**Supporting Documentation for C950**

Author: Joshua Groeschl

Email: jgroesc@wgu.edu

Student Id: 01133572

**A:ALGORITHM IDENTIFICATION**

The algorithm implemented for this project is a variation of the “nearest neighbor” algorithm. The algorithm will identify the nearest location to visit next in an attempt to deliver the packages in the most timely manner. For naming purposes, we can call this algorithm the “Groeschl Routing Algorithm”. The algorithm delivers all packages with a total mileage of under 140 as per the project specifications.

**B1:LOGIC COMMENTS**

The user will utilize the main.py to directly interface with the application. The main.py creates a command line interface application using argeparse which the user then uses for passing selected flags and data into the application for displaying/evaluating the packages. The command line interface is built within the cli.py package and instantiated in the main.py file.

When the application is first run, all the data is parsed from the CSV files from within the hub package inside the CsvParse class object. The package and distance data are passed through the application after being inserted into the custom Hashtable object. The hash table is created using all native python packages and NOT utilizing the dict library native to python. The custom hashtable is located in the structures package, within the HashMap class object.

The hashtable includes lookup functions to find any object by its package id and the hashtable also includes functionality to add elements to the hashtable and remove elements as needed.

The hashtable uses Linked Lists to avoid collisions and add objects within selected “buckets” within the hashtable, these buckets take the form of an array which is extended dynamically..

All packages are objects using the Package class object which is within the hub package. These package objects are then used through the program to adjust their values that the user will then query.

After the data is parsed from the CSV files, the application loads all 3 trucks with package objects, then proceeds to route the packages to all given addresses.

The package delivery is initially executed main.py on line 38 using the trucks.run\_deliveries() method.

There are 3 truck objects and all 3 trucks execute deliveries using the truck objects start\_deliveries method which in turn utilizes the “nearest neighbor” algorithm for determining the next nearest address to route the truck to.

All the addresses for the packages are separated into an address\_hash array for faster lookup times.

The algorithm gathers the distance between the current route using euclidean distance to gauge the nearest neighbor. This neighbor's value is passed back up the stack in the application to determine which packages should be delivered next.

This continues until all packages for each truck are delivered. The application does account for multiple packages, on the same truck, that all have the same address. These packages will be all delivered close to each other and generally at the same time.

The application, depending on the user input, will then display either all deliveries to the user, a single package to the user, or all packages at a given time range.

**B2:DEVELOPMENT ENVIRONMENT**

The development environment used was a mixture of the Pycharm IDE for the software development and a Linux system (ubuntu 20.04) as the primary OS for development. I did use poetry for creating virtual python environments, though no external packages were installed.

The OS has a physical structure of 16GB RAM, 500GB SSD storage space, and the processor Intel(R) Core(TM) i5-4310M CPU @ 2.70GHz.

**B3:SPACE-TIME AND BIG-O**

The time complexity of the application is based upon the worst time of any function within the application's run-time environment. The applications Space-time complexity/big-O is O(N^2) which is tied to the method sort\_fastest\_routes() found within the optimization.py file (optimization class object). This is the primary algorithm method where the primary execution, delivery, and updates of the packages happen. As this function iterates through packages and distance data multiple times, its Big-O is O(N^2).

**B4:SCALABILITY AND ADAPTABILITY**

The application is highly scalable as the hash-table and utilized algorithms are both self-adjusting to accommodate for more than just the given set of packages.

The application will adapt to more packages as the hash-table adjusts itself to grow for additional items and the algorithm will adapt to any number of packages, delivering all of the given packages.

The algorithm will also self-adjust to determine if the packages are all designated for the same address (if they are found on the same truck that is!). When this is the case, all packages are delivered in near simultaneity.

Most major aspects of the system operate in O(N) time.

Loading trucks, parsing CSV data, creating/modifying hashtable objects, and all general delivery elements operate in O(N).

The algorithms within the optimization packages is O(N^2), and by extension the delivery methods within each truck object is O(N^2).

Some aspects of the system (such as instantiation of constructors or building the command line interface and parsing data) execute in a time complexity or Big-O value of O(1)

**B5:SOFTWARE EFFICIENCY AND MAINTAINABILITY**

The overall efficiency of the application has a BIG-O notation of O(N^2). While that is not the most phenomenal of time complexities, the application is highly modularized making for significant simplicity in maintainability.

All aspects of the application are divided into packages/modules for simplicity of updates and application improvements.

The packages, trucks, optimization algorithm, routing, command-line interface, and formulas used are all split into packages and class objects.

You can easily adjust a class object or package to extend functionality.

**B6:SELF-ADJUSTING DATA STRUCTURES**

There are several self-adjusting aspects of the system.

First, let us look at the hashtable. The hashtable is within the structures package.

