Team name: KPMGComeGetUs

Name: Sng Hong Yao **Student ID:** 17205050
Name: Ankish Raj Prajapati **Student ID:** 17202456

NOTE

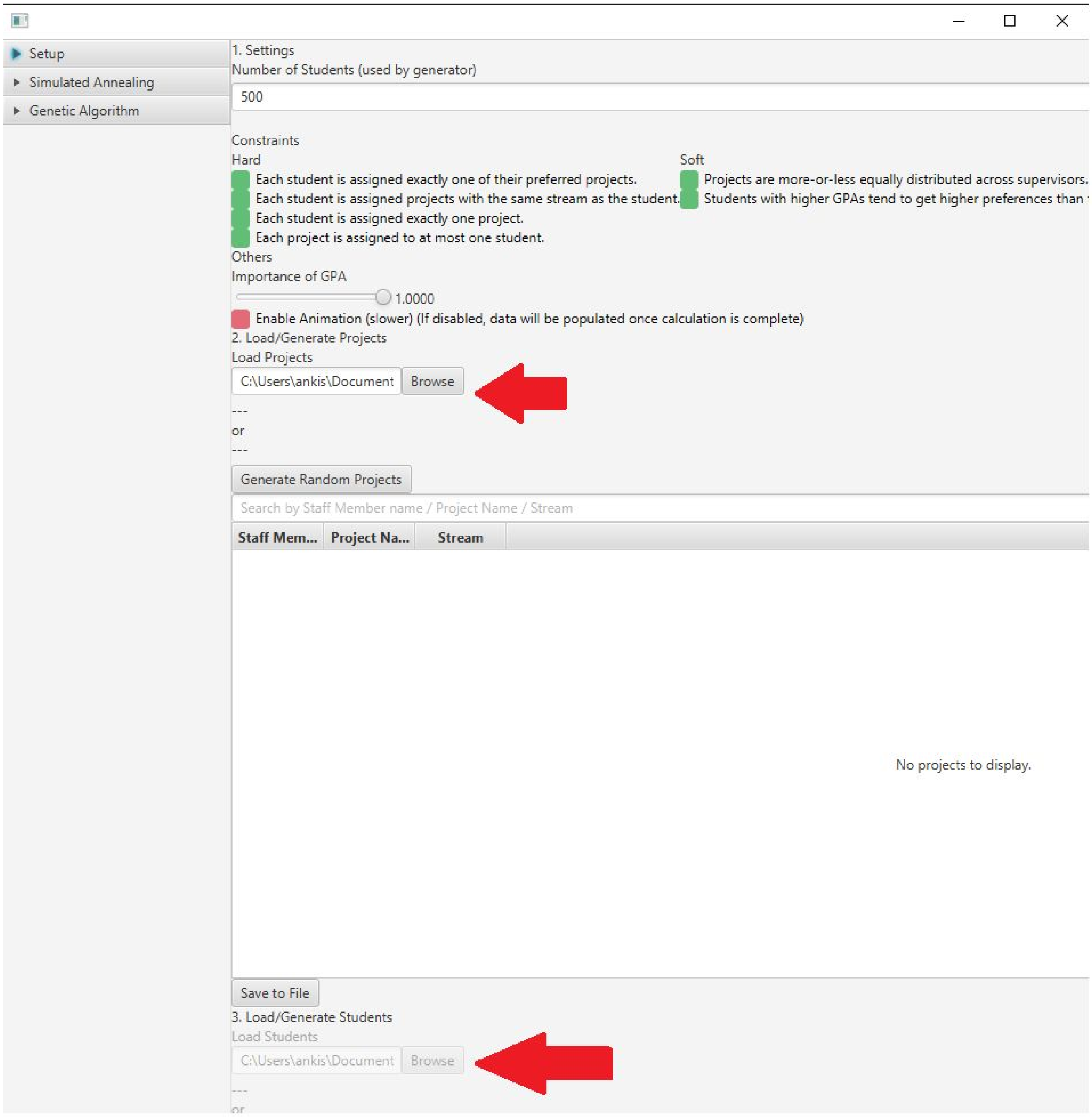
1. We are not too sure how you will be marking this (in terms of when it is running). Therefore we did not write anything in main, but a launch JavaFX application.
2. The graph runs Genetic Algorithm using everything required in this sprint.
3. Due to the memory requirements and UI lags that the animation will cause with these GA parameters, animation has been disabled. The graph will be displayed after the processing is complete.
4. You can specify parameters for the Genetic Algorithm from the user interface. However, the default parameters are (why these are used are explained below):
    - Number of generations = 200
    - Size of population = 1000
    - Mutation chance = 20%
    - Crossover chance = 70%
    - Stating cull percentage = 25%
5. Using these parameters will take awhile. This is due to the population size and number of generations. Max 4 minutes. There is a progress indicator showing the processing thread's progress.
6. Once it is done computing, give the UI awhile to update the graph and sheets.
7. After the UI is updated, the UI will feel sluggish. This is due to the constant updates JavaFX needs to render the LineChart object. Once you are done observing the chart, you can click on "Clear and Reset". This will clear the graph, and make the UI more smooth again. You can then move on to Inspect the solution, or Save to File as a CSV to inspect further.

