C950 WGUPS Algorithm Overview

Tim Stewart
ID #001476583
WGU Email: tstew91@my.wgu.edu
December 31, 2021

C950 Data Structures and Algorithms II

Introduction

For the course in Data Structures and Algorithms II (C950), students are required to use the Python programming language to solve a vehicle-routing problem (VRP) involving delivery of packages. The problem has various constraints such as: maximum number of trucks on the road, maximum number of packages on trucks, package deadlines, packages that must be on a certain truck, groups of packages that must be delivered together, packages that cannot leave the hub before a certain time, and a maximum cumulative mileage of all trucks.

A. Algorithm Identification

To solve this VRP, I wrote an algorithm that uses backtracking and a nearest-neighbor strategy. The backtracking portion of the algorithm tries to send various combinations of either 1 truck or 2 trucks at different possible times. The backtracking algorithm keeps track of the cumulative mileage for various combinations of these 1-truck routes and 2-truck routes. The combination of routes with the lowest cumulative mileage is selected to be the winner. (The cumulative total mileage for my winning route was 121.8 miles.)

Heuristics are encoded into the algorithm, for example to load the trucks with packages that have deadlines first. Business logic is also applied to ensure that packages that must be on a certain truck or packages that must be delivered with other packages are handled correctly. Packages that are not able to be delivered before a certain time are skipped over until they are eligible to be delivered.

After a truck is loaded with packages, a nearest-neighbor strategy is used to find the truck route that will deliver these packages with the least mileage: From the starting point of the hub, the nearest location by distance is chosen, and from that location the nearest unvisited location to that location is chosen next. This nearest-neighbor strategy continues until all locations have been visited and the truck can return to the hub.

B1. Logic Comments

My solution uses two important algorithms, and both of them are described below.

The backtracking algorithm recursively creates lists of truck delivery routes for packages that have not yet been delivered. When the number of packages reaches zero, that list of truck routes and its total mileage is recorded. When all calls to the backtracking function are completed, the list of truck routes with the overall least total mileage is selected as the winner. Here is pseudocode for this backtracking algorithm:

function: backtracking_algorithm(list_of_routes, current_time, remaining_packages):

- If the list of remaining_packages is empty, record this list_of_routes and its total mileage in a global list_of_list_of_routes and return.

- Otherwise if there are any remaining_packages:

- Create a single route to deliver the packages remaining and add this single route to the running list of routes. Then recursively call solver_helper() with an updated copy of the running list of routes, the current time, and the new list of remaining packages.

- Also create two routes to deliver the packages remaining and add these routes to the running list of routes. Then recursively call solver_helper() with an updated copy of the running list of routes, the current time, and the new list of remaining packages.

When solver_helper() returns from all its recursive calls, the global list_of_list_of_routes is sorted by mileage and the list_of_routes with the least mileage is returned as the winner.

A nearest-neighbor algorithm is used to create the actual driving route that a truck will take to deliver its packages. Here is pseudocode for the nearest-neighbor algorithm:

function: create_truck_route_with_nearest_neighbor_algorithm(list_of_packages):

- From the list_of_packages, we create a list of locations (or stops) the truck must visit to deliver all the packages.

- The truck starts at the hub.

- While there are still some locations the truck hasn't visited yet:

    - Sort the unvisited locations by distance from our current location

    - Go to the unvisited location closest to our current location

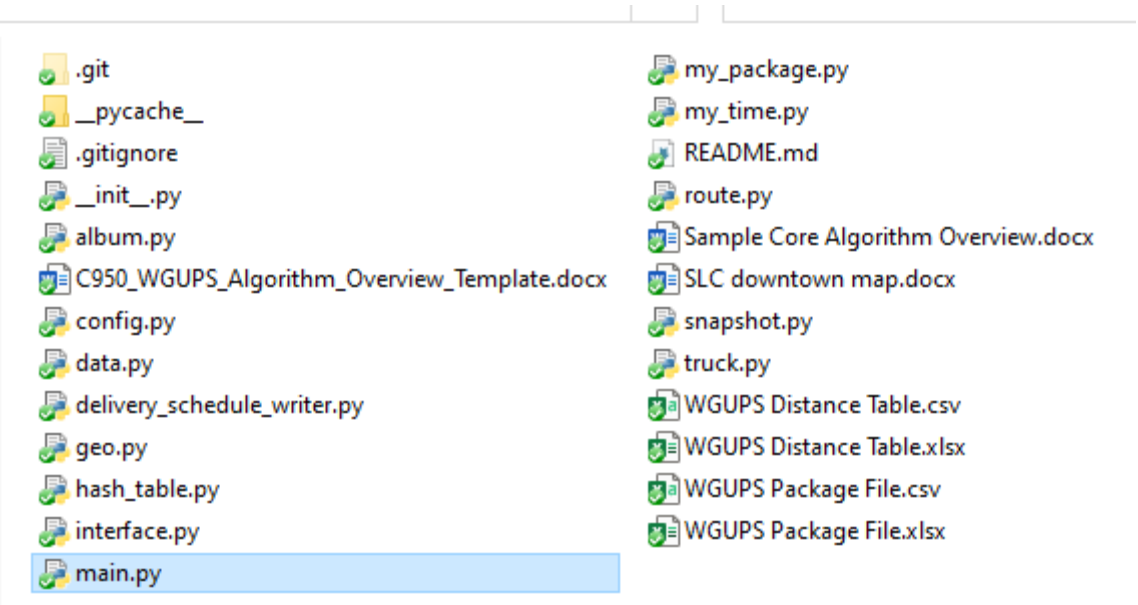
    - Mark this location as visited

    - Continue the while loop

- When all package delivery locations have been visited, the truck returns to the hub.

B2. Development Environment

I used the Visual Studio Code ("VS Code") editor on my Windows 10 personal computer to write this application in Python 3.9.7. The application was run in a Command Prompt (DOS prompt) by running "python main.py" in the directory containing all my source code files. The screenshot below shows the contents of this directory containing all my source code files:



B3. Space-Time and Big-O

Program segment: hash table with separate chaining

Insertion into my hash table is **O(1)** because the hash function is O(1) and if there is a hash collision it is still O(1) in Python to append the (K, V) to the list at that bucket, for a total cost of O(1).

Retrieval from my hash table is **amortized O(1)** because the hash function is O(1) and in the case of hash collision it is O(m) to search the list of length m at that bucket for the requested key-value pair. On average, lookup is O(n/m) for n lookups, which amortizes to O(1). In the very worst case, the cost of retrieval is O(n) if all insertions into the hash table collide to the exact same bucket and retrieval must search through a list of all key-value pairs stored in the hash table.