The hashtable will self adjust its size to accommodate for more than just the given number of packages. The hashtable utilizes the linked list data structure to avoid hash collisions, and will allocate values into selected buckets for storage.

The hashtable also uses a hashing function to curate a hash value for each key provided when saving elements to the table.

Lastly, the hashtable will also self adjust values when they are added. If a value already exists in the table, the value is seamlessly updated without any user interaction, using the provided value to overwrite the existing value.

The algorithm has self adjusting properties intrinsically tied into its nature.

The packages are manually loaded onto trucks, however, they are then dynamically routes using the nearest neighbors algorithm to adjust routes according to the euclidean time of the nearest addresses.

The algorithm will also accommodate for multiple packages being sent to the same address, meridiem time values, and the changing locations of each package for a constant,smooth delivery service.

Now, there are some weaknesses to the data structures that were created in this application.

The hashtable has a weakness of being harder to manage or update given its extensible nature. One must use care when adjusting the code to ensure both the updating of known keys is fluid and that the extension/addition of new buckets is also seamless.

The algorithm has weaknesses in its present capabilities of route selection and could be found to perform better with further optimization of determining the nearest neigbors.

**C:ORIGINAL CODE**

The code runs smoothly and the user is not presented with any errors upon execution of the application. Several try-catch methods were utilized to prevent unwanted errors or behavior being presented to the user.

No warnings are displayed to the user unless the input from the user in improper, in which case, the user will see a message displayed stating proper utilization of the program.

**C1:IDENTIFICATION INFORMATION**

The main.py file, on line 1, has a multi-line comment denoting whom it was that wrote the application and the student ID of the creator.

Both the first name, and last name are present alongside the student ID within this comment block.

**C2:PROCESS AND FLOW COMMENTS**

Through the application, all methods have a multi-line comment above them denoting the general purpose of the function and the Big-O notation of the runtime of the function.

There are also single line comments spread through the application, notable within the optimization package within the RouteingAlgorithm class object explaining elements of the sort\_fastest\_routes method.

**D:DATA STRUCTURE**

The data structure used for this application is a hashtable.

The hashtable that was created using only native python libraries and no dictionaries, utilizes linked lists to avoid hash collisions, a hashing function to hash keys for the objects to add, a lookup function to find objects by id, and a method for adding/updating elements in the hash table.

The hashtable will auto adjust its length to accommodate for more than the given 40 packages and will dynamically update existing elements of the hashtable if an object with the same key is presented upon adding an object.

**D1:EXPLANATION OF DATA STRUCTURE**

The hashtable is a data structure utilized to store elements via key-value pairing.

An object uses a hashed key to then store a given value alongside the key for O(n) retrieval times.   
Generally, each element will reside in an individual bucket, though one can utilize various methods of storing elements in an imperfect fashion, storing elements from collisions into arrays of elements.

A collision would occur when the same key is attempted to be given to an element of the table, thus causing a hash collision as the key is hashed denoting where the store the value.

**E:HASH TABLE**

The hash table created within this application has an insertion function called “add”.

This function also serves as an insertion function or update function. If an element added to the hashtable already has the given key, that element will also be updated with the newfound value that is presented by the user, thus serving a dual purpose.

The “add” function is purely built from native python libraries using a custom hashing function for hashing keys and a system of linked list nodes for storing key-values into the hashtable.

**F:LOOK-UP FUNCTION**

The custom hashtable has a lookup function called “get” which will retrieve any element via the provided key.

To obtain objects from the hashtable, supply the get method with a key and the returned object will have all the value data that was initially stored within the values of the hashtable object.

The lookup function operaties without error and displays all stored values from the object id.

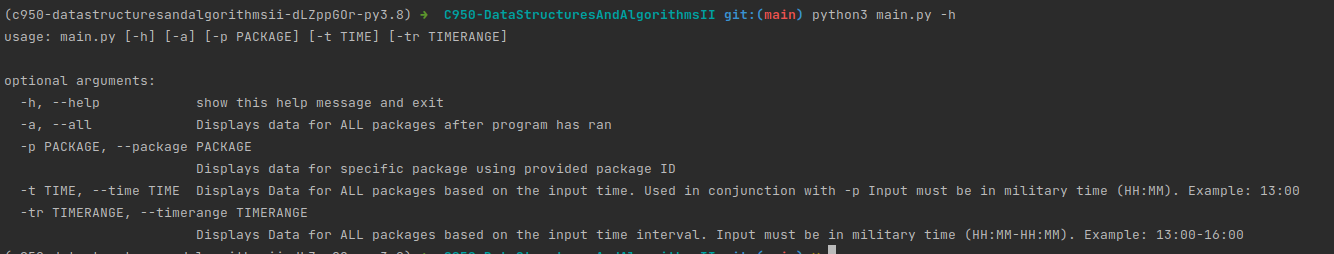
The services of linked list objects are parsed whilst performing the lookup until the node with the given key ID is found and returned to the user.