Setting up Environment

1. Open a terminal and navigate to the file where run.sh is.
2. Execute ./run.sh (Uses maven to build the JAR, then Java to run the JAR)

Instructions To Run - *Very Important!*

1. Our GUI is still in the development phase so it's not perfect yet but has a lot of the functionality required in future sprints. It's not very user friendly right now but we will improve that as we move forward. We included it in this sprint because it helps give us a clear visualisation of how our Genetic Algorithm improves over time.
2. Here is how the UI looks when it starts up:



1. Settings

Number of Students (used by generator)

500

Constraints

Hard

🟩 Each student is assigned exactly one of their preferred projects.

🟩 Each student is assigned projects with the same stream as the student.

🟩 Each student is assigned exactly one project.

🟩 Each project is assigned to at most one student.

Soft

🟩 Projects are more-or-less equally distributed across supervisors.

🟩 Students with higher GPAs tend to get higher preferences than

Others

Importance of GPA

1.0000

🟥 Enable Animation (slower) (If disabled, data will be populated once calculation is complete)

2. Load/Generate Projects

Load Projects

C:\Users\ankis\Document Browse

---

or

Generate Random Projects

Search by Staff Member name / Project Name / Stream

Staff Mem...	Project Na...	Stream

No projects to display.

Save to File

3. Load/Generate Students

Load Students

C:\Users\ankis\Document Browse

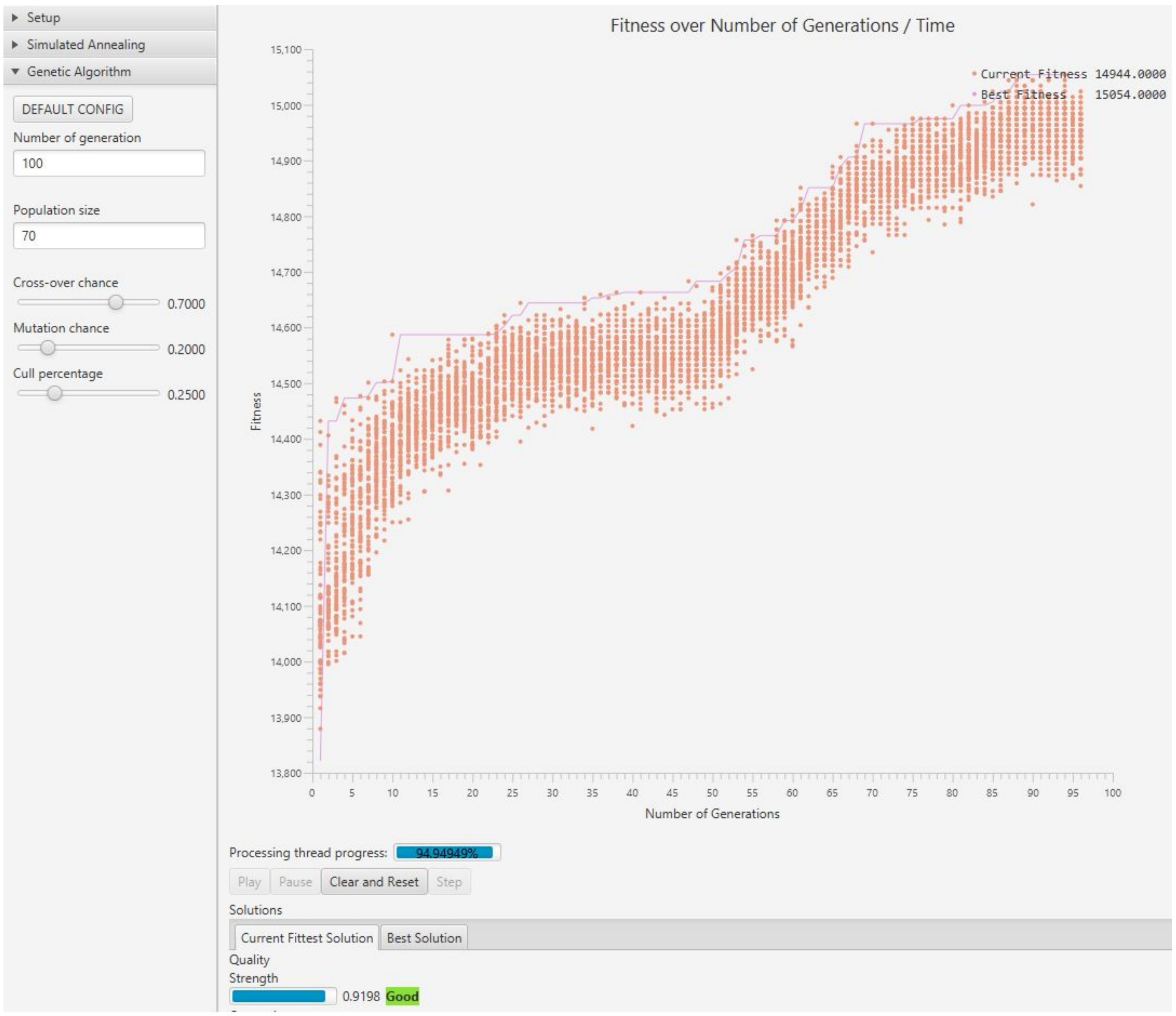
---

or

3. As marked by the red arrows, this is where you can click to load CSV files into the program. Click on Browse under Load/Generate Projects and *navigate to “src/main/resources/CSVs/ProjectsCSV500.csv”*.
4. Similarly under Load/Generate Students, click on Browse (may seem inactive but it’s not, will be fixed in the next sprint), and *navigate to “src/main/resources/StudentsCSV500.csv”*.
5. Once the files are loaded into the system, which happens in a split second, you should see the project and student information displayed in their respective tables.
6. Go to the left pane and click on Genetic Algorithm, scroll down and click on “Play” to begin simulating the algorithm. Wait for some time as the processing gets completed, after which the program will wait a few seconds as it plots the graph. If the progress bar displays 100% at the end, it means the program ran through all the generations and if it stops before that and displays the graph, it means a plateau was

reached and the processing ended earlier. We will remove this and add a better indicator for plateau in future sprints.

7. The graph you see gets populated only after the processing is done, as is shown by the Progress Bar. Real time graph plotting is off for now to ensure little to no lag. The number of coordinates displayed are also limited to make the UI a bit quicker.
8. After a simulation is complete, you can use the Clear and Reset button to reset the graph and begin a fresh simulation.
9. The parameters on the left are the default parameters that can be edited if you wish to. If you want to see a full simulation from first generation to last on the graph, you can reduce the population size from 1000 to 70. Obviously, this reduces the quality of the solution as well.



