Overview

The idea of solving this problem came from the MST algorithm for solving the Metric TSP problem. Despite the similar nature between these two problems, there are some nuances worth noticing.

The first being the distinction between locations and homes: the metric TSP problem will have to visit every node in the graph, but our problem only need to visit home nodes. In addition, in order to save total cost, the algorithm should visit few non-home nodes if possible. Our solution to this is to modify the original MST algorithm to something that satisfy our specific needs. Our MST algorithm is built upon a modified version of Prims MST algorithm. We call it <prims.py>.

After we got back a desired MST from our <prims.py>, we started to optimize the MST to incorporate drop-off locations and further prune the MST. The car won’t need to go to every home, because it can drop a TA along the way. Due to cost of driving and walking, we decided to prune all non-source leaves from the MST. Our MST algorithm guarantees all leaves are home or the source, so the car doesn’t have to drive to those leaf homes and drive back. We keep pruning the leaf node until the leaf becomes a branch or another home or a determined drop-off location. This process of pruning and finding drop-off locations is done in the drop-off method in <solver.py>.

When generate the final path using depth first search, we realized that different from a TSP problem, we cannot assume our graph is a compete graph. So after we recorded the DFS traversing order, we inserted shortest path between non-adjacent nodes in the ordering, and run shortest path to get back to the source after all homes are traversed.

<prims.py>

This is our modified prims MST algorithm. Instead of adding the shortest edge incident from the connected part each time, we run dijsktras to find the closest home from the connected part and add that path to the connected part. The original distance to fringe is modified to distance to homes fringe. Edge to dictionary replaced by a list of nodes representing the shortest path from the connect part to the home.

solver.py

Four things are accomplished in solver.py: …

1. Construct graph objects based on inputs in inputs/ directory.
2. Given reduced MSTs processed by functions in prims.py, returns dictionaries with key value pairs indicating drop off locations and the TAs who're dropped off.
3. Give MSTs and drop-off locations, returns optimal paths for Professor Rao to drive his vehicle that saves the most energy.
4. Put everything together and solves the problem, outputting lists of locations representing the car path and dictionaries mapping drop-off location to lists of homes of TAs that got off at that particular location.

…through these five functions: recover\_graph(), drop\_off(), pre\_order() + parse\_path(), as well as solve(). recover\_graph() was completed rather smoothly, generating no bugs along the way. However, drop\_off() was quite troubling. We encountered two major bugs where the starting vertex and non-homes drop-off locations are being pruned off when they are not supposed to. Then, in order to generate the optimal car path, we exploited the properties of pre-order tree traversal and called Dijkstra’s Algorithm to fill in the gap when incident elements in returned list not neighbors of each other—this is essentially what pre\_order() + parse\_path() do. When we have all the small pieces built and tested, it is rather straightforward to code up solve() to put everything together.

graph.py

We decided in Phase I to implement our own graph Class to the end of having more flexibility in Phase II, and indeed it has been beneficial. We were able to add attribute that indicate whether a vertex is a home, drop-off location, or starting vertex when other functions needed such attributes. Also, the deleteLeaf() method for graph Class played an important structural role, as it recursively prunes all the leaves until reaches a non-leaf or a TA’s home. Surely no open source python libraries, such as Igraph, has such case-specific functionalities. However, with this graph Class has its own drawbacks—it is not generic at all. If (Energy to walk / unit) / (Energy to drive / unit) is no longer 1.5, but a double larger than or equal to 2, this Class breaks, and is extremely difficult to amend. Luckily, it is doing alright in this project so far.