**G:INTERFACE**

The command line interface of the application presents the user with 3 options of interfacing with the application.

The user can utilize a help menu to see the commands the user has access to.

Here is a screenshot of using the help menu:



The user can view ALL packages using the -a command

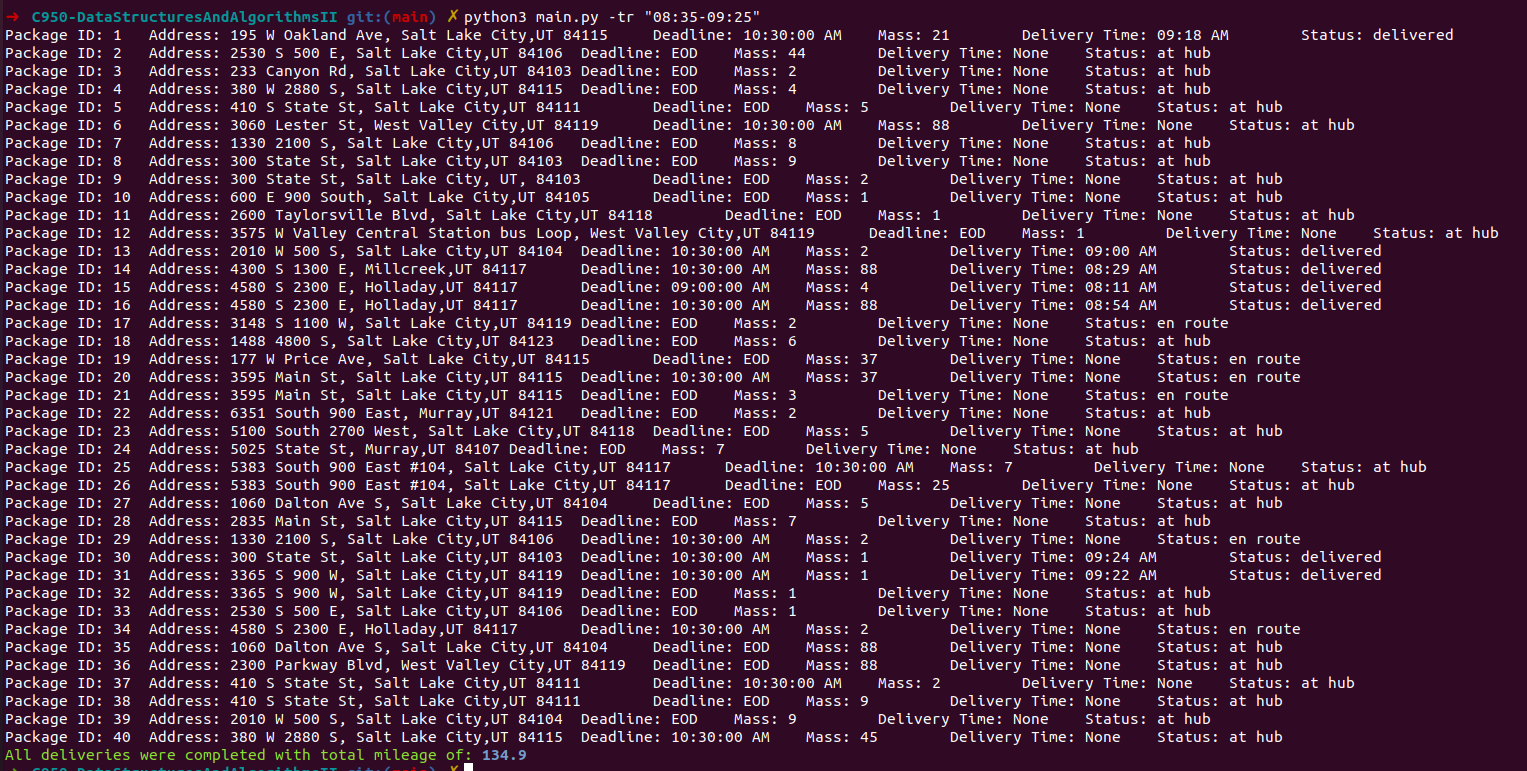
The user can view a specific package at a given time using the combination of -p and -t

The user can view specific time ranges using the -tr command.

The README.MD file states the uses of the command line application for further assistance if needed.

