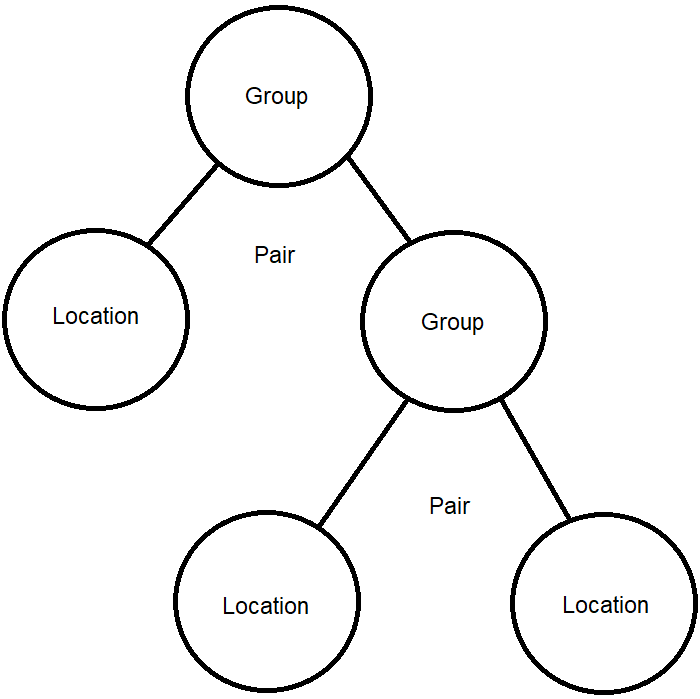
C950 Task

Wesley Lancaster

B1: Logic Comments

When I approached how to create an algorithm to handle the Traveling Salesman Problem, I noticed that the shortest path often contains the smallest edges. If you define your path from smallest edges to largest, however, the result is not always ideal. I decided to use a greedy approach that consecutively changes its requirements as it goes. It groups each location with its closest neighbor first, then it groups those groups if they are “compatible”.

I also realized that grouping many locations together was a waste of time, as you’d still have to find a good path through all its locations. Instead, I limited groups to only contain a pair of objects (Locations objects and Group objects), so the result is tree-like, with locations as leaf nodes. This makes it easier to define a path, because the tree stores close locations together, and far locations further away.

Here is a breakdown of this process, called group\_locs:

1. Create a dictionary that maps edge length to its vertices.
2. Sort all edges from shortest to longest.
3. Pair the vertices belonging to each edge, until all vertices belong to a group. Only group them if at least one vertex doesn’t already belong to a group.
4. Now that all vertices are grouped, combine “compatible” groups until they approach the maximum number of packages that a truck can carry

This procedure is done at the beginning of the day and any time package updates occur, such as when packages arrive at the warehouse. It is first called on to combine our “cluster” into a group of its own, or group of packages that require being delivered together. After this, it only considers locations whose packages are completely ready for delivery, rather than having to revisit locations when one of their packages is available later and waste miles.

Group objects take in all constraints that the packages in each of their locations have. For multiple constraints, the group takes the worst one, such as the earliest delivery time. Since we don’t want a group to have more than one truck requirement or cover too many miles, we make sure to not combine groups if it makes the parent group unmanageable for a truck to handle. This is what is meant by “compatibility” – a way to make sure groups will be feasible for delivery, and not have contradicting constraints. This makes it so that we never have to iterate over packages to compare these properties too much, since all groups contain all the worst constraints of their packages.

Once groups are chosen, the trucks will choose a group that they are able to carry, and that is the most critical. Then, the Group object realigns its locations to make a shorter path than the default. This is done recursively, by telling the group where we’re coming from and seeing which of the pair it contains is closer. It swaps their order if necessary, then continues down the line if one of the pair is a Group. The truck loads all the packages from the group’s locations, and drives to each package’s location and delivers them. It then returns to the warehouse to re-load or finish for the day.

B2: Application of Programming Models

The project was coded in Python, using PyCharm 2020.2.3. All classes were my creation, and can be found in the WGUPS\_Objects.py file.

B3: Space-Time and Big-O

Because the algorithm is location-centric, its running time greatly depends on the ratio of packages to locations. I decided to focus on locations because it’s a good way to shortcut processing time if there are many packages to each one location, which would be realistic in many instances. Given that n is the number of packages we have to deliver, if there is only one package to a location, the running time of group\_locs is O(n^3), because each location is essentially a package. If we have a scenario where there are 2 packages in each location, it halves n, which means the overall runtime will be an eighth.

B4: Adaptability

I handled project-required values by creating universal variables, like the number of trucks, maximum number of packages per truck, and start time. This way, no numbers are hard-coded into the project. The project can handle any number of packages, any number of trucks, with any package limit, and any start time. I planned on making a menu option to change these variables during runtime, but decided it was outside the scope of this project.

There is one problem with adaptability, I took the option given to us in the welcome video to hard-code all the package and location adjacency data. However, this can easily be exchanged for reading this information from file on a loop. All the methods required to create this data already exist and are used, so this implementation would be simple.

B5: Software Efficiency and Maintainability

The efficiency for this program is acceptable, and has been discussed in detail. I consider it to be very maintainable, as my classes and methods seem fairly straightforward, and are well-documented by comments.

B6: Self-Adjusting Data Structures

The primary self-adjusting data structure is the hash table used for all packages. It is able to add any number of packages, remove them, and look them up. Additionally, Group objects are in a state of constant change while they are being created and combined. They are the spokesman for all the locations and packages they represent, so if there’s a package with any constraint being added to the group, it takes that constraint upon itself so we can save processing time later.