To support my claim of O(1) time for insertion and O(1) on average time (i.e., amortized), I quote from *Introduction to Algorithms* by the famous Gang of Four in their chapter on hash tables using separate chaining:

> If the number of hash-table slots is at least proportional to the number of elements in the table . . . searching takes constant time on average. Since insertion takes O(1) worst-case time and deletion takes O(1) worst-case time . . . , we can support all dictionary operations in O(1) time on average. (Cormen et al., 2009, p. 260)

Program segment: backtracking algorithm

At each iteration of the backtracking algorithm, as many as four truck routes are created (i.e., one truck route named truck 1, one truck route named truck 2, two truck routes named truck 1 and truck 2, and two truck routes named truck 2 and truck 1). Even when only 1 package is passed to

the recursive backtracking function, four more instances of the function will be called. Thus the time complexity for the backtracking algorithm has an upper bound of **$O(4^n)$**.

To mitigate this exponential time complexity, I also implemented a branch-and-bound (B&B) technique to discard partial solutions that cannot possibly be winning solutions or that are highly unlikely to be winning solutions. For example, partial solutions whose route mileage exceeds 140 miles are discarded. Also, partial solutions whose mileage is greater than the current best solution are discarded. Also, partial solutions that have not completed by 6 p.m. on the day of delivery are discarded. By discarding these partial solutions that already fail to meet the requirements in some way or are not likely to meet the requirements, the backtracking algorithm can spend its computing time on only the partial solutions that have the likelihood to be a winning solution.

Program segment: nearest-neighbor algorithm

To create a truck route to visit *n* locations, my nearest-neighbor algorithm must sort the list of yet unvisited locations $n - 1$ times to determine which unvisited location is closest to the truck's current location. Python 3.9's inbuilt list sorting function is based on mergesort, which has a time complexity of $O(n \log_2 n)$. Since we perform this sort $n - 1$ times, the nearest-neighbor algorithm has an overall time complexity of $O(n^2 \log_2 n)$, or simply **$O(n^2)$** since we only have to account for the fastest-growing term.

B4. Scalability and Adaptability

As the number of packages grows, the greatest effect on my application would be the increased size of the search space that the backtracking algorithm has to explore. As noted in section B3, the time complexity of my backtracking algorithm is $O(4^n)$. Even though I have mitigated some of the exponential time complexity by implementing a branch and bound (B&B) paradigm, an increase in the number of packages would not scale very well. In real-world terms, if the number of incoming packages increased greatly, then it would also make sense to increase the number of trucks in operation and to increase the number of hubs in different parts of the city. For example, if the number of packages grew from 40 to 400 (a tenfold increase), then I would want to create ten zones around the city, each with its own hub and trucks and each responsible for only 40 packages. In this way, even if the number of packages were to increase tenfold, the amount of resources to handle the packages would also be increased proportionally, allowing the level of performance and efficiency to remain similar to before.

B5. Software Efficiency and Maintainability

The application is **efficient** and easy to maintain because I have used Python classes to create objects for the trucks, packages, routes, and lists of routes. By encapsulating the data for individual trucks, packages, routes, and lists of routes I have made it easier to make changes to the software if the business requirements change. For example, if an application uses fixed variables like truck_1_mileage and truck_2_package_list, then if the application needs to be heavily rewritten to account for three trucks or thirty trucks or three hundred trucks. But if the

trucks are created as objects, then it becomes much easier to create lists or sets of as many trucks as required and adjust the algorithms and functions of the application to process the lists of trucks. Using objects also makes it very easy to make copies of objects, which is essential for the backtracking algorithm used in this application.

Another way that the application is **efficient** is that I calculate and cache the truck details and package statuses for each minute of the day so that when the user wants to navigate the trucks details and package statuses throughout the day, the application can simply load the relevant information (or "snapshot") for each minute and show it instantly. An inefficient alternative would be if the application had to recalculate the truck details and package statuses every time the user wanted to switch to a different time. But by loading the details of each minute into a large array beforehand, the application lets the user browse the truck and package activity for the day easily and without any delays.

Regarding **maintainability**, a key aspect of this application's code is that I have used very descriptive names for the variables and for the function names. In the professional software world, you will write software that someone else will have to take care of, and you will have to take care of software that someone else wrote. Because of this, we should write software in such a way that it will be easy for the next programmer to understand what is happening in the code. Using descriptive and clear variable names and function names will help a future programmer to read the code and understand right away what is happening.

Another way that the code for this application is maintainable is that I have used block comments and inline comments to help show what certain code is doing and to explain to a future programmer what some of the business logic is. Since I might not be around to coach the maintainer of this software application, these code comments are one of the best ways I can document the source code so that it is easily updated in the future.

B6. Self-Adjusting Data Structures

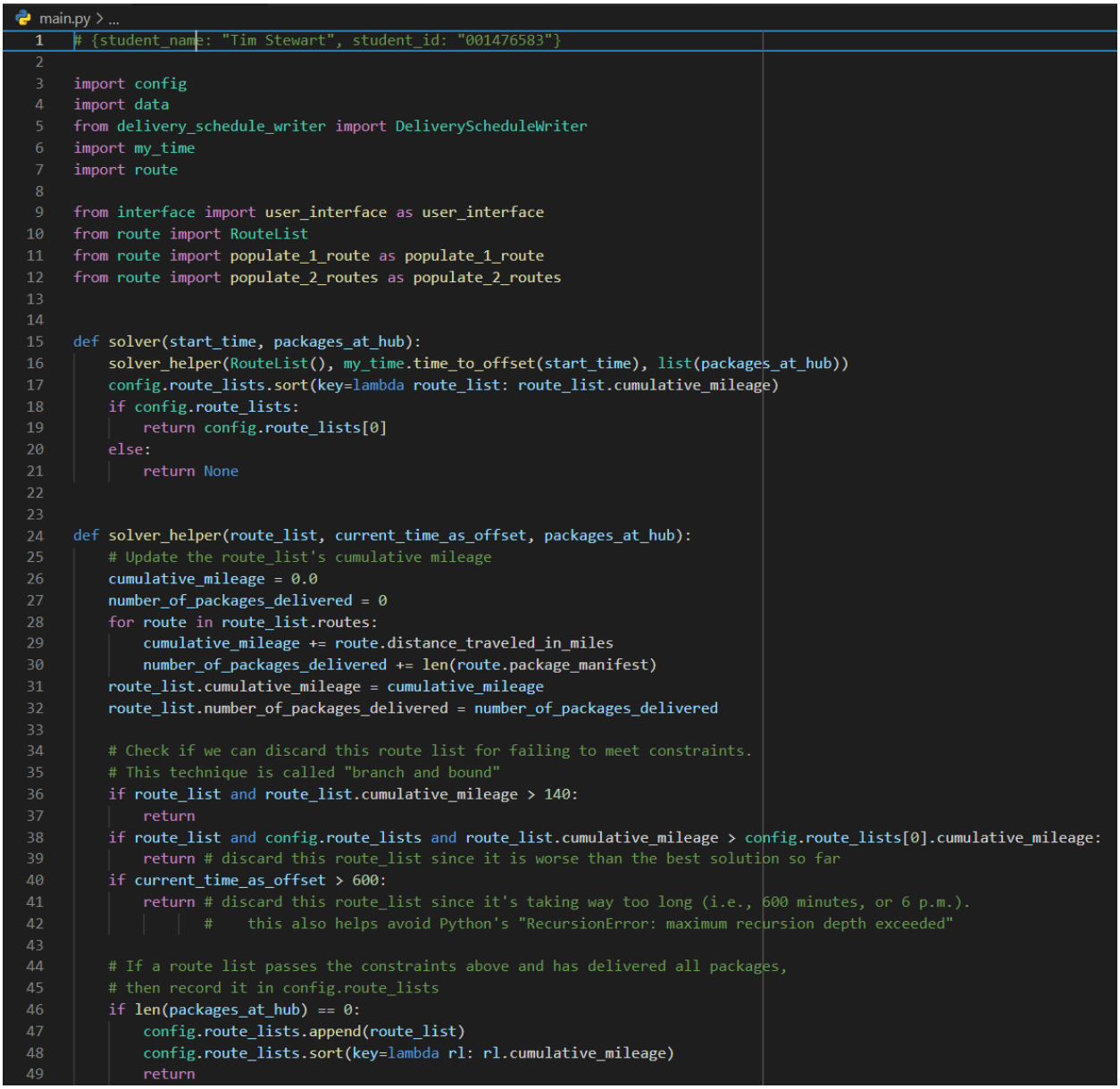
My chosen self-adjusting data structure is a simple implementation of a hash table. The main **strength** of my hash table is its very fast lookup of stored entries. The hash table does this by converting a string key into a numeric value (i.e., the hash code), which is used to store the key-value pair in an array or in a list that allows indexed access like an array. Indexed lookup is very fast for a data structure. Another **strength** of the hash table is that it uses separate chaining to handle hash collisions that might occur. If two keys hash to the same bucket, both key-value pairs are stored in a list located at that bucket.