**G1:FIRST STATUS CHECK**

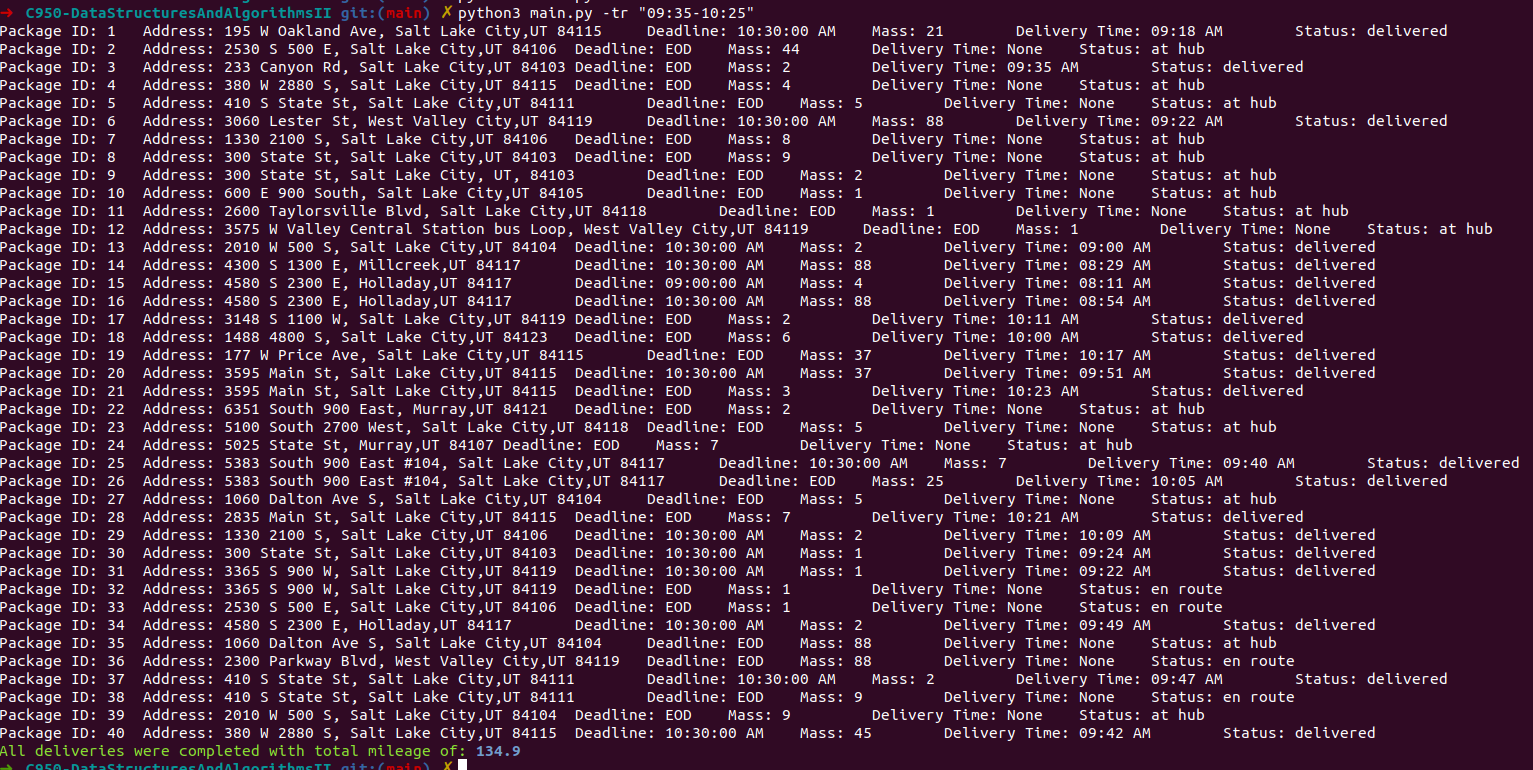
Here is a screenshot of the execution of the first status check which is showing the packages between the time range of 8:35 - 9:25 AM.



You can find the screenshot within the “screenshots” folder inside the application folder.

