170 Project Final Report (Group 11)

We first implemented the strategies outlined in the initial report, including a naive baseline that simply drops all students off at “Soda” - starting car location - which works surprisingly well on some inputs as we visualized some of them selectively. Then, we decided to go a little deeper, which involves a more advanced baseline that solves a TSP based on all homes. This is done by viewing only a subgraph of the original graph consisting of only the homes and starting location, and then use the all pairs Dijkstra’s result to fake an edge between any pair of homes even if there isn’t one. We then continue working on this path by randomly selecting a subset of homes to run our fakeTSP on. The random selection is done simply through random.random() and setting a seed initially. The TSP solver we use is developed by Google.

Furthermore, due to the large portion of grades allotted to solving our own inputs, we put extra hyper parameters and tried different seeds to find the best outputs we can find on our own inputs, and then insert a special check to return our own “optimal” solution directly if the input matches with our input’s pattern.

Afterwards, we implemented k-clustering for homes and then picking the center of each cluster to run this fakeTSP. The code follows from the idea in textbook, and after running it for a while, we discovered that this is more or less similar to randomly selecting a subset of all homes and fine-tuning the selectivity.

Being stuck on a local minimum, we decided to use some method to further improve our results. The one thing we found was simulated annealing. It was difficult to define the transition probabilities, so we decided to just randomly select pairs and see if it improves anything. This doesn’t help too much, so we implemented general TSP from an online paper. We always compare results with previous ones and change them only if we've got an improvement.

We used Google Cloud Platform for about an hour or so, and discovered that it didn't help with our efficiency too much. The main problem was that our Python kept giving a segmentation fault, so we wrote a script to basically continue from where it left off from the original inputs. Then, we could finish the inputs rather quickly. Lastly, we visualize inputs that always gives us a segfault, visualize it online, and basically do a special manual solve on each of them - there's about 10-20 of them, mostly consisting of fully-connected graphs or star-like graphs, and both of which are very easy to figure out the optimum. These are also specially detected/conditioned to directly return the optimum we've found. In cases where it's just fully-connected or star-like graphs, we basically ran the optimum solver for these two specific types of graphs on all inputs and compare with previous solutions to see which one's better.

Based on the current k-clustering algorithm, we added another layer which finds the number of times any location appears in all pairs shortest path among the locations currently chosen to be dropped off. We collect the locations that appear over the threshold-many times where the threshold is arbitrarily determined due to time constraints. We then run the fakeTSP on the locations collected to optimize the behavior of k-clustering. This works well in some outputs, but the bulk of the work is still done by k-clustering, which also gives better results for most inputs.

These are all the methods we've tried. Since we're essentially comparing multiple methods within a large solver.py file, it's not completely clear which algorithm gives the best outputs overall, but solely from the result of seeing which algorithm makes the last modification before termination - a print() statement indicates that it is the best algorithm currently - we found that k-clustering works best in most cases, especially with the modification of doing an additional round of random selection within each cluster again after the clustering is done. Almost 90% of the graphs, especially the non star-like graphs, find k-clustering to work best on it. We believe k-clustering works well because the algorithm essentially is just finding clusters of TA homes and drops that cluster off together. The fine-tuning of the additional layer of randomly choosing some more homes within the same cluster gives us the flexibility to drop off the TA's not just at one point, but potentially at several locations, even within the same cluster, and would be a boost to outputs' quality since the graph could potentially be similar to a "recursive-star" structure. That is, it has a star-like structure for each branch of the main star at the bottom level.

If we have more time, we'd definitely like to refine our algorithm by running simulated annealing on the current-best outputs. For this project, it has not been plausible since we just came up with the idea of annealing, and by the time we finished implementing and debugging, the time left is not enough to run this additional step on all inputs and outputs anymore. Moreover, we could test and fine-tune the hyper parameters of the randomizations we've done. For example, we would test how changing the threshold for different graph sizes would modify the behavior of the outputs.