Development Cycle:

During our First meeting we discussed how our project would be organized, and the overall goals that we hoped to achieve.

Initially we were moving towards a diamond shaped structure, with our genetic and annealing algorithms interacting directly with the GUI and also creating the output. But further investigation into the requirements of the project led us to use a structure with much more flow.

With the overall picture in our heads of the direction we were going we began development the 2 algorithms and the GUI, with each of us being assigned a particular section.

The Annealing Algorithm:

With the annealing algorithm, we searched for an overall optimum solution as opposed to an actual best solution. The three main areas for development were the temperature, the cooling amount, the probability of accepting a solution and the algorithm that drives everything.

The class was first written entirely in pseudo code, which allowed for any logical obstacles to be addressed and overcome before implementing any code. Helper methods were created as required with the pseudo code, with the possibility of these being added to or removed as the program functionality was tested.

The algorithm was then written in full. The Boltzmann algorithm was used to help us find our solution. Annealing works similar to a hill climbing search, however once a local max is found, there is a possibility to keep searching and to find an even greater result. To calculate the probability of the change in solution being accepted, we created a separate method to include the Boltzmann distribution value for the energy and temperature of a certain solution.

It is here that the Temperature and cooling come into play. The temperature is set and applied to the algorithm, the higher the temperature then the looser the algorithm and the more likely that a bad solution is accepted. The cooling amount is applied to the temperature meaning we are less and less likely to accept a poor answer.

We initially created a helper method to randomize an element of the solution. Keeping in line with our original ethos and attempting to maintain a good flow with readable code, this evolved to be a very small method. The choice was made to included it in our go method instead. The notion of a ‘go’ method, meant that while looking through the code, it was easy for any user to track what was happening should any revisions need to be made.

The program initially had constant values for temperature and the amount to be cooled by each time. In testing this, these were altered to determine what combination of variables achieved the best results in under a minutes computation time. This was then altered to allow the user to enter these values as desired through the CLI.

After this was finished, we merely added in functionality to alter the progress bar and allow for status updates as the algorithm begins and ends, before cleaning up the flow of the code and ensuring all variables were global/private as required.

Development cycle of the genetic algorithm.

Getting it working:

With a strong frame work in place we were able to move on to developing the algorithm which would generate our solutions.

For the genetic algorithm, generations are the key to improving a solution. There were three main steps to consider. Generating a population, mating the best candidates, and replacing the weakest members of the population with the children created from the mating process. This finalized group would then become the population for the next generation.

The idea is that each generation will become stronger and yield a good result.

A good result in the scope of our program would be considered one with minimal energy. Energy was chosen as a marker for consistency throughout the program with both the genetic and annealing algorithm referring to it.

Creating the population:

Creating a population was a relatively easy step due to the strong framework which we already had in place and which was discussed in our interim report.

GeneratePopulation is the function which we designed to create an ArrayList of randomly selected CandidateSoltions. Using this type of data structure allowed us to easily amend our population by sorting it and adding/removing members which will become important once the algorithm has run.

This strong basis meant that this function remained consistent throughout the development process and changed very little from beginning to end.

Finding the fittest group:  
In order for our solution to improve in each generation, we needed to find the best members of the population (or the members which have the lowest energy scores). The Collections class allowed us to compare each member of the population based on their energy simply by implementing a compareTo method within the CandidateAssignment class. Once the population was sorted on this criteria it was simply a matter of selecting the top N elements of the population, with N being an even number, the size of which we would like our fittest group to be. Initially these were set as constants for ease of testing.

Finding the weakest group:

With the population already sorted, finding the weakest group was trivial, and ArrayList contained all of the tools necessary for removing the lowest elements and adding in the children created during the mating process.

Mating:

Mating was the most complex element of this algorithm and as such was subject to a lot of change throughout the development cycle. Firstly we needed to decide what mating actually meant for this program and how it would ultimately improve the solution.

As a group we discussed the different elements. Taking into account the project specifications and what we ultimately wanted to achieve. We came to the conclusion that the mating should happen on the level of CandidateAssignments as this is where the energy really originates.

First run

Each member of the fittestGroup was paired with their neighbor in the array. This meant that best solutions stayed with the best solutions. The idea being that these would in turn result in the best children.

Our algorithm took the two parents, both of whom are CandidateSolutions, and created a new CandidateSolution child based on the best elements of these parents. The CandidateAssignments within the parents are always listed in order. This meant that we could iterate through the parents and compare each Candidate assignment.

Additional code was added to the CandidateAssignment class which would allow us access to each students satisfaction. Whichever student had the better satisfaction score was then used in creating the child. This meant that the child was essentially a combination of the most satisfied students.

Result.

This process resulted in a relatively poor score and two large obstacles became apparent.

A large penalty is applied whenever there is a duplication in project selections among students. Our algorithm did not take into account a project allocation already existing within the child solution. So, while the solution was better for the individual student, The overall energy produced by the solution incurred a huge penalty.

The other big problem which occurred was that as our list of fittest solutions was sorted every generation, often the same parents would produce identical children each generation. This meant that our solution became full of duplicates and stopped improving after only a few generations.

Round2:

While the overall approach to mating was sound, our implementation was flawed. These situations needed to be handled.

The algorithm was restructured completely while retaining the core ideas. The individual Candidate assignments were still compared but other checks were applied.

We introduced a ChildCandidateSolution class with the function energyWith(). This allowed us to check whether the overall energy would improve, preventing larger penalties from incurring.

To prevent the possibility of identical parents producing an identical child, the fittest group was shuffled and a new check was introduced. Before each child can be added to the children list, its energy must be greater than the energy of both parent1 and parent2.

All of the children who don’t make the cut are drowned in a river and discarded.

Result:

Immediately results were much better. With energy dropping from the thousands into the range of 300-400

Mutation:

An element of the Genetic algorithm is the idea of mutations. A mutation is a random adjustment made in some way to the solution.

We initially introduced a random student assignment (with adjustable mutation frequency) and moved this onto a random candidate assignment added into the population.

After running numerous tests with both of these possibilities, and various mutation frequencies, we discovered that no improvement to the final solution was made. In fact, the solution often fluctuated resulting in poorer final results.

After discussing the possibilities we decided as a group to remove mutation entirely. While it is an interesting element of the genetic algorithm, it took us further from our goal of achieving a good low energy score.

Refinement:

One of our mission statements was to create clean, readable code. Once we finished the main coding and testing, we then proceeded to run through the code, being extra critical about its ordering and functionality.

The names of the functions were refactored to reflect the modularity of those functions. Encapsulation was refined to ensure elements were public only if required.

Our constructor was then parameterized in order to allow adjustments and commands to be input through the CLI and GUI.

The GUI and CLI: