the user to enter a satisfaction range. This would first allow the user to choose a range e.g. 100 – 250. This range would then be used when the energy was being calculated in the algorithms. The algorithms then wouldn't stop running until the energy produced was within the range that was entered by the user. This feature was scrapped as we agreed that the user may not understand what the energy is. We also thought the user should get the best solution, not one that they think is good. If we had decided to include this then the user would also need to know the code and understand how the energy is calculated. We agreed that the user should not be concerned with how projects were allocated and should judge the solution not on the energy but on the percentage of students that got their first preference, second preference etc.

Another aspect that we opted not to implement was the idea of a project solver interface. The point of this class would essentially be the superclass for both the algorithms. However the only methods that would need to be in this class would be the run method to run the algorithm and the compile method to compile the results that the algorithms produced. The reason that this class was scrapped was because we thought that it would be unnecessary to have a class with just two methods and would look neater in the GUI class. Both algorithms do share a printing and compiling method, both of which are in the GUI instead of a superclass called project solver interface.

Another obstacle we encountered was the problem with spelling mistakes in the file that we were giving. Since we didn't have access to a master project list we were unable to determine how to identify where the spelling mistakes were in the file and then how to fix them. We also didn't have the time to address the situation due to other features that we wanted to implement and we didn't notice these errors until later in the project. This was the biggest failure that we encountered as it can impact the overall results that are given. We are aware that not addressing the issue of spelling mistakes could possibly lead to duplication of projects but, once again, without the master list we simply just didn't know how to fix this error quickly.

Throughout this project we came across many features, some more successful than others and some that we decided to scrap for various reasons. As a group we made important decisions of which features had to be implemented. Communication was the most important skill that we needed when implementing this project. The communication was necessary to decide which methods and features we thought should be included and which ones should not. The decisions that were made were made as a group and despite different opinions on certain features every member of the group in the end was content with the final product.

Another success that we had as a group were the regular team meetings and the deadlines that were set in these meeting. For all of the methods that were implemented throughout this project almost all of the deadlines were reached on time. Only on one or two occasions did the deadline that we had set at the beginning of the project have to be extended. The weekly group meeting were also a success as it allowed every member to express their opinions on the progress of the project. The only problem that we experienced with communication over the course of the project was when every individual of the group had a different opinion on how a certain feature was going to be implemented. Nonetheless, through discussion, we were always able to find a solution that we were all satisfied with.

Overall, however, we feel that that the project was a success and with minimal failures. However due to the fact that the Genetic Algorithm was more difficult than the Simulated Annealing we are a little disappointed that we couldn't get the run time and results to be lower. However we do feel that the results of the two algorithms give a very fair result to all of the students even if the Genetic Algorithm could be improved, and feel like it would work very successfully if ever actually used.

Additional features

The purpose of the assignment, as stated above, was to find the best mapping of students to projects. In order to ensure that the results returned from our algorithms were accurate we had to add a few features to ensure that all of the data we were working with was valid. We had to ensure that each student had filled out their preferences correctly, which not all had. A student could list less than ten preferences or have listed a preference more than once, some students listed another student's preassigned project as a preference, and a student with a preassigned project could have listed other preferences as well. These are all ways a student's information could be invalid. As we felt that we could not create an honest mapping if not all the students had listed honest information, we created various methods to deal with each of the problems detailed above.

If a student listed a preference more than once or had listed a preassigned project that was not theirs then those projects would have to be removed from their list. As some students had not listed ten preferences and some students had preferences removed we had to give these students new projects until they had ten. This is where our first main additional feature arose. We thought we could help lower the energy of the overall solution by filling students' lists with the least popular projects. For this we had to search through every student's preference list and count the number of times each project was listed. We then stored and sorted the projects based on popularity. When we encountered a student with less preferences than required we would add the least popular project to their list. We then increased the popularity of that project by one and sorted again.

Our next additional feature was a follow on from the first, this was to not only store the popularity of the projects but also store the students who chose each project. We then give the user the ability to search for a project and have a list the students interested in that returned. We thought this could be a useful piece of information for the user.

Another useful feature we added was the various lists the GUI could print. We thought the user might want to know certain information about the data the program was using. There are two buttons on the GUI that do this. The first of these buttons is entitled "Student Lists", clicking this button will make a new window appear with a drop down menu, on this menu the user can chose whether they want to know which students have preassigned projects, which students don't have preassigned projects, or just want to know the names of all the students. The second button reads "Projects Lists" and works much the same as the first: It gives the user the choice to look at a list of preassigned projects, nonpreassigned projects or all projects.