But one of the **weaknesses** of my hash table is that once it is created it is a fixed size. A possible future improvement of my hash table implementation would be a dynamic resizing property. Dynamic resizing could compare the number of entries in the hash table to the current capacity of the hash table, and when the number of entries approaches the capacity of the hash table then a larger hash table would be created and the entries from the old hash table would be re-hashed and stored in the new hash table. Another **weakness** of my hash table is that the keys must be strings. This means it is not reliable to use objects or class instances as keys. A possible future

improvement of my hash table would be to accept objects as keys and either retrieve the object's own hash code representation or use the object's string representation as the input to the hash function. There is a lot of risk and complexity involved with using objects as keys, so this feature would need to be added carefully and thoughtfully and after a great deal of planning.

C. Original Code

Please see the main.py file included in my project submission for the entry point into my program. A screenshot of the first several lines of main.py is shown below for reference:



```python
# {student_name: "Tim Stewart", student_id: "001476583"}

import config
import data
from delivery_schedule_writer import DeliveryScheduleWriter
import my_time
import route

from interface import user_interface as user_interface
from route import RouteList
from route import populate_1_route as populate_1_route
from route import populate_2_routes as populate_2_routes


def solver(start_time, packages_at_hub):
    solver_helper(RouteList(), my_time.time_to_offset(start_time), list(packages_at_hub))
    config.route_lists.sort(key=lambda route_list: route_list.cumulative_mileage)
    if config.route_lists:
        return config.route_lists[0]
    else:
        return None


def solver_helper(route_list, current_time_as_offset, packages_at_hub):
    # Update the route_list's cumulative mileage
    cumulative_mileage = 0.0
    number_of_packages_delivered = 0
    for route in route_list.routes:
        cumulative_mileage += route.distance_traveled_in_miles
        number_of_packages_delivered += len(route.package_manifest)
    route_list.cumulative_mileage = cumulative_mileage
    route_list.number_of_packages_delivered = number_of_packages_delivered

    # Check if we can discard this route list for failing to meet constraints.
    # This technique is called "branch and bound"
    if route_list and route_list.cumulative_mileage > 140:
        return
    if route_list and config.route_lists and route_list.cumulative_mileage > config.route_lists[0].cumulative_mileage:
        return # discard this route_list since it is worse than the best solution so far
    if current_time_as_offset > 600:
        return # discard this route_list since it's taking way too long (i.e., 600 minutes, or 6 p.m.).
               #    this also helps avoid Python's "RecursionError: maximum recursion depth exceeded"

    # If a route list passes the constraints above and has delivered all packages,
    # then record it in config.route_lists
    if len(packages_at_hub) == 0:
        config.route_lists.append(route_list)
        config.route_lists.sort(key=lambda rl: rl.cumulative_mileage)
        return
```

C1. Identification Information

The first line of my main.py file includes my name (Tim Stewart) and my student id (001476583) in a Python comment.

C2. Process and Flow Comments

I have included block comments and inline comments throughout my code, in main.py and in the additional files that are imported in various places.

D. Data Structure

The self-adjusting data structure I used was a hash table. I used three instances of my hash table class to store information used by the backtracking and nearest-neighbor algorithm described in part A above.

D1. Explanation of Data Structure

I used three hash tables in my program to store important information used by my program. As shown in my config.py file, I used hash tables to store three important data relationships:

(1) the distances between pairs of locations/stops (key = a string containing names of a pair of locations, value = the distance between the locations as a float as given by the project Excel/CSV file); this hash table is defined in my config.py file and is typically accessed using an identifier of config.distances_between_pairs_ht;

(2) my stop tuples that represent package delivery locations (key = a string containing a street address, value = the associated stop tuple); this hash table is also defined in my config.py file and is typically referred to as config.all_stops_by_street_address_ht;

(3) my package class instances (key = the package id cast to a string, value = the package object that has the associated package id); this hash table is likewise defined in my config.py file and is named config.all_packages_by_id_ht.

In each hash table, the provided key in a key-value pair is cast to a string and then used to calculate a hash code that determines the array index in which the key-value pair will be stored. The value in a key-value pair could be a float value, or a named tuple, or a class instance. For the first hash table above, the value is a float for the distance. For the second hash table above, the value is an instance of my named tuple Stop (which is defined in my geo.py file). For the third hash table above, the value is an instance of my Package class (which is defined in my my_package.py file).

