Team name: KPMGComeGetUs

Name: Sng Hong Yao **Student ID:** 17205050
Name: Ankish Raj Prajapati **Student ID:** 17202456
Name: Ronan Mascarenhas **Student ID:** 17379773

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 Simulated Annealing using everything required in this sprint.
3. Due to the memory requirements and UI lags animation will cause with these SA parameters, animation has been disabled.
4. Click Play and wait for a few minutes (took us a maximum of 2 minutes to finish processing). We planned to add a non-laggy indicator in the next sprint. Once processing is done, only then will the graph show the 2 lines, and the solution sheets will be populated with the current and best solutions.
5. You cannot specify parameters for the Simulated Annealing yet. The default parameters are (why these are used are explained below):
    - Starting Temperature = 100.0
    - Cooling Rate = 0.0001
    - Minimum Temperature = 0.000000000000001
    - Max Iterations = 10000000

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)

Meeting Demands

Basic Hill-climbing

- **Prerequisite**
    - Since our Simulated Annealing (SA) was implemented before this sprint. We were acquainted with the ins and outs of SA. As asked in the first Hangout class meeting with Tony, this is just a stepping stone to implementing SA, and it is definitely nice to have it implemented and make it a subclass/interface for SA, but it is not required.
    - Now since SA uses many similar features as Basic Hill-climbing, we could just copy and adapt the functions/methods we already implemented in SA, into a new class dedicated for hill-climbing. We did not do this as it wasn't the goal of the white paper, instead we focused more on better UI.
    - To not leave you guys hanging, we will write down how we would have adapted our SA to Hill-climbing.

- **Introduce a succession of random changes to the initial random solution**
    - This is the main loop of the solver, while currentIndex is less than maxIteration. For each loop, we make a random change to the candidateSolution.
    - currentIndex is incremented by 1 each loop.

- **Keeping only those that make a solution better (i.e. reduce its energy)**
    - In each loop, if the random change made the solution better (i.e. nextEnergy < currEnergy, or nextFitness > currFitness), we greedily accept (accept straight away) the next solution as our new solution. If the randomly changed solution isn't any better, we simply ignore it and move on to the next iteration.
    - At the same time, if the new solution is better than our best solution, that new solution will be our new best solution.

Simulated Annealing

- **Introduce a succession of random changes while maintaining a global temperature variable** *(Cooling schedule (control of system temperature))*
    - Similar to hill climbing, except now we have an optional loop end condition of checking if the currentTemperature is greater than the minimumTemperature.
    - currentIndex is incremented by 1 each loop.
    - currentTemperature is decremented exponentially each loop. (we tried both linear and exponential decrease, no noticeable difference was observed).

- **Keep all beneficial changes** *(Implementation of Acceptance function (e.g., for detrimental changes))*
    - If the nextSolution is better than the currSolution (i.e. nextEnergy < currEnergy, or nextFitness > currFitness), we accept right away.
    - In our calculateAcceptanceProbability() in SimulatedAnnealing.java, if nextEnergy < currEnergy, we return an acceptanceProbability of 1.0.

- **Some detrimental changes with a probability given by an acceptance function (the Boltzmann distribution)** *(Implementation of Acceptance function (e.g., for detrimental changes))*
    - If the nextSolution is not better than the currSolution, we "consider" accepting the nextSolution. This is done via calculating the Boltzmann probability, which decides if we should accept the solution. That probability depends on the currentTemperature and the absolute difference between the solutions' energies (currEnergy and nextEnergy). Usually at higher temperatures, the probability will be higher (SA will accept higher risk solutions), and at lower temperatures, the probability will be lower (SA will take more calculated risks).
    - In our calculateAcceptanceProbability() in SimulatedAnnealing.java, if nextEnergy => currEnergy, we calculate the acceptanceProbability via e^((currEnergy - nextEnergy) / currTemperature).

Quality of OOP design (encapsulation/data-hiding, abstraction, interfaces, clean integration with components from previous sprints)

- When looking at our SimulatedAnnealing.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 SimulatedAnnealing.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 Simulated Annealing, 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.

Solution quality (suitability/viability of the generated solutions)
+
Explain your choice of parameter settings

- This depends on the parameter settings.
- But in general, the bigger the CandidateSolution (i.e. number of students), the better the solution quality if the number of loops the SA executes is large. This is because only then will Simulated Annealing have sufficient time to search through all random changes and find the best solution.
- So we naively first try to increase the total iterations SA will run on the CandidateSolution.