The next feature our code implements is to do with the file the program gets its data from. As the purpose of this project is to build a piece of software that can be used to allocate projects in the future, it needs to work on many different files not just the file we were give. When the application is run the user first has to choose a file from their computer from which the program will extract the data. This file must be a tab separated file, if it is not, the user is forced to choose again. This feature means our software can be used for years to come and the user will not have to change the code.

Another feature we added was a progress indicator. As the Genetic Algorithm can take some time to run we decided to add a progress indicator to our program to reassure the user that the application was still running and that progress was being made. We opted to inform the user about the progress of the algorithm in the form of a pop up window, for reasons discussed above. A window will pop up when the algorithm is started and will remain on the screen for three seconds. The window will then disappear if the user has not already cancelled the window. The window can be cancelled in two ways, the first clicking the “ok” button located at the bottom of the window and the second by clicking the red “X” in the top right hand corner of the window. The window will continue to pop up in 10% intervals informing the user that the algorithm is 10%, 20%, 30% etc. complete.

The final feature we added was the option to save the final mappings produced by the algorithms. As the algorithms work mainly with randomisation, there are countless different mappings that our program can produce. The save functionality may seem to be a fundamental part of our program as opposed to an additional feature but we believe the way we save makes this untrue. Every time an algorithm has produced a mapping, we give the user the option to save or discard, instead of automatically saving every time. We did this as the user might want to save a solution but continue allocating projects to see if they can find a better solution. Due to the random nature of both algorithms there is no guarantee that running the algorithm more times will give a better result. Having save as an optional feature allows the user to keep running the algorithms and only save if/when they find a better solution, otherwise they can discard it.

Team Analysis and Work Breakdown

We split the jobs at our first meeting based on what we planned in our interim report. The jobs stayed mostly the same with some people switching a job or requiring assistance. We split the algorithms into two sub teams and flipped a coin to decide who got what.

Mark

Mark was given the first job to be completed: Checking the students' entries and finding errors. He wrote the code that ensured that a preassigned student didn't accidentally enter more than one project and that no one listed the same project more than once. Karl found that some students were listing another student's preassigned project as one of their preferences. He then helped Mark plan the code. Mark then wrote a method to find those students, remove the preassigned projects and add a new one to the end. This job was submitted quite late as the list was not checked properly for invalid entries.

Next he worked on creating lists of students. He made a list of all students as well as ones for those with preassigned projects and those without. These would later be printed in the GUI by Caroline.

The next job Mark had been assigned was discarded before he started. The group thought entering a range for the energy would not be useful as the user would more than likely want the best solution and they probably wouldn't understand how our energy was calculated.

After that Mark worked with Caroline on the Genetic Algorithm. They split the algorithm into different jobs. Mark wrote the combine method in CandidateSolution and they both worked on the combine method in the Genetic Algorithm class together.

When the algorithms were done Mark had the final task of creating a progress bar. Though he couldn't manage to get threading in the GUI he worked around it and created popup windows that would inform the user of the algorithm's progress in 10% intervals. After this Mark helped with testing the final code and fixing small bugs.

Niamh

I was in charge of the GUI which involved creating it and implementing all of the buttons. I was assigned this task with Caroline though so left the printing to her. I also designed the GUI and carried out the final testing via the GUI with some help from Mark.

Karl and I then worked on optimally filling all the student preference lists with less than 10 projects. Karl did most of this though as he had worked on a method that could be used here. Next, I was given the job of creating lists of projects. I implemented a method that would print all of the projects and those that were either preassigned or non-preassigned to be used later in our GUI as an extra feature.

The final task I was assigned was to work on the Simulated Annealing algorithm with Karl. I took the lead on this one as I had a slightly better understanding of it but we worked fairly close together regardless of this. After the algorithm was completed I did further work on it in order t) speed it up while ensuring we still got the best result possible.

After this, I took over Caroline's job of allowing the user to select a file to use. I also helped Caroline to print the final mappings to a file and edited parts of the code we started with to speed up both algorithms. After all this I fixed small bugs in the code along with Mark during the final test.

Karl

Karl's first job was to create a list that contained every project and sort them by popularity. This feature was later implemented in his second job with me when we had to optimally fill the preference lists. Karl took the lead using the list he had created, as mentioned above.

Next, he worked on the Simulated Annealing algorithm. As aforementioned, I did take the lead on the algorithm but Karl helped to come up with the best Boltzmann function that we could create. After this, he created a method that would allow a user to see who chose which project by inputting the project name and getting a list of students that chose it.

Finally, he made a method that would inform the user if a specific student had not been assigned any of their chosen preferences. He also edited the base code in order to fix some of the methods to optimise the run time. This also sped up the Genetic Algorithm which previously had a very poor run time. I wrote the README file too.

