Group Members

Group Name: Staff_Solution
Karthik Krishnan: 3034843072
Vinayaka Srinivas: 3034803890
Tarang Srivastava: 3034634981

Generating Inputs

We started with a desired output that we wanted to create an input for. That is, how many breakout rooms and how many people in each breakout room. We generated a script that used a Gaussian distributed points in a certain happiness and stress range. We had two Gaussian distributions for happiness and stress based on if the edge was in our desired output. For edges in our output, we generated happiness using a higher mean Distribution with higher values and stress in a range with lower values. We did the opposite for stress, with lower happiness and higher stress. This would allow our input to match our desired output. We then picked a stress budget that was as close as possible with our desired output.

$$\text{Undesired} = \mathcal{N}(3, 0.65) \qquad \text{Desired} = \mathcal{N}(0.25, 0.125)$$

If we had more time, we would likely try to take a more random approach. Instead of picking a safe difference in the happiness to stress ratio for our unused edges in our output, we would try to bring it closer to our desired edges. This would test algorithms more and make it more difficult to determine a solution. We would also try to pick tricky rooms with edge cases such as 0 stress to trip up algorithms that are entirely happiness-based. This would be a more rigorous input that could have been created if we had more time.

Approach 1: Greedy Local Search

We initially implemented a local search algorithm that greedily chose vertices to add to a breakout room based on highest happiness while monitoring the stress budget. The only moves to change neighbors that were valid were additions of one vertex to the room.

$$\text{happiness} = \begin{cases} -100 & \text{if not a valid solution} \\ \text{calculate\_happiness} & \text{else} \end{cases}$$

Upon testing this algorithm on our data, it performed really well on approximately one-third of the data while performing poorly on the remaining two-thirds. In order to incorporate more flexibility with the moves being made to access different neighbors, instead of freeing one student and adding them to another room, we would free two vertices and then re-add them optimally. We also incorporated swaps to get closer to optimality when a vertex addition was too drastic of a move to make. Although this partially improved the results on some of our data, it ended up performing worse on other inputs. From this, we realized that a degree of freedom of two students was not sufficient to consider the broad possibility of solutions and would get us stuck in a particular configuration that is not optimal.

Approach 2: Simulated Annealing

In order to give our algorithm freedom to consider more various options, we decided to incorporate probabilistic decision-making through the form of simulated annealing. With the difference in happiness between a move being considered and the current position as the primary factor in the decision-making for suboptimal moves, we allowed for student additions and student swaps. Where an addition, is moving a student from its current breakout room to a new one and a swap is swapping two students' respective rooms. That is let s' be a swap or add in the neighborhood of our current assignment, then

$$cost(s) = \text{current happiness} \qquad cost(s') = \text{happiness after } s$$

We also incorporated removing a vertices from a room and combining two rooms of vertices. Initially, we implemented the algorithm on the configuration already determined by local search, but it offered little improvement. Therefore, we implemented the algorithm with the default being everyone in their own room. This produced better outputs for some of the medium-sized inputs and large-sized inputs, while failing miserably on smaller inputs. We also tried an approach where we started off by relaxing the stress budget and overtime decaying it to the true stress budget. For a few inputs this approach provided useful results, but was not a silver-bullet solution. We believe that optimizing a combination of our greedy outputs on the small inputs and our probabilistic algorithm (annealing) on the medium and large outputs led to a decent performance across the board.

Computational Resources

To run our algorithm, we used all our computers by distributing the data to each teammate evenly. From there, we individually split the data into thirds or quarters again, and processed them in parallel on each device to take advantage of our multi-core processors. Since, naive Python code runs on a single thread, this was our simplest solution of taking advantage of parallel processing.

Further Improvements

Due to less time at the end, we would try to guess and check our ratios of adds, swaps, combines, and removes in our algorithm. However, given more time, we would have looked at our data and tried to understand exactly which move is occurring too much or too little to create a sub-optimal answer and adjust our distribution of moves accordingly. Furthermore, our annealing ratio to allow for moves that decrease edges was relatively untested. As a result, more time would have allowed us to look at the data more closely and adjust our annealing case such that our algorithm makes an ideal amount of moves that decrease happiness. Lastly, we would have tried to make our moves themselves more dynamic. Instead of adding and removing one vertex and swapping a pair of vertices, we would have implemented a randomization of how many vertices or rooms change such that our moves are more dynamic and allow for more freedom to explore other possible moves. Because adding a vertex in the beginning can introduce some order bias towards high stress edges, allowing for moves with more vertices would correct for such a bias.