Decreasing minimum temperature

Starting Temperature	Cooling Rate	Minimum Temperature	Max Iterations	Total Iterations	Last "current" energy
100.0	0.001	0.01	50000	9206	12.7665
100.0	0.001	0.001	50000	11508	10.1595
100.0	0.001	0.0001	50000	13809	7.8363
100.0	0.001	0.00001	50000	16111	6.4383
100.0	0.001	0.000001	50000	18412	6.2727
100.0	0.001	0.0000001	50000	20713	5.8886
100.0	0.001	0.00000001	50000	23015	5.3482
100.0	0.001	0.000000000001	50000	32221	4.4773
100.0	0.001	0.0000000000000001	50000	39125	4.1112

Conclusion: Decreasing minimum temperature helps a lot.

Increasing starting temperature

Starting Temperature	Cooling Rate	Minimum Temperature	Max Iterations	Total Iterations	Last "current" energy
100.0	0.001	0.01	50000	9206	12.7665
500.0	0.001	0.01	50000	10815	12.1521
1000.0	0.001	0.01	50000	11508	12.8916
5000.0	0.001	0.01	50000	13116	12.8518
10000.0	0.001	0.01	50000	13809	12.8712
20000.0	0.001	0.01	50000	14502	12.9836

Conclusion: Increasing starting temperature may help, but only to some extent. This may also be due to our cooling function that uses exponential decrease, meaning at larger initial temperatures, the temperature will within a short span of time, decrease to zero.

Decreasing cooling rate

Starting Temperature	Cooling Rate	Minimum Temperature	Max Iterations	Total Iterations	Last "current" energy
100.0	0.001	0.01	10000000	9206	12.7665
100.0	0.0001	0.01	10000000	25376	7.9605
100.0	0.00001	0.01	10000000	921030	7.1342
100.0	0.000001	0.01	10000000	9210336	7.2218

Conclusion: Decreasing cooling rate helps to some extent, but at the expense of time.

Best of both worlds: Decreasing cooling rate and decreasing minimum temperature

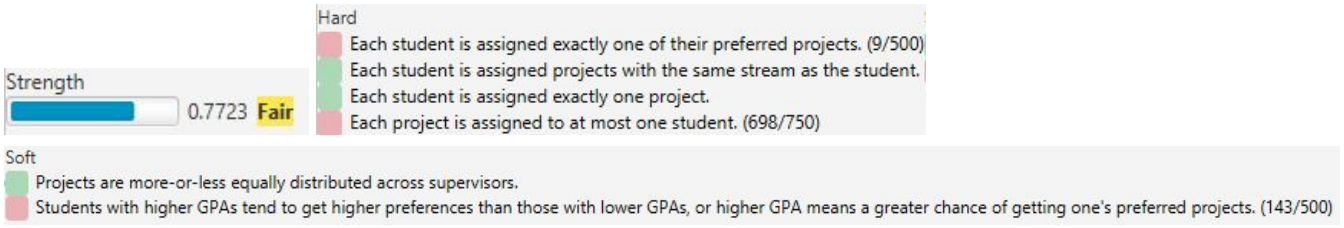
Starting Temperature	Cooling Rate	Minimum Temperature	Max Iterations	Total Iterations	Last "current" energy
100.0	0.0001	0.000000000000001	10000000	320610	2.8285
100.0	0.00001	0.000000000000001	10000000	3914376	2.6670

100.0	0.0001	0.00000000 0000000001	10000000	460494	2.8450
100.0	0.0001	0.00000000 00000000001	10000000	529569	2.8119
100.0	0.0001	0.00000000 000000000001	10000000	598643	2.7699

Conclusion:

- For reference, the random candidateSolution generator we created from the previous sprint achieves an energy of 2.5022. So the above configurations are to the team, considered quite good.
- The first row is therefore used as our parameters for this submission because it is good enough and fast enough.

Energy might seem good, but the solution is still not good enough



- So we created a solution "Strength" indicator (required in future sprints). The method to calculate this strength is different from the one used in satisfaction calculation. For each hard and soft constraint, we allocate a maximum point. The points mean the effect each constraint has on the strength.
- Soft constraints (we have 2) are given 0.05 out of 1.0 each. Hard constraints (we have 4) are given the rest of the points (i.e. (1.0 - (0.05 * 2)) / 4) = 0.225.
- With this point, we can calculate the strength of a solution.
- i.e. Based on the images above,
    - If a constraint is violated, their number of violations and their maximum possible violations are listed at the end of the sentences, i.e. (numViolationsOfConstraint/maxViolationsOfConstraint)
    - Violation cost = 9/500 * 0.225 + 698/750 * 0.225 + 143/500 * 0.05 = 0.22775
    - Strength = 1.0 - 0.22775 = 0.77225
- To relate things together, we achieved a strength of 0.7723 for our solution with energy of 2.8285. This means the solution SA produced is still not good enough, but this is as close and as fast as our PCs allow.