**G2:SECOND STATUS CHECK**

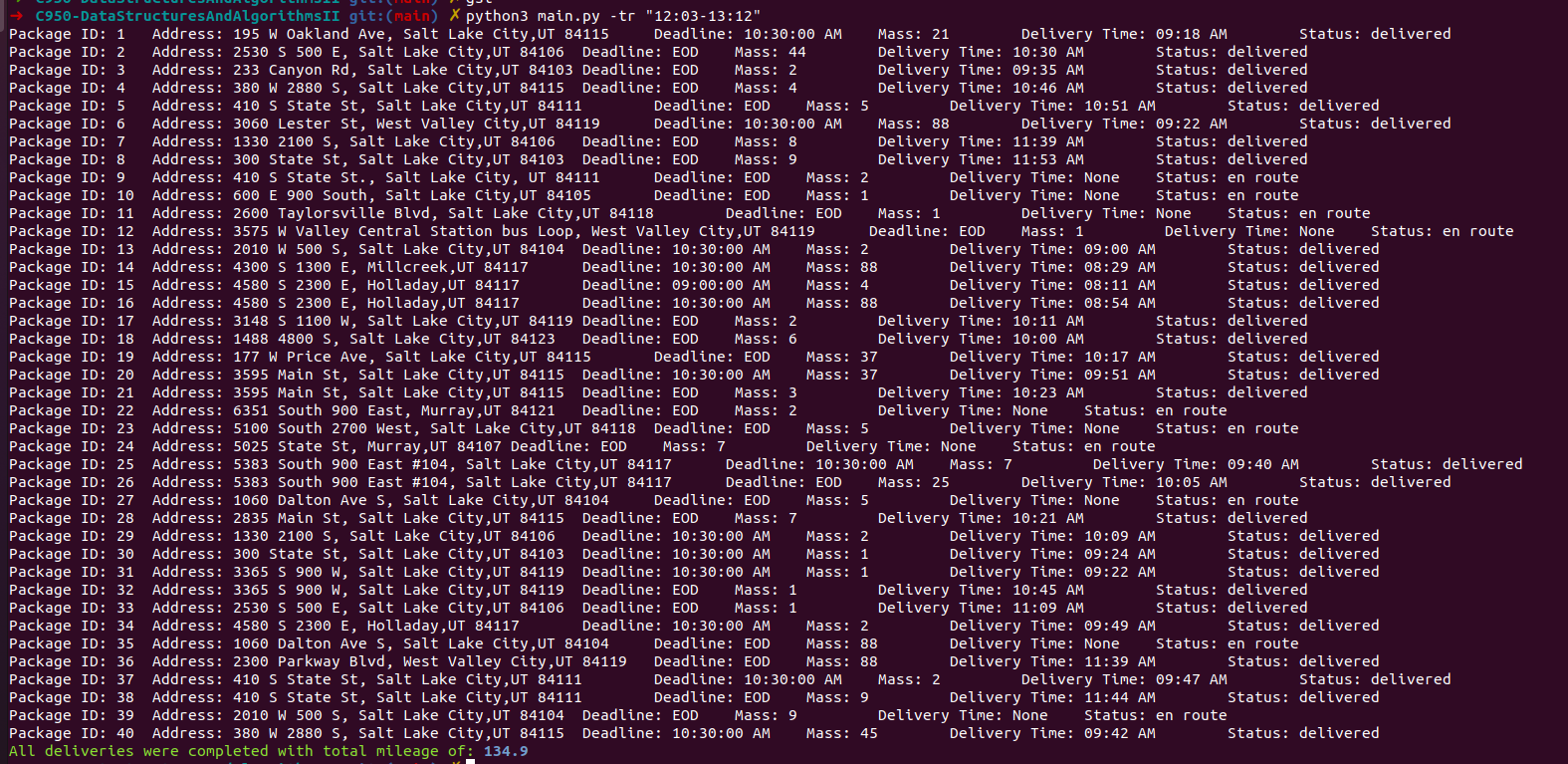
Here is a screenshot of the execution of the application for the second status check which is showing the packages between the time range of 9:35 - 10:25 AM.



You can find the screenshot within the “screenshots” folder inside the application folder.