Number of generation: 100

Population size: 70

Cross-over chance: 0.7000

Mutation chance: 0.2000

Cull percentage: 0.2500

```
Current Fitness 14944.0000
Best Fitness    15054.0000
```

Processing thread progress: 94.94949%

0.9198 Good

10. Lastly, the orange/red dots display the satisfaction of each member of a single generation and the violet line shows the best overall satisfaction achieved yet. The Current Fittest Solution at the bottom is the fittest solution found in that generation and the Best Solution is the overall best solution found in any generation. As you can see from the given screenshot, the graph is plotted but the progress is at 94.94% which means that a plateau was achieved in this case.
11. As is evident from the graph, initially the solutions have a much bigger range and are much more sparsely placed on the graph, showing that there is a lot of diversity and as the algorithm goes on, the solutions

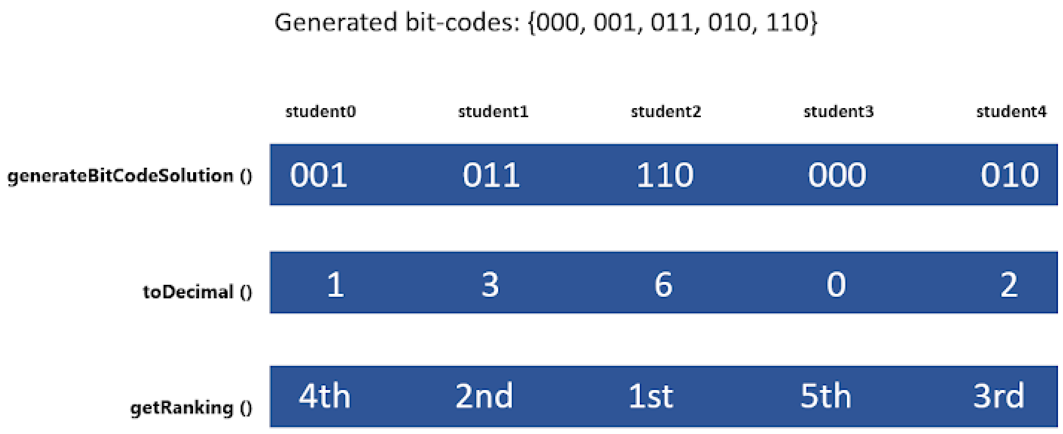
are more and more closely plotted which means they are getting not just better in terms of fitness but also more and more consistent. This proves that our Genetic Algorithm works and does not produce degenerate offspring.

Meeting Demands

- **Generating bit-codes**
    - We begin by generating 10-bit gray codes which will represent a chromosome in our bit-code solution. These are sufficient for representing a solution of up to 2^10 or 1024 students. We generate as many bitcodes as the number of students generated.
    - We start with "0000000000" as the first generated bit-code. We flip one random bit to generate the next bit-code (gray code), repeat until we have enough bit-codes, and maintain a Hashset of used bit-codes to ensure no repetition.
    - Each solution is a combination of these bit-codes and contains as many bit-codes as there are number of students, so each solution contains 10 x (number of students) bits.
    - These bit-codes are contained in an ArrayList of String type. The reason for this will become obvious when we get to mating.

- **Creating generation zero** (GeneticAlgorithm.generateInitialPopulation())
    - We produce an initial order of bit-codes which is used to create our first solution. This order of bit-codes is an array which contains integers from 0 till one less than the number of students. Each integer in the order represents the index of the bit-code that will go in that position. For example, if we have three students, our ArrayList of bit-codes contains three bit-codes
      (000, 001, 011) and the order of bit-codes is (2, 3, 1), the solution generated will be 001/011/000. The slashes are only placed here for clarity, in the program the solution String will simply be 001011000.
    - To further populate this generation, we swap the position of any two random integers in the order of bit-codes and generate the next solution. From the previous example, say we swap and get the new order as (1, 3, 2) where 1 and 2 swapped positions with each other. This will give us a new solution.
    - We do this until we have enough solutions as required in a single generation. We generate this before we enter the main generation loop of our program to give us a starting point.
    - Each solution is also passed through the mutate function to encourage diversity.