E. Hash Table

I developed a hash table, and the code for it is in my hash_table.py file. The insertion function for instances of my hash table is named "add()". As described in part E of the requirements, the package ID number, delivery address, delivery deadline, delivery city, delivery zip code, package weight, delivery status (e.g., delivered, en route) for each package are loaded into the

hash table. To make the data elements easier to work with, I store each package's data elements in a Package object (which is defined in my my_package.py file). The name of the hash table used to store the package data elements is all_packages_by_id_ht, which is in config.py. The package data elements are loaded into the hash table in the ingest_packages() function that is defined in data.py.

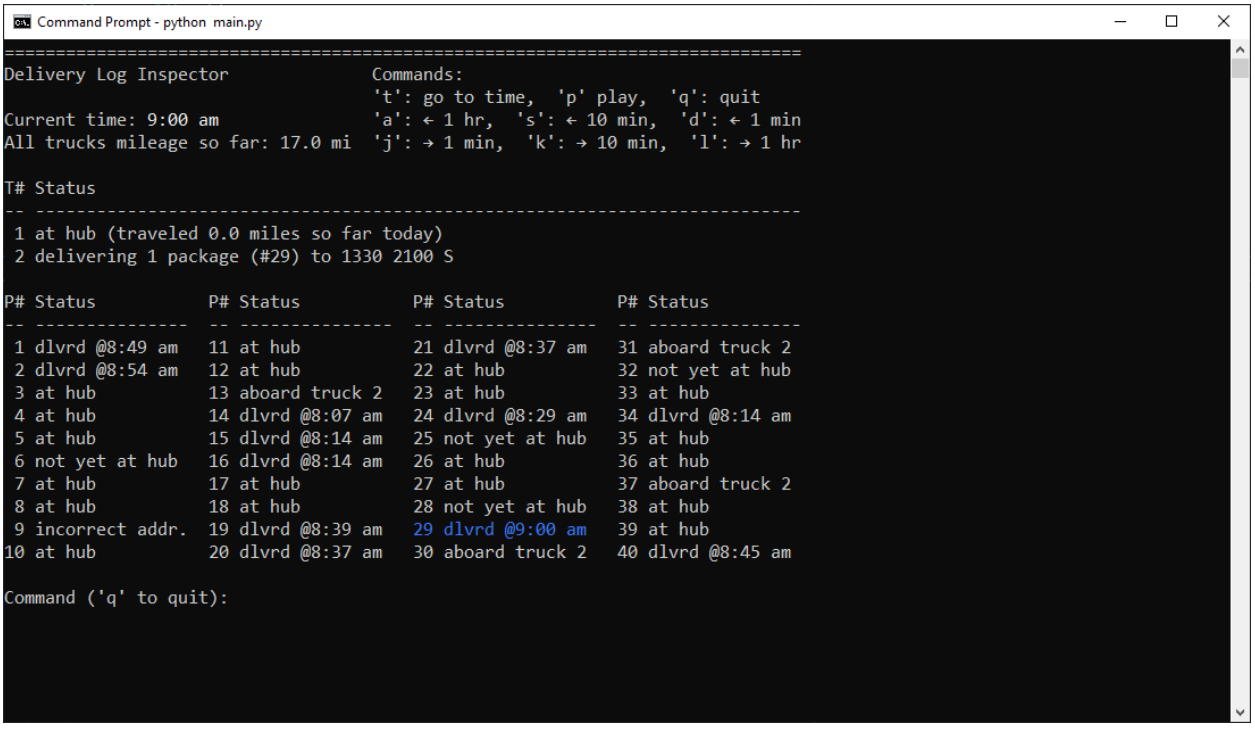
F. Look-Up Function

The look-up function for my hash table is named "get()". I retrieve package data elements from the hash table in multiple places in my program. For example, in route.py I access package data in the hash table so I can correctly group packages that have to be delivered together. As an another example, in my delivery_schedule_writer.py file I use package data in the hash table to correctly update the package delivery statuses.

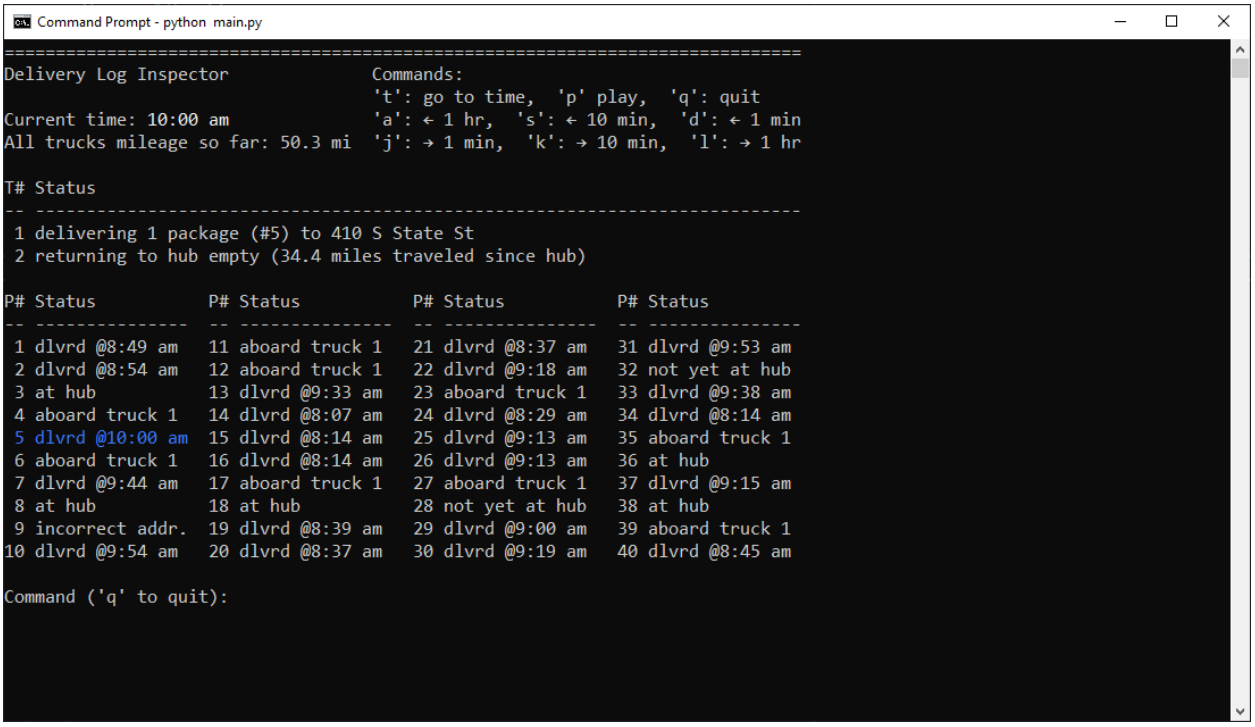
G. Interface

I wrote a text user interface (TUI) to explore and examine the delivery details for all 40 packages during the entire time period from 8 a.m. until the last truck returns to the hub after delivering the last packages. The screenshots below show the package status and truck info at 9 a.m., 10 a.m., and 12:30 p.m. As described in part G of the requirements, the interface shows the current time, the total mileage of all trucks, the delivery status of all packages including time. Package statuses include: "not yet at hub", "at hub", "aboard truck X" (i.e., en route), and "dlvrd @ time".

