Core Algorithm Overview

Scott Aylward
2021-01-20
C950 – Task NHP2

Preface – Problem Statement

Determine and implement an algorithm for delivering a set of packages for Western Governors University Parcel Service. Packages must be delivered on-time and within individual package requirements. The algorithm used should find delivery routes for these packages that does not exceed 140 Miles, given three trucks and two drivers to complete the deliveries.

A: ALGORITHM SELECTION

After review of the TSP and various approaches, I elected to use a greedy 'Nearest Neighbor' algorithm to choose the next package location based on proximity to the current location of the truck. This method allows substantial room for future optimization. In a better case, the entire package manifest would be analyzed before loading, and the trucks would be loaded with an ordered list of packages representing the optimal delivery route.

For this project, I chose the more efficient algorithmic approach rather than use a heavier algorithm for a more optimal route. Because the packages are delivered within requirements and the total route distance is under the limit, no further route optimization was necessary.

B1: LOGIC COMMENTS

The nearest neighbor algorithm I implemented has very straightforward logic. It executes these steps:

1. Given a current location(trucks always start at the hub) and a package list
2. For each remaining package
    a. Get the destination address
    b. Check the distance table for the distance between current and destination address
    c. Check whether distance for package n is lower than for other packages
3. The smallest distance wins, and the associated package is delivered

This algorithm can be found as the choose_next() function in main.py.

B2: DEVELOPMENT ENVIRONMENT

The program was developed on Python 3.8, using PyCharm CE 2020.3.2. Development machine is Windows 10, with venv. Original data provided in Excel spreadsheets was cleaned up and exported to csv. The three csv data files can be found in the 'data' directory. The standard lib csv methods are used to read the data files into the appropriate data structures. The program uses only the python standard library.

B3: SPACE-TIME AND BIG-O

Each function or significant code block is commented with space-time complexity in Big-O notation. The overall time complexity for the program is O(n^2), which is the time complexity of the core routing algorithm(O(n) run_route() * O(n) choose_next()). Below is a breakdown of each relevant section of code:

hashtable.py

Function/Method	**Space Complexity**	**Time Complexity**
__init__	O(n)	O(n)
bucket_hash	O(1)	O(1)
insert	O(1)	O(1)
lookup	O(log n)	O(log n)
remove	O(log n)	O(log n)
table_size	O(n)	O(n)

package.py

Function/Method	**Space Complexity**	**Time Complexity**
__init__	O(1)	O(1)
print_long	O(1)	O(1)
print_inline	O(1)	O(1)

utility.py

Function/Method	**Space Complexity**	**Time Complexity**
sort_packages	O(1)	O(1)
time_convert	O(1)	O(1)

importdata.py

Function/Method	**Space Complexity**	**Time Complexity**
read_packages	O(n)	O(n)
read_distances	O(n)	O(n)
read_addresses	O(n)	O(n)

main.py

Function/Method	Space Complexity	Time Complexity
choose_next	O(n)	O(n)
run_route	O(n)	O(n^2)
run_sim	O(n)	O(n^2)
__main__	O(n)	O(n^2)

B4: ADAPTABILITY

The simple nearest neighbor algorithm scales well within the known parameters because it operates at the level of each truck. It would produce a less optimal result and take longer to run(O(n^2)) if trucks had increased capacity. If the total number of packages increased, additional complexity would be needed in the form of automated sorting.

If the assumption of the number of packages holds true, sorting could be done effectively with a greedy algorithm. Using the deadline and notes field, a greedy algorithm would load packages on to trucks until full. If the number of packages increased substantially, and/or the delivery area increased, the sorting algorithm would need to take addresses and routing into consideration. At that point, I'd opt to combine the sorting and loading(creation of truck manifest) steps into a single, more complex algorithm. The route plan would be known at load time, so the trucks could just proceed in a specific order of packages.

B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

The overall time efficiency of O(n^2) is appropriate for the complexity of the program. The naive, greedy sorting algorithm discussed in the above section would have linear time complexity, so the whole program would still be O(n^2) if sorting was automated. Finding a more optimal solution by considering distance at the time of sorting is likely also possible without reducing efficiency.

The program uses a small number(5) of modules, with only 2 classes. The functions and methods are commented and have clear variable names. Variables that are passed or returned have more descriptive names. Functions have been organized in a way that permits easy rework of individual components.