**G3:THIRD STATUS CHECK**

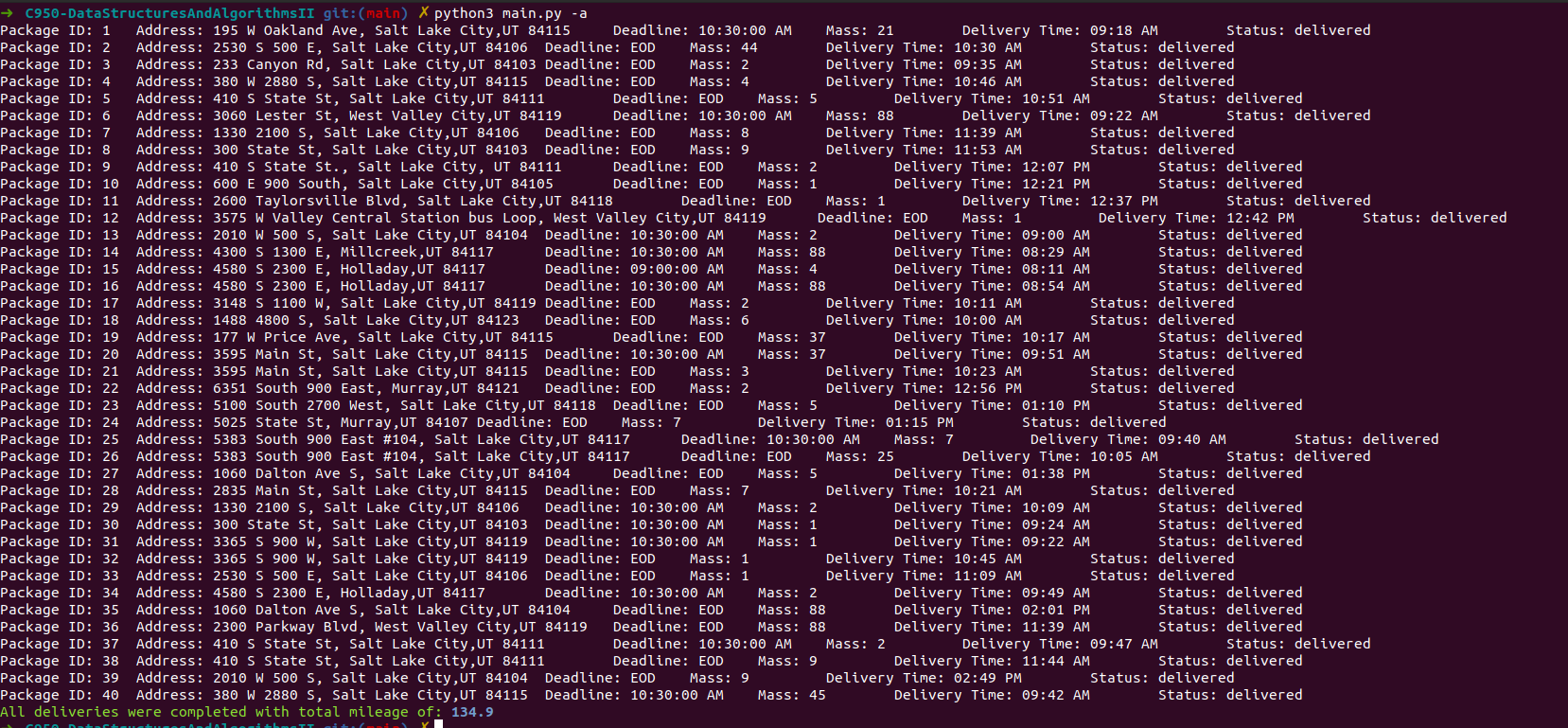
Here is a screenshot of the execution of the application for the third status check which is showing the packages between the time range of 12:03 AM - 01:12 PM.



You can find the screenshot within the “screenshots” folder inside the application folder.

**H:SCREENSHOTS OF CODE EXECUTION**

Here is a screenshot of the application after using the -a option showing all packages after the application has run.



You can find the screenshot within the “screenshots” folder inside the application folder.

**I1:STRENGTHS OF THE CHOSEN ALGORITHM**

The chosen algorithm is a variation of the “nearest neighbor” algorithm.

The nearest neighbor algorithm is common for attempting to optimize NP-Hard problems such as the traveling salesman problem which is the problem provided in this course.

The strengths of the selected algorithm is its ease of implementation and its fast execution time.

Both strengths apply to this given problem as we desire simplicity in the algorithm so we can either extend the algorithm easily or maintain the algorithm with ease.

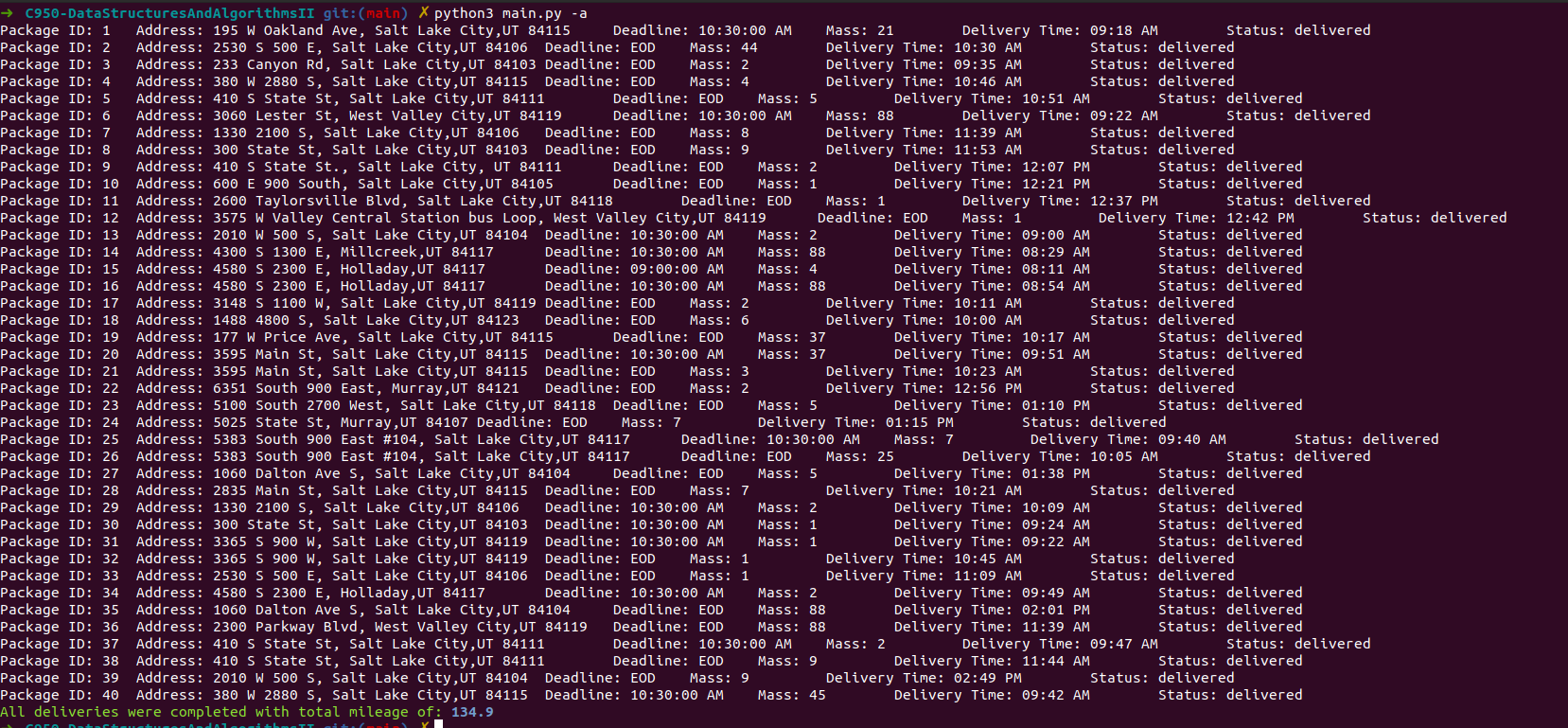
Additionally, we desire fast execution time to deliver packages in a timely manner and allow for the user to execute the code quickly.