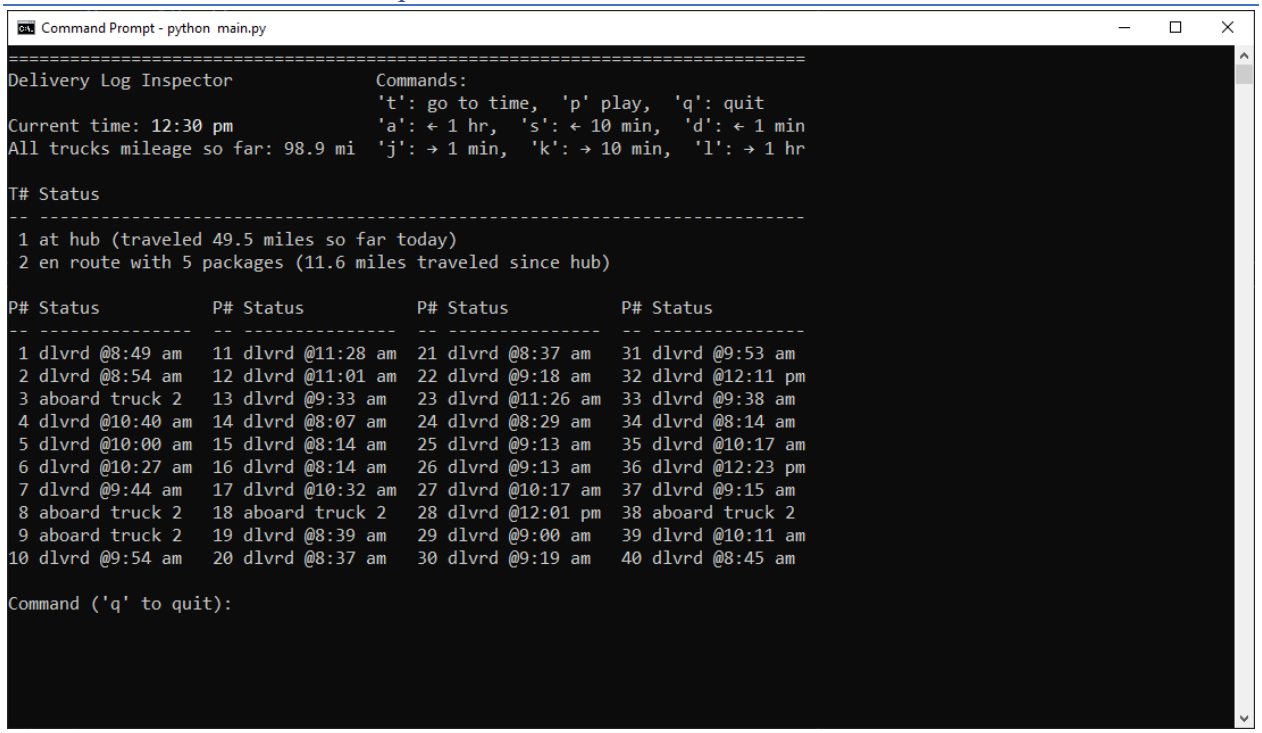
G1. First Status Check: 9 a.m.



G2. Second Status Check: 10 a.m.