B6: SELF-ADJUSTING DATA STRUCTURES

I elected to use a chaining hash table for this program. I preferred it mainly for its simplicity in collision management. This data structure is easy to implement from scratch and works well for a dataset of the size this project required. It is important to keep in mind that the lookup and remove methods must iterate through a bucket. The number of buckets may need to maintain acceptable response time from those calls, so that each buck still has a relatively small number of items.

I considered using a linear probing hash table for this program. It has an advantage over chaining in that a search only needs to look at the keys, and can be as efficient as O(1). Ultimately, the added complexity of lookups and deletes didn't seem worth a possible, slight improvement in efficiency.

C: ORIGINAL CODE

WGUPS is an original program developed by Scott Aylward to meet the provided requirements.

- All packages delivered on time
- Total mileage is under 140 miles
- Descriptive comments can be found throughout the code
- Identifying information can be found in main.py, lines 1-2
- The program presents a basic CLI for querying package information at a given time, viewing route mileage, and querying a specific package

---

D,E,F: DATA STRUCTURE

The package objects are stored using a common chaining hash table implementation. The hash table is initialized as a list of lists, with the inner lists functioning as chained hash buckets. The hashing function is of the simplest type, using only the package ID as a key and key % 10 for the bucket hash. The package objects are then stored in the buckets. The hash table is self-adjusting through chaining, and can be manually adjusted via the bucket count and hash modulus.

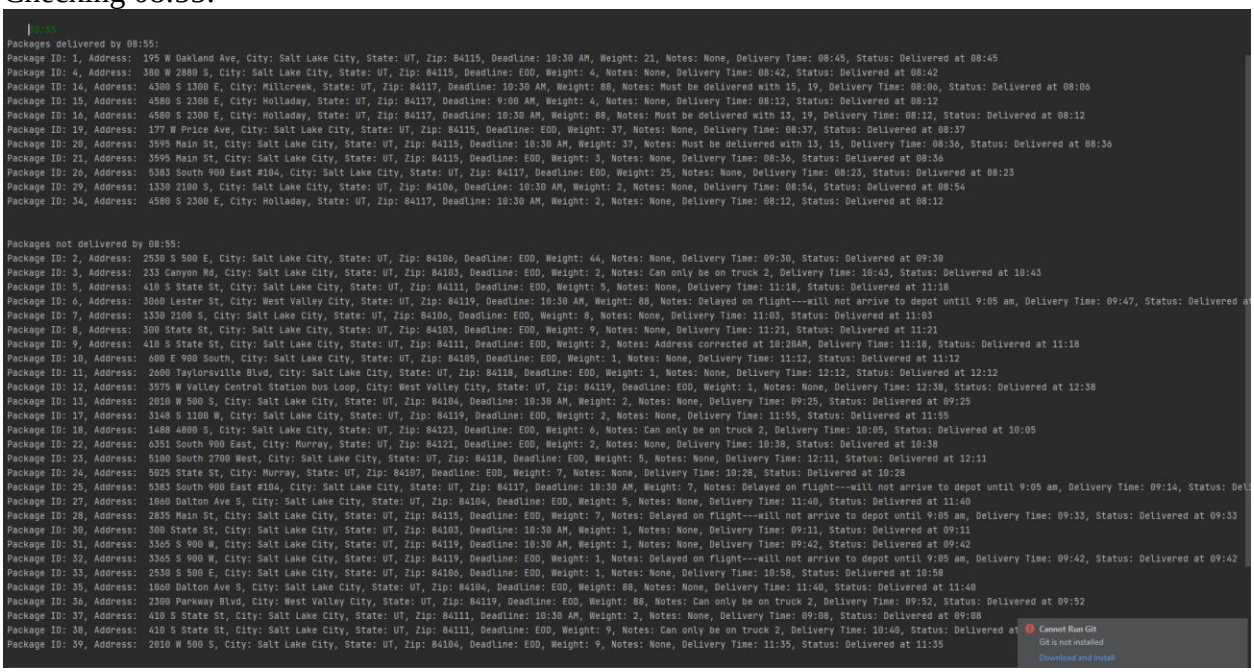
The insert and hash methods run in O(1) time. Lookup and remove only look through objects in one bucket, making them O(log n). Table_size must iterate through each bucket in order to get a total count, making it O(n).