Caroline

Caroline's first allocated job, not allowing duplicate assignments, was dropped in a team meeting before she got started when we realised that our algorithms should not allow these duplications. She was also assigned to work on the GUI with me. She created most of the popup windows to display information and implemented scroll options and choices in the pop ups.

Next, she worked on the Genetic Algorithm with Mark. She wrote the run and sort methods for their algorithm as well as worked on the combine method in their class. She gathered and calculated what percentage of students got their first-last preferences from the algorithms and after this she was given the task of printing the final mappings to a file for the user. She wrote the GUI part herself that gave the user the option to save or discard the mapping given.

All

We were all given the task of cleaning the code and each member took two classes as we had eight in total. Every member was involved in writing up the report. Mark wrote about the success and failures and helped me in the proofreading of the report. I wrote about the team analysis and work breakdown as well as formatting and, as mentioned above, proofreading the report. I also added the table of contents and conclusion. Karl wrote the implementation description and Caroline wrote about both the project specification and the extra features.

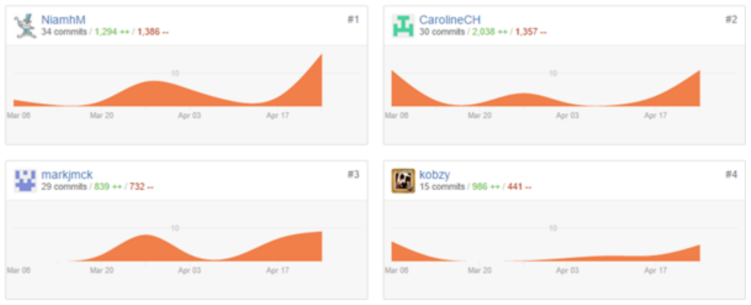
Work Breakdown

This is the table of the jobs assigned including when they were finished:

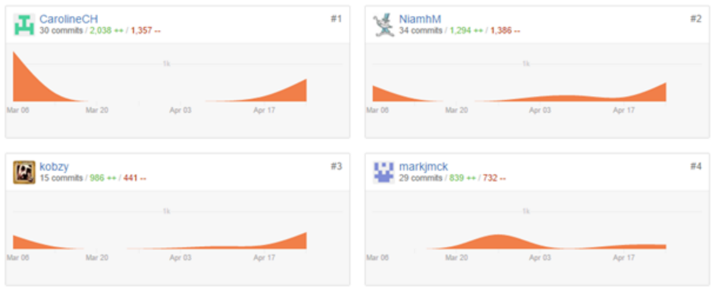
Job	**People**	**Deadline**	**Completed?**
Student Validity	Mark	30th March	20th April
No Duplicate Allocations	Caroline	4th April	**No**
Projects by Popularity	Karl	13th April	12th April
Optimally Fill Projects	Karl & Niamh	15th April	12th April
Student Lists	Mark	18th April	20th April
Energy Range	Mark	18th April	**No**
Project Lists	Niamh	18th April	7th April
Simulated Annealing	Niamh & Karl	20th April	26th April
Genetic Algorithm	Caroline & Mark	20th April	27th April
Student by Project	Karl	23rd April	25th April
Progress Indicator	Mark	25th April	27th April
Print Results	Caroline	25th April	26th April
GUI	Niamh & Caroline	25th April	29th April
Upload new file	Niamh	25th April	27th April
Exit Button	Niamh	25th April	15th April
Print Final Mapping	Caroline	25th April	28th April
Final Test	Niamh	29th April	30th April

These were the graphs produced by GitHub:

Commits:



Additions:



However, these graphs are not completely accurate in showing how much work was done by each individual as people were pushing the original classes to use and sometimes people were pushing the same code repeatedly with small changes in an attempt to fix errors. Also, on sub team jobs only one person was pushing to GitHub.

Our push history and code can be viewed at: *https://github.com/NiamhM/The-Shady-Bunch-SEP3.* Viewing the history gives a better idea of work breakdown.

Conclusion

Overall we’re fairly happy with how the assignment went. Our Genetic Algorithm was a slight disappointment but we tried to fix it as best we could in terms of runtime and the actual results. We are pleased with how our GUI turned out but acknowledge that threading would have improved it.

If we could change anything we would definitely have started our algorithms sooner. Though Simulated Annealing worked out well regardless of time the Genetic Algorithm suffered. We also would have looked into threading sooner and perhaps allocated two people to look into it though both Mark and I tried to implement it on separate occasions but time was not on our side.

We worked together well and, as a result, greatly improved both our teamwork and communication skills. Though sometimes we missed some deadline we supported one another as best we could to meet them and were understanding of any complications. I feel we performed well as a group and hopefully this shows in our final product