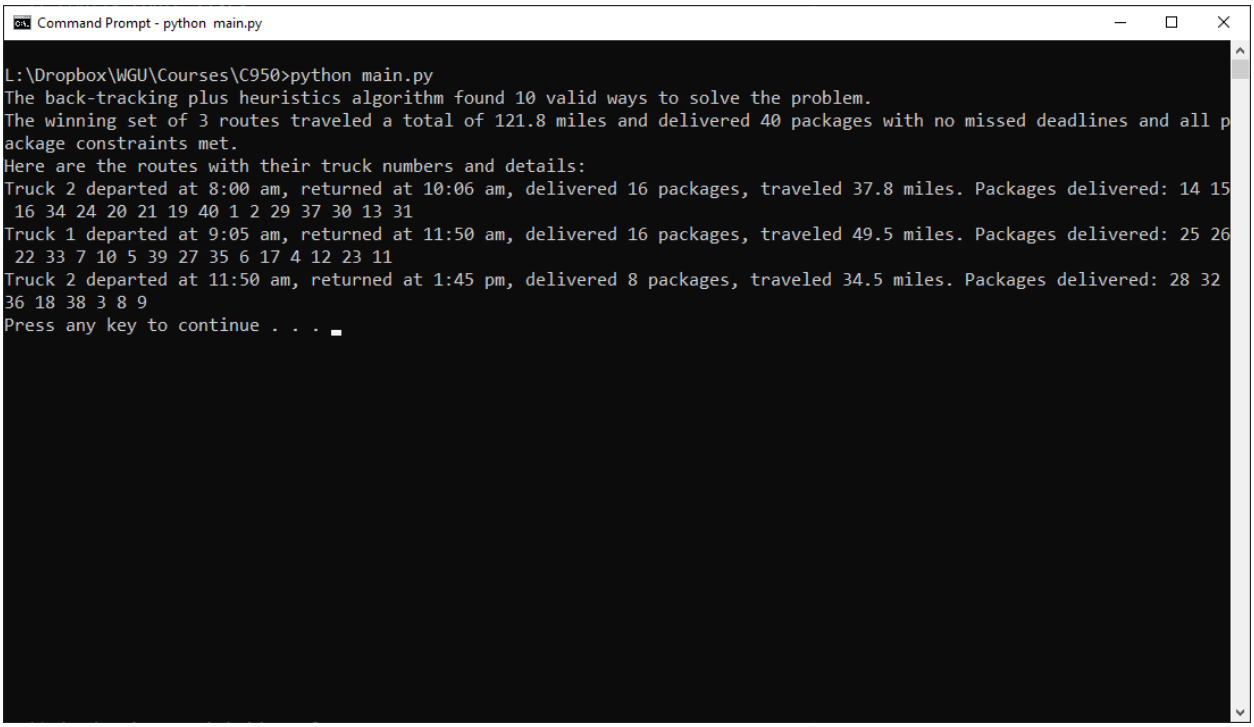


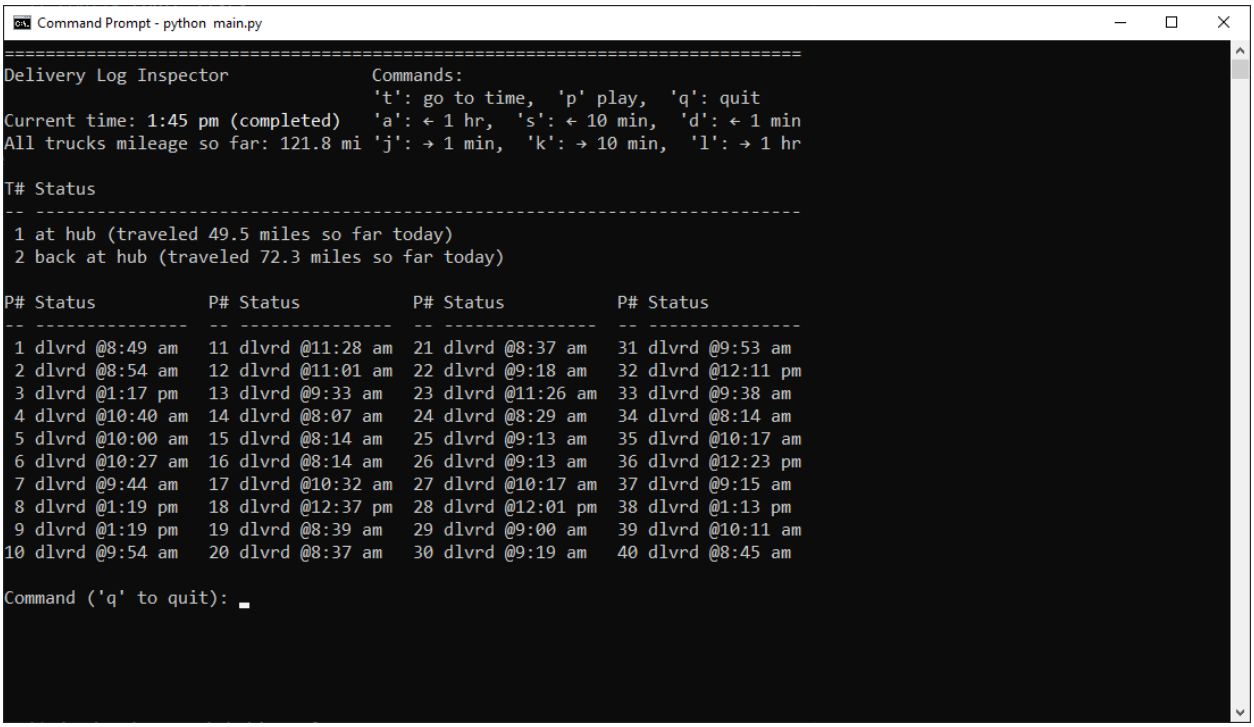
G3. Third Status Check 12:30 pm



H. Screenshots of Code Execution

The following screenshots show the successful completion of the program code. The first screenshot shows a summary of package delivery details, and the second screenshot is the user interface where the user can visit specific times of the day and inspect truck and package statuses for that time.





I1. Strengths of Chosen Algorithm

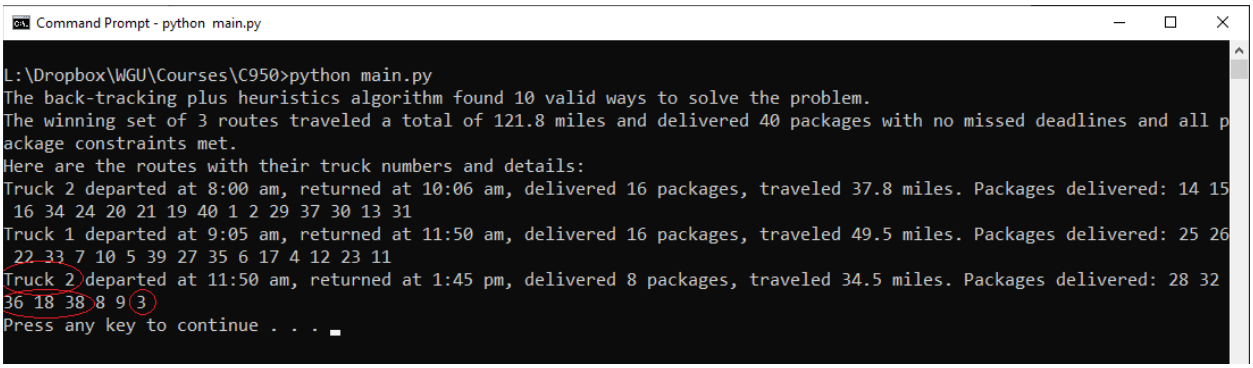
The backtracking strategy using branch and bound (B&B) and the nearest-neighbor algorithm have several **strengths**. One strength is that backtracking offers a very thorough search of the possible solutions, so if a solution exists it is often just a matter of time for the backtracking algorithm to find it. The nearest-neighbor algorithm also has its **strengths**: for example, it is a very easy algorithm to implement and it is very well researched and discussed in the academic literature. The nearest-neighbor algorithm is also typically very successful if the primary goal is to have the least distance/mileage traveled, which is the case for this C950 project.

The backtracking algorithm has **weaknesses** as well. As noted in section B3 above, the backtracking approach can consume a lot of time and space to exhaustively search all possible options. And if the number of packages that need to be delivered is increased, then the search space increases exponentially, which means the backtracking algorithm will have many more possible solutions to have to evaluate. The nearest-neighbor algorithm has its **weaknesses** as well. For example, the nearest-neighbor algorithm only considers distance in its calculation of which location to visit next. This could be a problem if a package going to a faraway location has an extremely soon deadline. (In this case, the nearest-neighbor algorithm would need to be modified to first visit high-priority locations and then visit low-priority locations, in which case it isn't truly just a "nearest neighbor" algorithm anymore.) Fortunately for this project, primarily choosing the nearest neighbor still resulted in all packages being delivered on time. But with a different set of packages that have alternative delivery deadlines, it's possible that the nearest-neighbor algorithm would not be so successful. This is why algorithms should always be verified to ensure that they are meeting the project goals and satisfying the project constraints.