See hashtable.py

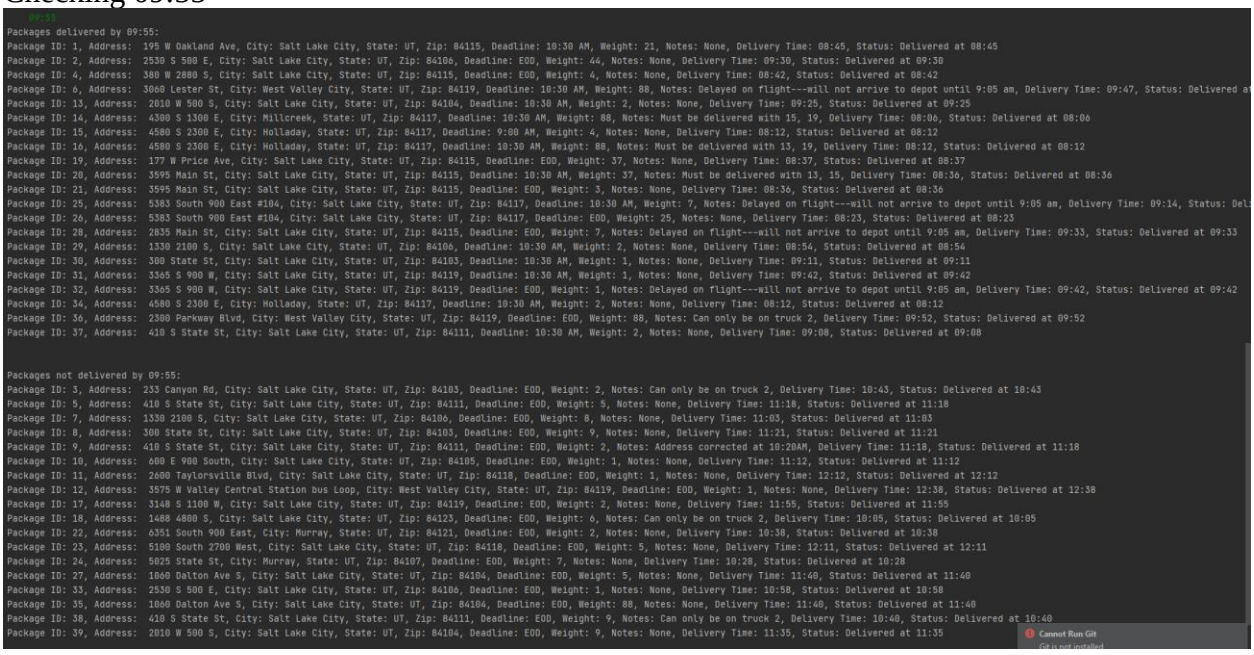
G: INTERFACE

Screenshot images also included with project files, as the text is hard to read here.

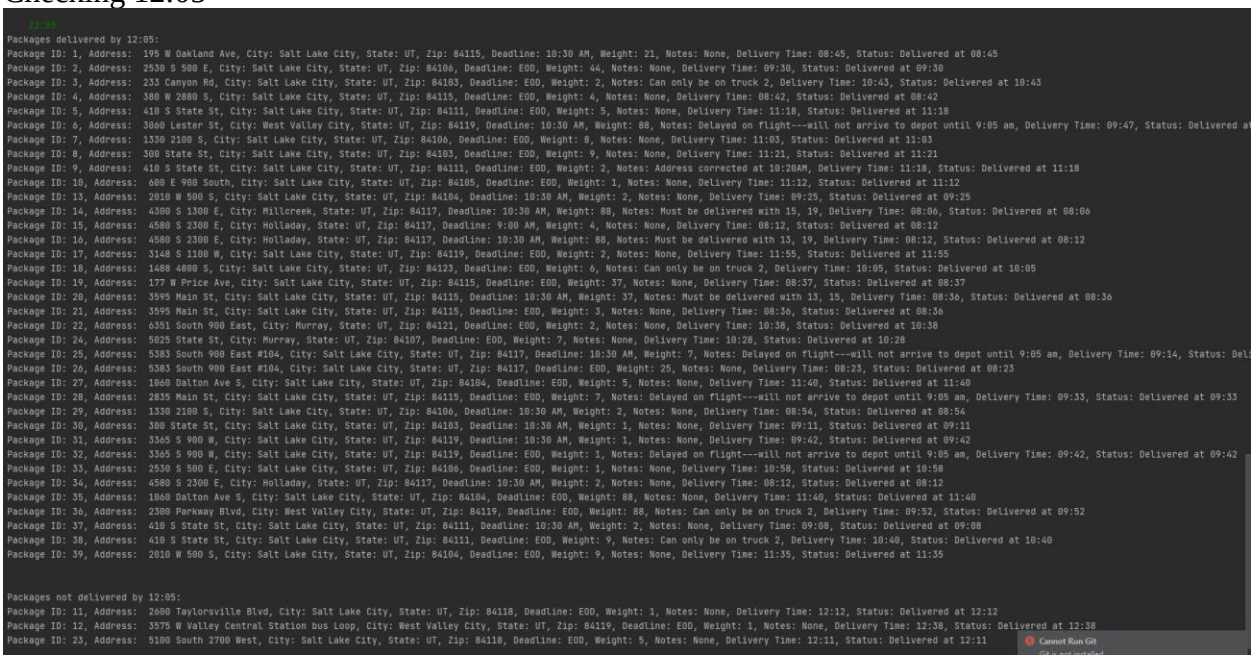
Checking 08:55:



