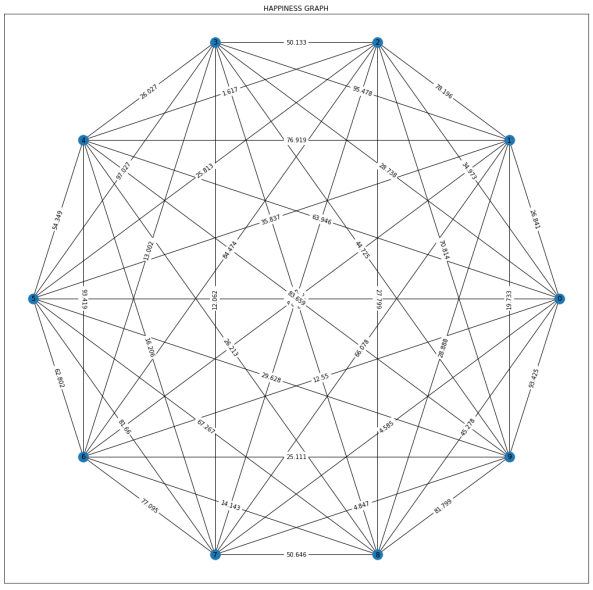
# Project Reflection

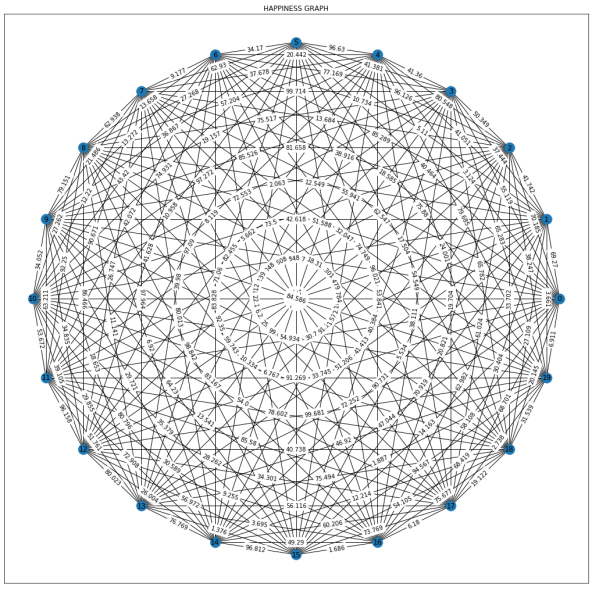
*Disclaimer: Our groups' Phase 1 submission was done by three people: Raisa Bunyamin, Trang Van, and Yuner (Evelyn) Lu. This reflection and Phase 2 is done by Raisa and Trang. Yuner Lu is submitting Phase 2 separately (was approved by Nate Young).*

(Code: Github)

## Phase 1 - Inputs

We started by creating a random unweighted, directed graph, and then we picked a small set of nodes at random. This had to be a complete graph (ie. each node is connected to every other node) because each student causes every other student some amount of stress or brings them a certain amount of happiness. (See the figure below) Each edge would contain the weights: `{happiness, stress}`. Then, we were able to force it to be an independent set by removing all the edges between our set of edges, and we know this is still a hard input for the independent set problem. We referenced the `networkx` documentation to complete our input graph function, and as well as the solver and other helper functions.



## Phase 2 - Algorithm & Approach

In Phase 1, when we were generating the inputs and outputs, we found one room that maximized the happiness, but kept under the stress budget. Since it's possible for a breakout room to have 1 student, we assignment the remaining students to be in their own breakout room.

In the solver phase, we wanted to continue this idea. However, we didn't want to just assign the remaining students to separate rooms. So, we decided to implement a greedy algorithm. The general process was to decide a number of rooms, $k$, and start filling each room with students until there are no more students. To do this, we randomly select a node (student), $x_0$, from the graph. Then, we match the node with another node, $x_m$, that generates the maximum happiness. If the room is currently empty and the two students' stress levels don't surpass the maximum stress threshold, we add them to the room. Then, we disconnect both these nodes from our graph. In the next iteration, we again choose a random node, $x_1$, from the updated graph and find it's best match, $x_m$, (a node that would bring the most happiness). There are several cases to check for. The first case is if adding $x_1$ and $x_m$ will exceed the current room's budget. As a small detail, we also have to check the stress between the students already in the room and the node we want to add. If the stress of adding both nodes exceeds the budget or if adding just $x_1$ exceeds it, we add the pair to the next available empty room. If adding both doesn't exceed the budget, we add the pair into the current room. Otherwise, if adding either would be under the current room's budget, we add the node whose happiness is higher. This process would continue until there are one or two nodes left. Once there is, we can add the last pair to an available empty room.

We felt this might be a good approach because the nature of the problem sounds like it could be reduced to a greedy matching algorithm, but with the stress levels accounted for. We thought that randomly selecting nodes would better than starting at node 0 because it adds some variability when we generate outputs.

Before settling for this approach, we thought of reducing it to other solvers such as the Traveling Salesman Problem. However, the problem is more focused on grouping nodes, not what path would optimize happiness. After conversing with a TA and looking at some other similar problems, we also thought of relating it somehow to Rudrata's, and realized that this would also end up reducing to the Traveling Salesman Problem (or that they are closely related). Before eventually deciding to implement our

greedy algorithm, we had also researched about stable marriage and its derivative stable neighbors.

## Resources & Errors

We didn't get the chance to run any computational resources. Ideally, we would move our files and inputs on the Hive instructional machines to run `solver.py` on the folder of inputs. However, we ended up just generating output files on Jupyter Notebook, where we did a lot of experimenting.

Given more time, we would definitely try to add a heuristic to our greedy algorithm that is able to predict a more precise number of total breakout rooms. We would also try to work with translating our graph onto a matrix, and then we can attempt the Traveling Salesman Problem with the stress level as the restriction. We can determine a set of elements from this matrix (ideally a square matrix), one in each row and one in each column that maximizes the total happiness, and repeat that algorithm until the stress cap has been reached. We also thought about the Max Flow problem and its capacity. We thought our graph had a similar frame work with `happiness` being the units of flow and `stress` being the capacity. However, the actual problem wasn't exactly what we wanted. Another approach was the Minimum k-cut problem or other graph partitioning algorithms. This type of problem was most similar to what we were trying to do - partition our graph/ group our students. For this problem, we found this resource very helpful and would've liked to attempt this.

Our approach was no where near perfect. The way we tried to handle a room's stress did not translate the same into Python code. We felt that even though we had a good idea of what to do, it was difficult to translate it and we ran into various errors. We would've liked to explore more of what `networkx` has. For instance, there were multiple functions available to perform the algorithms we learned about in class, like Edmond-Karp algorithm.