- **How bit-code solutions are converted to project assignments**
    - Each bit-code in a solution represents a rank for the corresponding student. To get this rank we convert the binary code to a decimal number and then rank them from highest to lowest.



- The ranking represents the order in which the students will be assigned a project. For each student, in order of the rank, we use their preference list to assign them the best available project and keep track of projects that have already been assigned using a Hashset.
- In case no project from the preference list is available, we will keep a track of such students and once everyone else is allotted a project, these students will be allocated a random project that has not been assigned to anyone yet.
- Following on from the previous example, let us consider the following example.
  Projects available: {P0, P1, P2, P3, P4, P5, P6}
  Preference Lists:
  Student0 : {P1, P3, P4} : 4th rank
  Student1 : {P3, P2, P6} : 2nd rank
  Student2 : {P3, P4, P1} : 1st rank
  Student3 : {P0, P1, P4} : 5th rank
  Student4 : {P3, P2, P5} : 3rd rank
  Since Student2 has 1st rank, we start in their preference list and since the first preference P3 is available, we assign it to them.
  Student1 has 2nd rank so we go into their preference list and see that their first preference is unavailable so we assign them P2, which is the next available preference. Similarly for Student4 with rank 3, the first two preferences are unavailable so we assign them P5 and so on.
- This also shows why a crossover population will still be valid because effectively we are just producing ranks for the students and hence literally any combination of binary numbers in a solution will give us a valid result.
- An important reason for choosing this design is with the foresight that we will have to later on also take into account the GPA while allocating the projects. This ranking system will help us because each student can be assigned a weight based on this rank. We can then convert each student's GPA into a weighted measure and add these to the previous

weight of the corresponding student, so students with better GPA will get a bigger boost of weight and can overtake a student above them in the ranking who may not have as good a GPA.

- **Creating and managing subsequent generations, begin main loop**
    - Now that our initial population, stored in currPopulation has been generated and we understand how our bitcodes represent an actual solution, the main loop of the program to generate the subsequent populations will begin.
    - When we enter the loop, first we calculate the satisfaction/fitness for each member of the currPopulation and also keep track of the fittest solution in this population. A bit-code code solution and its corresponding fitness are stored in a BitCodeSolution object and the population is an array of such objects. The decision to use an array was made because it makes it easier to sort the solutions on the basis of their fitness which is especially useful when we cull the population.
    - After we have evaluated the population, we compare the fittestSatisfaction with the bestSatisfaction, which stores the last known best satisfaction of all generations.
    - We use this bestSatisfaction and check if a plateau has been reached (more on this later).
    - We sort the population in descending order of fitness using a sort method that employs an anonymous function consisting of Arrays.copy and a lambda. We then cull a certain number of bottom members based on the cull percentage.
    - We then proceed to create the population for the next generation. This is the inner loop of our algorithm.
    - We use the populationAfterCulling pass it to the generateNextPopulation function which returns a new population, which we store in currPopulation in preparation for the next iteration.
    - We also increase the cull percentage, so more and more members are culled each time. This can go up to maximum 97% or 0.97.
    - Again, we go to the beginning of the main loop, calculate the fitness for each member, and repeat the whole algorithm until either a plateau is reached or we have created as many generations as we needed.
    - After we exit the loop we store the last bestSolution is stored as our finalSolution. The Genetic Algorithm simulation is now complete.

- **Generating the next population**
    - Done in generateNextPopulation(). To do this, we choose any two random parents, mate them based on the crossover/mating probability, mutate the offspring, again based on mutation probability, and if crossover occurs, add it to the nextPopulation. We repeat this until we have enough members to populate the next generation.

- **How probabilities are handled**
  - To check for a probability, we multiply the required probability, stored as a double in the range of [0.0, 1.0] , by 1000 and store it as the bound, and then generate a random integer in the range [0, 1000).
  - If this generated number is less than the bound, our probability is true.