Checking 09:55

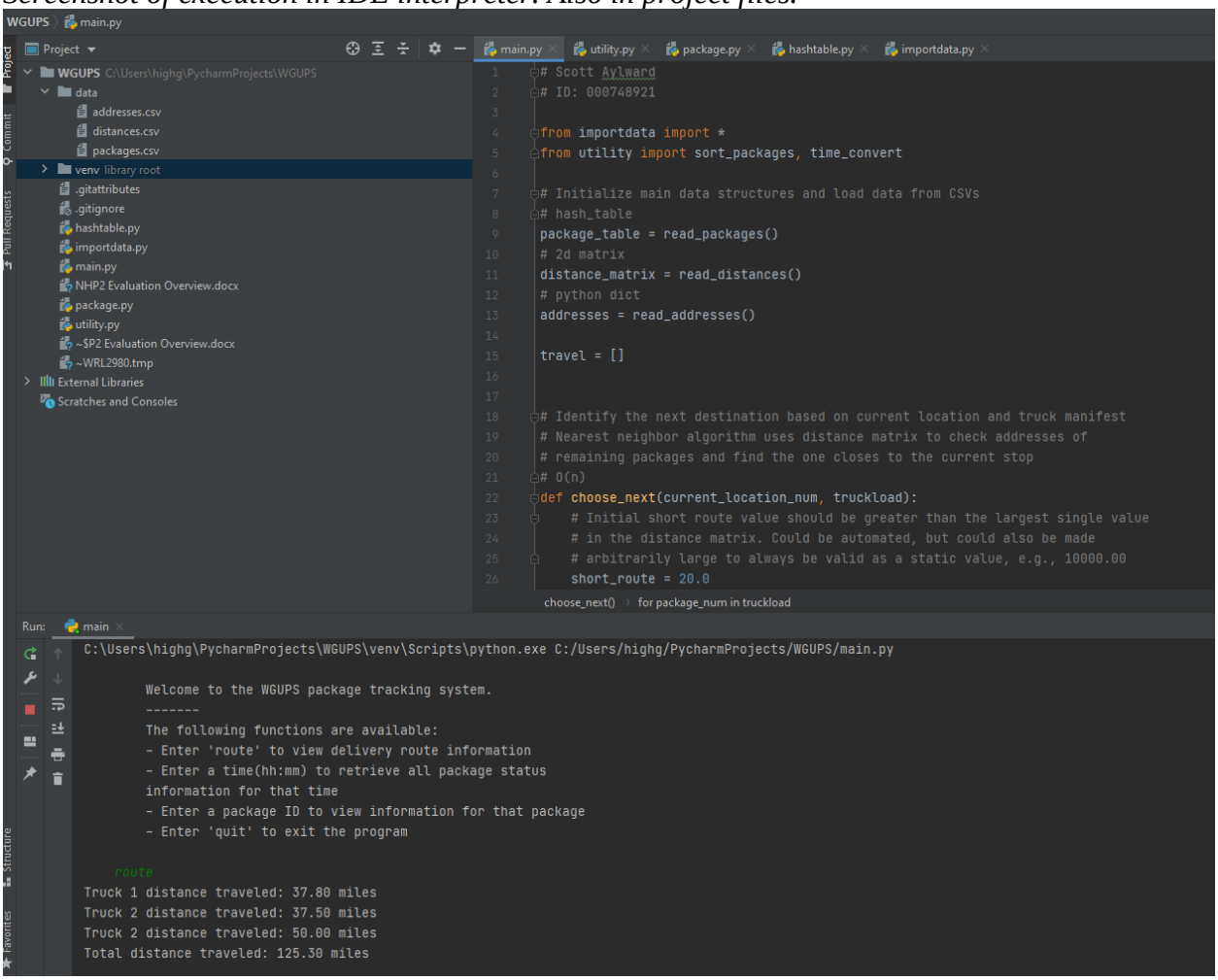


Checking 12:05



H: SCREENSHOTS OF CODE EXECUTION

Screenshot of execution in IDE interpreter. Also in project files.