**I2:VERIFICATION OF ALGORITHM**

As can be seen from the screenshots of the execution of the application, the algorithm delivered all packages within 134.9 total miles, beginning us under the 140 required mark for total mileage.

Additionally, we can see that all packages were delivered and all packages with deadlines were delivered before the deadline time.

Screenshot showing the successful delivery of packages, on-time, and with the total mileage of 134.9 miles.



**I3:OTHER POSSIBLE ALGORITHMS**

There are several other common algorithms that could be used for this application that would meet all criteria.

Notably, one could utilize Dijkstra's algorithm for determining the shortest path between addresses for the packages in an efficient manner.

An alternative to Dijkstra's algorithm would be the Bellman-Ford algorithm which also computes shortest paths from a node-like structure of vertices (which one could think of as being package distances).

Either of these algorithms could also be used to solve this problem and route packages effectively within the given time constraints.

**I3A:ALGORITHM DIFFERENCES**

In part I3 I stated alternate algorithms one could utilize for solving the package routing problem. The listed algorithms would be Bellman-Ford and Dijkstra's

Comparatively, each of the presented algorithms both determine the shortest paths between nodes in a graph-like structure.

The algorithm I utilize, being the nearest neighbor algorithm, is similar to the nearest neighbor algorithm (in my case using euclidean distance between address distances), and also derives a shortest path of sorts to the nearest neighbor.

Now both Bellman-Ford and Dijkstra traverse whole graphs and also use a relaxed methodology of updating nodes which approximates nodes to replace with better nodes close to the objective until a solution is reached.

The nearest neighbor algorithm is more simplistic as it does not necessarily depend on a graph like structure and simply utilizes the node values within the distance data to find the euclidean distance of the next nearest address, but does not search through the entirety of all elements to find the best routes all together as Bellman-Ford or Dijkstra’s could.

**J:DIFFERENT APPROACH**

In retrospect, an alternate approach I would take would be to look at building out a graph structure to hold data elements of all the distances between addresses. Using this graph structure I would then look to optimize the delivery of the packages using an alternate algorithm such as Dijkstra’s in order to get the delivery times down further and the total mileage to be as far below the allotted total mileage as possible.

**K1:VERIFICATION OF DATA STRUCTURE**

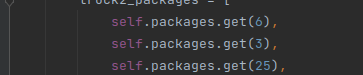
For validation of the data structure I created for this application, I am adding some screenshots to validate that the total mileage of all the trucks is below 140 and displaying an example of the “get” method to obtain a given element from the hashtable by the elements id.   
 In this case, the loading of the trucks uses the “get” function frequently to obtain the Packages from the hashtable and load them into the truck.