I2. Verification of Algorithm

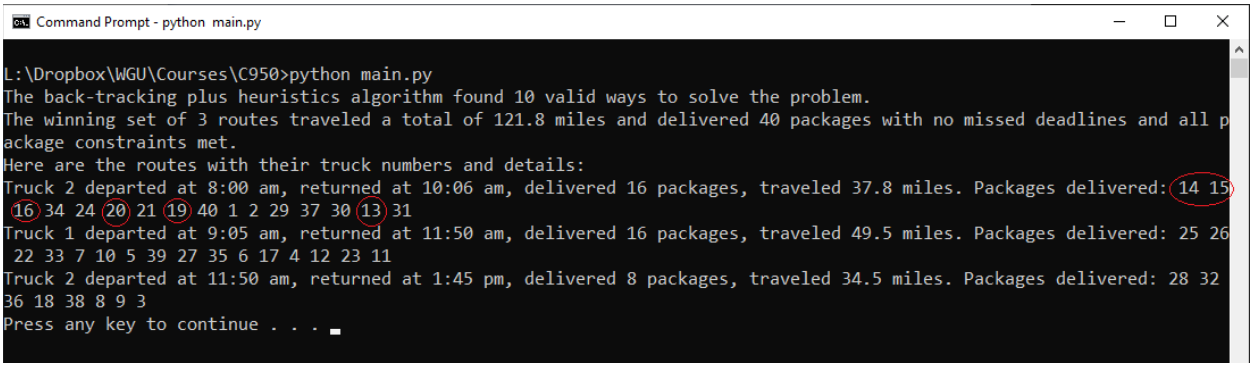
Constraints on Which Truck Packages Can Go On

Some packages had a constraint about which truck they could be delivered on. (In my source code I refer to this constraint as "truck affinity.") For example, packages 3, 18, 36, and 38 needed to be on truck 2. This constraint was coded into the algorithm, and I manually verified that the algorithm put packages 3, 18, 36, and 38 on truck 2. See the red circles on the screenshot below.



Constraints on Which Packages Must Be Delivered Together

Some packages had a constraint about which other packages they needed to be delivered with. (In my code I refer to this constraint as "package affinity.") Packages 13, 14, 15, 16, 19, and 20 needed to be delivered together. This constraint was coded into the algorithm, and I manually verified that the algorithm put packages 13, 14, 15, 16, 19, and 20 on the same truck route. See the red circles on the screenshot below.



Constraints on Package Delivery Deadlines

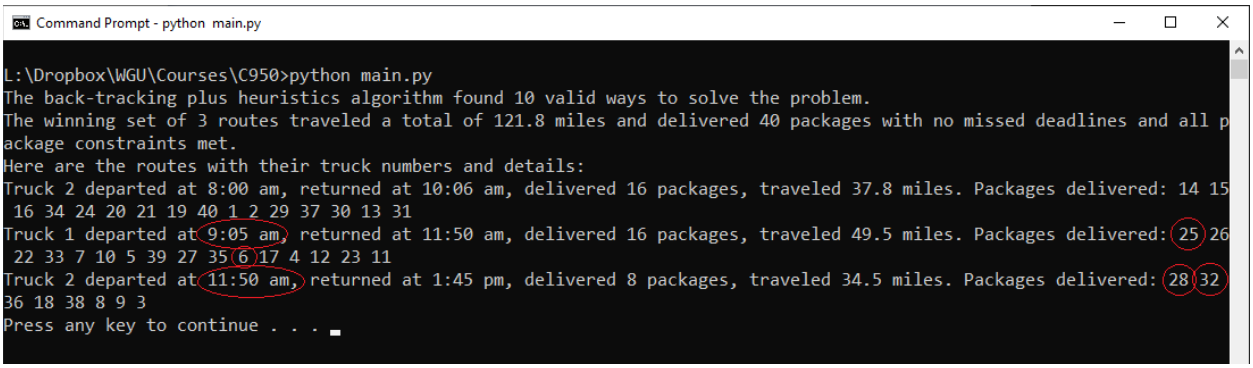
Some packages had a constraint about what time they needed to be delivered by, and this constraint was coded into the algorithm. I manually compared the package delivery deadlines to the actual delivery times produced by the algorithm, and for all packages with delivery deadlines, the algorithm delivered the package before the deadline arrived. See the table below for a comparison of package deadlins to the actual delivery time, including how many minutes ahead the deadline each package was delivered. (For example, the package with id #6 was delivered only 3 minutes before the deadline! Just in time!)

Package ID	Delivery Deadline	Actual Delivery Time	Minutes Ahead of Deadline
1	10:30 AM	8:49 AM	101
6	10:30 AM	10:27 AM	3
13	10:30 AM	9:33 AM	57
14	10:30 AM	8:07 AM	143
15	9:00 AM	8:14 AM	46
16	10:30 AM	8:14 AM	136
20	10:30 AM	8:37 AM	113
25	10:30 AM	9:13 AM	77
29	10:30 AM	9:00 AM	90
30	10:30 AM	9:19 AM	71
31	10:30 AM	9:53 AM	37
34	10:30 AM	8:14 AM	136
37	10:30 AM	9:15 AM	75
40	10:30 AM	8:45 AM	105

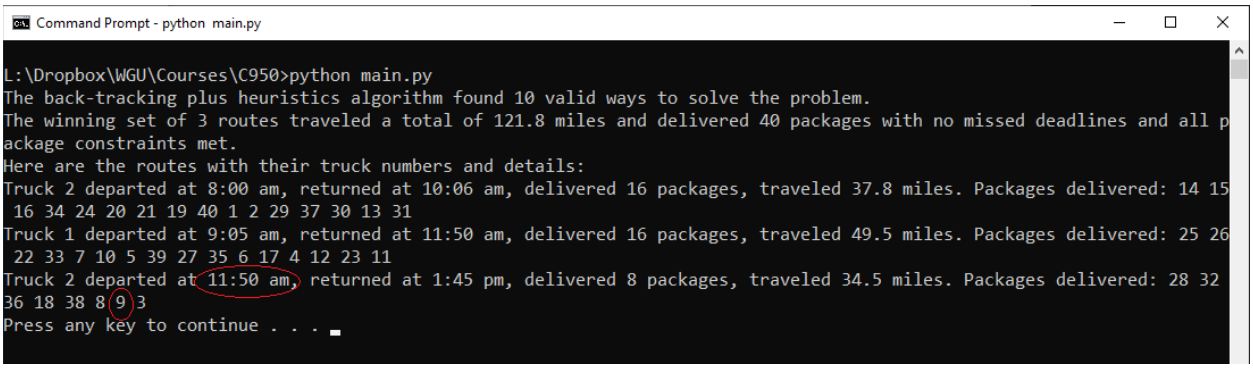
Constraints on When Packages Can Leave Hub

Some packages had constraints on when they could leave the hub. Packages 6, 25, 28, and 32 did not arrive at the hub until 9:05 a.m., and package 9 couldn't leave the hub until 10:20 a.m. because it had an incorrect address. These constraints were coded into the algorithm, and I manually verified that these packages did not go out for delivery until they were allowed to.