I1: STRENGTHS OF THE CHOSEN ALGORITHM

A greedy Nearest Neighbor algorithm is a good choice for this problem for two reasons. Firstly, it provides a sufficiently efficient solution that addresses all requirements. An algorithm that determines a more efficient solution is likely to have greater space complexity, and possibly greater time complexity. There's no stated benefit to a more optimal route, therefore no reason to incur an increase in complexity.

Second, this problem is undirected, with possible edges between all points. A distance table was provided with the package manifest, allowing for efficient lookups and comparisons of

distance between delivery addresses. Like the TSP, this problem provides a very literal and easy to conceptualize application of an NN algorithm.

I2: VERIFICATION OF ALGORITHM

See CLI output for verification of truck/overall mileage and package delivery within requirements.

I3: OTHER POSSIBLE ALGORITHMS

Other good approaches to this problem might use a Branch and Bound algorithm or a Dijkstra's Algorithm. Branch and bound method can find the minimum possible cost of following each node, and select the node with the best solution. While this method is capable of finding a more efficient solution to the problem, its worst case time complexity is the same as a brute-force approach(O(n!)).

Dijkstra's algorithm could be used if modified slightly to produce a round trip. For example, one could use Dijkstra's algorithm to find the shortest distance between the hub(start) and a delivery location near the hub(end). A following step, as with the algorithm I used, would have the truck return to the hub from the final location. This method could be implemented with a time complexity of O(n^2)(with n == V in this problem), or made more efficient through the use of a priority queue structure.

J: DIFFERENT APPROACH

I will likely revisit this problem in the future when I have a better command of Python. I may wish to restructure the modules and classes, and I'm sure my code could be more 'pythonic'. I would implement a more complete and interactive CLI with argparse. It would be a fun exercise to allow a user to control the progress of the deliveries, manually moving forward/backward one step(package) at a time.

I would also want to write an algorithm for truck loading. My first attempt would be a greedy algorithm that starts with notes('must be on truck n', 'must be delivered with package i') to group packages with common requirements. It would then use the delivery deadline to load packages on trucks based on planned departure time. Finally, it would add remaining packages to empty slots on trucks with packages going to the same Zip code.

K1: VERIFICATION OF DATA STRUCTURE

Compliance with requirements is demonstrated in above sections D-H

K1A: EFFICIENCY

The hash table implementation achieves an efficiency of O(1) on inserts and O(log n) on lookups. As mentioned briefly above, I would scale the number hash buckets proportionately to the number of packages. That way the lookup retains the same execution time, as it only needs to iterate over the items in one bucket.

K1B: OVERHEAD

The hash table implementation has a consistent space complexity of O(n), regardless of scale. The initialized size of the empty list pointers is negligible at any scale. As items are added to the hash table, space usage grows linearly.

K1C: IMPLICATIONS

My usage of the hash table to store package objects means there's no impact to the data structure as the number of trucks or locations grows. My initial approach used multiple hash tables. One was used to store all packages pre-sorting, after which the packages would move to a truck hash table. As each package was delivered, it was removed from the truck table and inserted into the original table. I later moved to storing only the package IDs with each truck, and update each package object at the appropriate time. Even with a larger scale of trucks, or hubs in multiple cities, I would continue to rely on a single data structure for all packages. Each hub might use a local hash table with packages it is responsible for, but only to preserve low latency to the machine running the program. The tables from each hub could be merged back to the primary source of truth on a regular basis.

K2: OTHER DATA STRUCTURES

Other data structures that could be applicable for this project are graphs and binary search trees.

K2A: DATA STRUCTURES DIFFERENCES

A graph could be useful for holding the package data and/or the address distance data, depending on what the algorithm implementation looks like. Packages could be placed as adjacent vertices on a graph based on known delivery criteria – grouping, deadline, address proximity. That is best used if the sorting/routing components are connected. A graph could also have been used to represent the delivery destinations, with address as vertex and distance as edge weight. Either of these uses would require a different core algorithm to be implemented.

A binary search tree could be most effective if used to represent a truck's packages/destinations, with keys based on the sorting algorithm. The delivery process itself would just involve traversing the tree in order, which is very efficient.