- **Culling**
  - We use a combination of the Arrays.sort function and a lambda to sort our array of which contains members of the population.
  - We then remove the bottom (cullPercentage * sizeOfPopulation) elements before mating occurs.
  - Cull percentage can go up to a maximum of 97% which means that eventually only the top 3% are allowed to mate.
  - We also maintain a variable called cullPercentageIncrementFactor which is final, and is calculated using the formula (1 - cullPercentage) / numberOfGenerations, rounded off to three decimal places. This is added to the cullPercentage at the end of each main loop.

- **Crossover/Mating**
  - Crossover chance is set at 70% or 0.7.
  - Because we store each bitcode solution as a String inside its object, mating becomes very straightforward. We pick a random index in the range [0, length of String - 5) and use that as the crossover index.
  - We subtract the 5 to ensure that at least 5 bits are taken from the second parent.
  - We use the String substring method in Java to get the bits of the first parent from 0 to crossoverIndex and of the second parent from crossoverIndex till the end and join these using the String concat method to produce our offspring. This is all dependent on the crossoverChance probability.
  - Note that at the beginning of the crossover() function we initialise offspring with a blank string (“”) so in case mating doesn’t occur because the probability didn’t allow it, a blank String is returned. There is a conditional check when we use this function in run() and so an offspring only gets added to the nextPopulation only if it is not blank.

- **Mutation**
  - Again, this is dependent on the mutation probability which we have set at 0.1 or 10%.
  - To mutate a solution, which is a String, we convert it to a character array which makes it very convenient to change a single bit.
  - We then choose a random index in this array, flip the bit at that position, convert this array back to a String and return it. If mutation doesn’t occur the same String is returned so we don’t need any additional checks when we use this function anywhere.

- **Checking for plateau**
    - We've used four main parameters to achieve this.
        - isPlateauReached (boolean) - true if plateau is reached. Initialised with false.
        - plateauPercentage (double) - set at 0.40 or 40% - this is an indicator for the program to know where it should start looking for a plateau from. We do not want to be looking for a plateau too early because there are still a lot of generations to go and an improvement may be achieved. This parameter also gives dynamism and great flexibility if we want to change or disable this feature.
        - plateauCheckFrom (int) - Indicates the generation number from which the program starts looking for a plateau. It is calculated as (1 - plateauPercentage) * numberOfGenerations. So if we have 100 generations, the program will start looking for a plateau from generation 60 onwards if plateauPercentage is 40%.
        - minRepititionsForPlateau (int) - Indicates the number of times a fittest satisfaction must be repeated to be considered a plateau. This is currently set to

        - (0.2 * plateau percentage * numberOfGenerations). So if we have 100 generations, with plateauPercentage at at 0.4, we should get a minimum of 8 repetitions of our current fittestSatisfaction to consider it to be plateaued.

    - We maintain an ArrayList of CandidateSolutions type called possiblePlateauSolutions to keep track of such solutions. From a higher level (in run()), we first see if a generation number is greater than or equal to plateauCheckFrom. If it is, we pass the possiblePlateauSolutions and current bestSolution to the checkForPlateau() and if a plateau is reached (isPlateauReached becomes true), we set the current break out of the main loop and stop producing any more generations, otherwise we add this solution to the ArrayList and move on.
    - The checkForPlateau function, when called upon, only does something when the possiblePlateauSolutions is not empty so for a major part of the program, we save the time it would take to unnecessarily run the loop in this function.
    - The loop compares the current bestSatisfaction with the members of possiblePlateauSolutions from last to first, and counts the number of times repetition occurs. If repetition occurs minRepititionsForPlateau times, a plateau is reached. If a lower fitness is found, it breaks out of the loop and stops looking anymore. We will make this feature more dynamic in future sprints but it works well for our simulation for now.

Parameters - why and how we chose them

These parameters have been chosen after extensive testing and represent what we believe to be the optimum balance between processing time and getting good solutions. All the final satisfactions mentioned are the mean of 3-4 test runs.