This screenshot shows that packages that could not leave the hub until 9:05 a.m. all left the hub at 9:05 a.m. or later:



And this screenshot shows that the package that couldn't leave the hub until 10:20 a.m. did not leave the hub until after 10:20 a.m.:



I3. Other Possible Algorithms

For this project, I used the backtracking algorithm and the nearest-neighbor algorithm, but there are other algorithms that could have been used instead. One example is a **brute-force** algorithm in which every possible combination of routes and departure times and package loads is tried and the outcome with the lowest mileage that still meets the constraints is used. Brute-force methods are impractical for all but the smallest problems. The backtracking approach that I used can be used to implement a brute-force approach, but since I also implemented branch and bound (B&B) as part of my backtracking approach, I didn't need to explore every possible candidate in order to find a winning solution. Another alternative is a **genetic algorithm** (GA) in which candidate solutions are combined and mixed to form new solutions that are then evaluated and combined with other solutions. With each iteration of a genetic algorithm, the best solutions are kept and re-combined and the worst solutions are discarded. After many iterations, the resulting solutions contain the best parts of all the previous solutions that were tried. Genetic algorithms also usually incorporate elements of randomness, which can improve routes more quickly than deterministic techniques such as brute-force and backtracking.

I3A. Algorithm Differences

My backtracking with branch and bound and nearest-neighbor algorithm is quite different from both the pure brute-force approach or the genetic algorithm approach. Brute force has the disadvantage that it continues to evaluate candidate solutions even when those candidate solutions don't meet the constraints or are worse than already existing solutions. Genetic algorithms are very different from the algorithm I used. For one thing, genetic algorithms introduce an element of randomness, and my algorithm is completely deterministic. Also, genetic algorithms combine components of past solutions to form new ones, and my backtracking algorithm treats every candidate solution as an individual entity and doesn't combine them.

J. Different Approach

If I do this project again, I would try to implement a solution using dynamic programming (DP). The dynamic programming approach breaks down a large problem into smaller subproblems that are easier to solve. So, for example, a dynamic programming algorithm might first create a solution for 1 package. Next, it would create a solution for 2 packages. Each time the new solution can build on the strengths of the previous solution. Eventually the dynamic programming algorithm reaches all 40 packages. I think this would be an interesting and challenging problem-solving algorithm to learn about, and I think it would be informative to apply dynamic programming to a Vehicle Routing Problem (VRP).

K1. Verification of Data Structure

The self-adjusting data structure I used for this project is a simple implementation of a hash table using separate chaining (which is defined in my hash_table.py file). My hash table meets all the necessary requirements. My hash table has an insertion functioned (named "`add()`") and a look-up function (named "`get()`"). My hash table stores these package data elements as described in sections E and F of the requirements: the package ID number, delivery address, delivery deadline, delivery city, delivery zip code, package weight, delivery status (e.g., delivered, en route, etc.).

K1A. Efficiency

The time needed to complete my hash table's look-up function is affected by changes in the number of packages to be delivered. For example, if the number of packages stored in the hash table greatly exceeds the hash table's capacity, the hash table will have to utilize a lot of separate chaining. This will increase the amount of time needed to perform the look-up function because although the hash code function itself is constant time, it would take linear time to search through the list of multiple entries stored in the same bucket due to separate chaining. For this reason, it is very important to intelligently estimate the number of entries that the hash table will need to store and to apply a load factor of 0.6 or 0.75 to ensure that there are plenty of buckets available for entries to hash into. For this project I used a load factor of 0.75, which means that if I plan to store information for 40 packages, then I will want to create a hash table that has an array with a size of about 53 (since $40 / 0.75 \approx 53$). The load factor of 0.75 is a widely used rule of thumb in the software development industry. For example, the API documentation for a very

widely used hash map implementation, the Java 16 HashMap class, says this: "As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs" (Oracle, 2021).

K1B. Overhead

The space needed by the hash table to store packages increases linearly with the number of packages that need to be delivered. For example, if the number of packages to be delivered increases tenfold from 40 to 400, the amount of space needed by the hash table to store the package data will also increase tenfold. Even taking into account the use of a load factor to overallocate space in the hash table array for possible entries, the increase in the amount of space used by my hash table as the number of packages increases is still linear.

K1C. Implications

Regarding the hash table used to store package data, changes to the number of trucks or the number of cities would not have an appreciable impact on either the space storage requirements or the look-up times of the hash table. If the number of trucks is increased and/or the number of cities is increased, then the package data hash table will be accessed more often because more truck routes will need to be considered, but the hash table's access time will still be amortized O(1) and its space requirements will stay the same as long as the number of packages to be delivered for the day is the same.

I do also use a hash table to store the distances between pairs of locations and also to store my named tuples for stops using the street address as the key. For these two hash tables, the implications of increasing the number of trucks and/or the number of cities would mean that I would need to initialize these hash tables to a larger capacity, so in this scenario these two hash tables' space requirements would increase. Nevertheless, the look-up times for these two hash tables would still be amortized O(1) even if the number of entries they each held increased.

K2. Other Data Structures

I used a hash table to store the package data, but other data structures could have been used instead. For example, a **simple array** could have been used to store the package data, or a **binary tree** could have been used to store the package data. Each of these data structures is quite different from the hash table data structure I used, and they also have their own advantages and disadvantages compared with the data structure used for this project, and I will describe these differences, advantages, and disadvantages in detail below.

K2a. Data Structure Differences

A **simple array** could have been used to store the package information since the package id's are integers. An array of package objects or tuples could have been created, and the packages could be added to the array so that the package id corresponds to the array index location where that package was stored. For example, the package with an id of 1 could be stored at index 1 of the array. This offers very fast lookup of elements since the package id is equivalent to the array

index, which offers O(1) lookup. In this case, the index 0 (the first element of the array) would be left empty since there is no package with an id of 0. An array has the advantage of very fast lookup if the package id (i.e., the index) is known. One way that the simple array data structure is different from my hash table data structure is that the hash table can use a key that is not just an integer. For example, if the package id's for this project included letters of the alphabet or special symbols (e.g., "package id A33-2") then this alphanumeric id would not be directly usable as an index in an array. But a hash table can easily use alphanumeric strings with symbols as keys by applying the hash function to the string and returning an integer.

A **linked list** is another data structure that could have been used to store the package data. A linked list offers advantages such as being able to easily insert new elements anywhere in the list and also efficiently delete elements that may no longer be needed. The major disadvantage of a linked list is that accessing elements of the list takes O(n) time since the list must be traveled until the desired data element is found, whereas the access time using my hash table is amortized O(1). The major advantage of a linked list is that insertions and deletions are very efficiently done on the list. However, since for this project all the package data is known at the very beginning of the project, and since packages are not inserted or deleted during the algorithm, there is little advantage to using the linked list data structure.

L. Sources - Works Cited

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd edition). The MIT Press.

Oracle Corporation. "HashMap (Java SE 16 & JDK 16)." (2021, December 31). https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/HashMap.html

M. Professional Communication

Thank you for reading my report on my C950 project involving a Vehicle Routing Problem (VRP). I look forward to reading any feedback or corrections you may suggest. By the way, if you did not get a chance to explore the 'p' ("play") command in my text user interface for this Python project, please try it out.