Supporting screenshots:

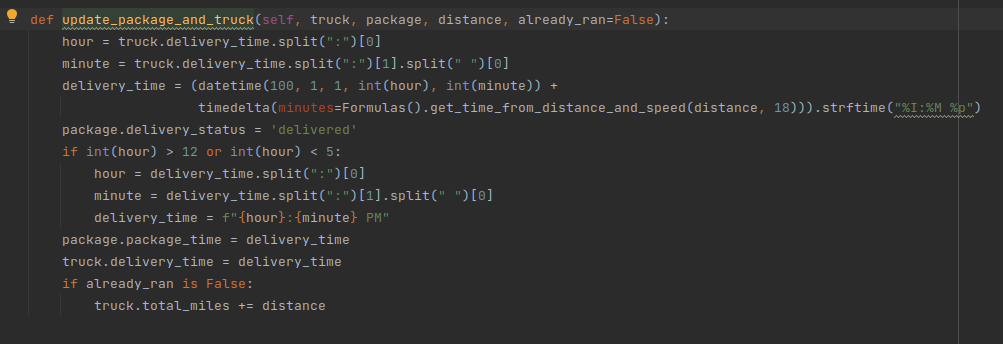
Total mileage of all trucks:



Lookup function used when loading trucks:

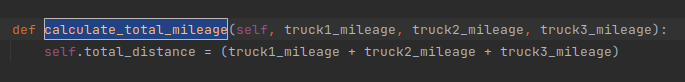


To display the effectiveness of the timekeeping elements of the application, I have a screenshot that displays the method that updates packages and the time of each truck as packages are delivered. The time elements of trucks are tracked by both the departure time of the truck and the time value of each package delivery. The method that controls the updating of the packages/truck values is the update\_package\_and\_truck method which is part of the optimization package RoutingAlgorithm class object.

Screenshot displaying updating of times and packages:

Total mileage of all trucks is calculated at the end of the application's runtime when all deliveries have been made. The calculation is done within the Trucks class object using the calculate\_total\_mileage method

Supporting screenshot:



**K1A:EFFICIENCY**

The lookup function overhead is O(N) as the linked list of elements must be searched to determine which hashed key may match the given key provided to the “get” method whilst lookup up objects.

This directly ties into the number of elements present in the hashtable (i.e. number of packages).

If the number of packages grows, as does the time required to search through all packages to find the matching key hash value.

**K1B:OVERHEAD**

For total memory usage and resource consumption, there are no constraints as to size of memory taken whilst building out the hashtable.

As the number of elements grows, the initial array of buckets grows on a continual basis, this is decent for small scale, but if there were hundreds of packages, the memory usage would grow out of normal/safe bounds.

**K1C:IMPLICATIONS**

As the number of trucks or cities grow, there is a direct response in relation to lookup time and overall memory/space usage of the application.

In both cases, more trucks would allow for more package allocation in a more diverse way, but this would also take space for a new Truck object to be created holding an array of packages, which also takes additional memory.

If more cities are added, then the overall lookup time of the algorithm for finding the “nearest neighbor” would also grow as would the arrays/dictionaries holding the cities distance data sets.

The more cities, the larger the memory allocation needed to both parse the data and store the various cities into memory, alongside direct growth of time taken for finding nearest neighbors.

**K2:OTHER DATA STRUCTURES**

An alternate usage of data structures one could employ for performing this task could be either a binary tree or graph.

Either a binary tree would work for effective storage and value lookup or a graph like structure would also suffice for storing data as vertices in the graph alongside data values.

**K2A:DATA STRUCTURE DIFFERENCES**

To compare the listed data structures in section **K2** both Binary Trees and Graphs can be used explicitly for searching. A binary tree may be better at sorting as well depending on how the Graph is implemented, though the initial creation, modification, and handling of the nodes in the binary tree could provide more overhead.

In this project, I used a custom hash table implementation which has effective techniques for adding, modifying, and getting elements from the hash table structure. The hashtable also sues key-value pairs for storing data where a binary tree would store elements within the tree as nodes and each node would have associated children, which the implementation in this case could be troublesome.

A graph consists of nodes and edges and has some functionality of adding and storing vertices on the graph which would provide similar functionality to the hash table implementation of the “get” and “add” functions.

A graph also generally uses a “neighbors” function which for this particular project would be most helpful depending on implementation of the function.

**L:SOURCES**

In writing this supporting document, I did not directly quote any published works and worked purely off of how this application was written and my own knowledge on these subjects.

If I must place down a few source examples, I would state that parsing through some Wikipedia did suffice for ensuring I was not speaking out of term.

**Article title**: Graph (abstract data type) - Wikipedia

**Website title**: En.wikipedia.org

**URL**: <https://en.wikipedia.org/wiki/Graph_(abstract_data_type)>

**Article title**: Bellman–Ford algorithm - Wikipedia

**Website title**: En.wikipedia.org

**URL**: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford\_algorithm

**M:PROFESSIONAL COMMUNICATION**

Hopefully, this documentation finds you well. I do tend to write in a longer form, so I apologize for the litany of words one may find prior to this reading.

In summation, I hope that the technical discussions above ,specifically denoting the functionality of the application, will be pertinent and easy to follow.

All primary ideologies are displayed within this document denoting general thoughts on the process and inner working of the application.

Thank you for your time in grading this application and reading of the supporting document.