Population Size = 1000

Common: numberOfGenerations = 125, crossoverChance = 0.5, mutationChance = 0.1, cullPercentage = 0.25

Population size	Least starting fitness	Best final fitness
100	14100	15177
150	14203	15293
250	13976	15387
400	13902	15674
500	13957	15796
750	14098	15935
1000	14103	16287

Conclusion: The higher the population size, the better the final fitness because it just means that more and more solutions can be evaluated in a single generation and therefore we decided to keep it at 1000 becauses while it does increase complexity, we feel this strikes the right balance between running time and producing a good solution considering there is an average jump of over 2000 points from the starting solution to the best.

Number of generations = 200

Common: populationSize = 1000, crossoverChance = 0.5, mutationChance = 0.1, cullPercentage = 0.25

Number of Generations	Least starting fitness	Best final fitness
70	13805	15869
100	13977	16028
125	13771	16196
200	14008	16336
500	14771	16434

Conclusion: More generations generally means better final strength but at a very high cost. The difference between the solution strengths at 200 and 500 iterations is only about 100 points, which is not worth a much much higher running time since the size of population is 1000. Hence we believe 200 to be the optimum.

Crossover chance = 0.7

Common: generations = 100, populationSize = 1000, mutationChance = 0.1, cullPercentage = 0.25

Crossover Chance	Least starting fitness	Best final fitness
0.3	13796	15986
0.4	13992	15988
0.5	13996	16081
0.6	13997	16175
0.7	13813	16202
0.8	13917	16067
0.9	13995	15950

Conclusion: Increase in crossover chance improves the solution up to a certain point hence we pick 0.7 because it gives us consistently the best solutions. 0.6 will also produce very very similar results but theoretically it doesn't cost us in terms of complexity to have a higher number so even if it is slightly better, there's no harm in picking it.

Mutation chance = 0.2

Common: crossoverChance = 0.7, cullPercentage = 0.25

Mutation Chance	Number of Generations	Population size	Least starting fitness	Best final fitness
0.1	100	750	13939	15772
0.15	100	750	13941	15927
0.2	100	750	13981	16209
0.3	100	750	13746	15995
0.1	200	1000	13817	16129
0.15	200	1000	13873	16204
0.2	200	1000	14062	16526

Conclusion: This one was a little bit trickier to test and hence we compared different mutation chances at different population sizes and while the general increase in fitness can be attributed to the number of generations and population size, we notice that 0.2 gives us the best fitness at both settings and hence we go with that. This is because mutation is important to introduce diversity but very high mutations will produce too much diversity in the latter stages of the program where you want your satisfactions to get more and more consistent so they may converge towards a solution.

Starting cull percentage = 25%
While this was not supposed to be a parameter according to the requirements, we felt it was nice to allow the user more control over how the algorithm runs.
Test settings: numberOfGenerations = 200, populationSize = 1000, crossoverChance = 0.7, mutationChance = 0.2

Cull Percentage	Least starting fitness	Best final fitness
0.10	13701	16300
0.20	13786	16293
0.25	14044	16480
0.30	13722	16434
0.4	1397	16300

Conclusion: Increases up to a certain point but then gets worse. This could be because heavy culling in the initial stages of the algorithm discourages diversity but at the same time a very low cull percentage also means that some parents which are not very fit get picked. Therefore we pick what gave us the best results, a middle optimum.

OOP and Design Decisions

- When looking at our GeneticAlgorithm.java, you will find some unrelated UI variables (i.e. currSheet, bestSheet, sheets, visualizer, solverPane) being used in the run() method. This is because our GeneticAlgorithm.java is quite tightly coupled with our UI elements. This is required to finely control the animation of Sheets and the Visualizer (the line chart graph) on a thread level.
- This is because the JavaFX application runs on the main application thread. If we execute a long-running piece of code like running GA, we risk freezing the JavaFX UI application as execution of any piece of code is on the same thread.
- To counter this, on running any of our solvers (SA and GA), they are wrapped in a Runnable object. This Runnable will be executed on a separate thread. Effectively, we have a solver processing thread (that adds new data to the UI thread) and a JavaFX UI application thread (that takes the new data, and updates the graph and sheets) simultaneously.
- Also, to finely control the pausing, resuming, and stepping of the processing thread, monitors (notify() and wait()) are used.
- Other design decisions have already been explained throughout the course of the report.