A. Algorithm Identification

The project's version of the Travelling Salesman Problem is considered STSP(symmetric), because any given location can be visited directly from any other given location in a bidirectional manner. Therefore, precise algorithms that often deal with directed graphs aren't necessary for providing a valid solution to this project. The chosen core algorithm is based on the Nearest Neighbor approach. This choice does not produce the most optimal result. However, it does provide an overall optimal result for the given problem resulting in 118.1 miles round-trip for all package deliveries while demanding a smaller overall space-time complexity cost than more exact pathfinding algorithms such as Dijkstra’s. Additionally, the chosen algorithm offers both simplicity and scalability for future software enhancements.

B1. Logic Comments/Pseudocode

The pathfinding algorithm is implemented in the route.py package as follows:

Helper Methods:

1. **get_unvisited_loactions**

```
For a given list of tuples storing (x=Package ID#, y=Address Alias):
  If the Address Alias isn’t already in the unvisited locations list, append it
Return a list of unvisited locations, storing only unique delivery address aliases
```

2. **get_nearest_to_hub**

```
For a given list of unvisited locations:
  Get the distance cost from the HUB to each location
   & append to the cost list as a tuple(location=Address Alias, cost=distance from
   HUB in miles)
Assign the nearest location to HUB with the value returned from
heapq.nsmallest(1)-the smallest y(cost in miles) found in the cost list
```

3. **get_nearest_location**

```
For a given list of unvisited locations:
  Get the distance cost from the last visited location to each location
  & append to the cost list as a tuple(location=Address Alias, cost=distance from
  Last visited location in miles)
Assign the nearest unvisited location with the value returned from
heapq.nsmallest(1)-the smallest y(cost in miles) found in the cost list
```

Pathfinding Algorithm-get_route:

```
Generate the route's unique list of unvisited locations from the input route
Get the closest location to HUB-append to visited & remove from unvisited
While there are remaining locations in the unvisited list:
  Get the closest location from the current location
  Append it to visited locations & remove it from unvisited
  Append the cost of travel to total cost list(mileage required for each future delivery)
Get the distance cost from the final delivery location back to the HUB
Append to visited locations(Address Alias) & total cost list(mileage)
Return the optimized ordered route & the sum of the distances required to execute the
route by summing the mileage distances stored in the total cost list
```

B2. Development Environment

A. The program software was coded using Python 3.7.7 & only standard libraries.
B. The program was developed using the following PyCharm IDE Configurations:
   PyCharm 2020.1.4 (Community Edition)
   Build #PC-201.8743.11, built on July 21, 2020
   Runtime version: 11.0.7+10-b765.65 amd64
   VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.
   Windows 10 10.0
   Memory: 1964M
   Cores: 12
C. All required input data are stored within the project as .csv files. Screenshots of runtime & each required time status interval are stored within the project as a .pdf file & were captured via console logs. This software can be executed locally & does not rely on any network connection or server. Therefore, bandwidth & other communication protocols aren't relevant concerns for this project.

B3. Space-Time & Big-O

package.py	Space Complexity	Time Complexity	Big-O
main loop-"for row in read_csv"	O(n)	O(n)	O(n)-linear

truck.py	Space Complexity	Time Complexity	Big-O
get_total_miles_driven	O(n)	O(n)	O(n)-linear
is_available	O(1)	O(1)	O(1)-constant
get_package_count	O(1)	O(1)	O(1)-constant
Dominated By O(n)-linear			

distance.py	Space Complexity	Time Complexity	Big-O
get_distance_csv	O(n)	O(n)	O(n)-linear
map_distances	O(n)	O(n)	O(n)-linear
get_distance	O(1)	O(1)	O(1)-constant
Dominated By O(n)-linear			

route.py	Space Complexity	Time Complexity	Big-O
get_unvisited_locations	O(n)	O(n)	O(n)-linear
get_nearest_to_hub	O(n)	O(n)	O(n)-linear
get_nearest_location	O(n)	O(n)	O(n)-linear
get_route	O(n)	O(n)	O(n)-linear
Dominated By O(n)-linear			

deliver.py	Space Complexity	Time Complexity	Big-O
convert_to_delta	O(1)	O(1)	O(1)-constant
update_package_address	O(1)	O(1)	O(1)-constant
load_truck	O(n)	O(n)	O(n)-linear
get_route	O(n)	O(n)	O(n)-linear
run_delivery_route	O(n^2)	O(n^2)	O(n^2)-quadratic
first_status_check	O(n)	O(n)	O(n)-linear
second_status_check	O(n)	O(n)	O(n)-linear
delivery_status_check	O(n)	O(n)	O(n)-linear
Dominated By O(n^2)-quadratic			

hashtable.py	Space Complexity	Time Complexity	Big-O
insert_package	O(1)	O(1)	O(1)-constant
remove_package	O(n)	O(n)	O(n)-linear
search_for_package	O(n)	O(n)	O(n)-linear
Dominated By O(n)-linear			

B4. Scalability & Adaptability

The core algorithm housed in route.py is comprised of 3 main helper functions: get_unvisited_locations, get_nearest_to_hub, & get_nearest_location. This composition allows the methods to operate as pure functions, allowing for both adaptability & scalability without unwanted side effects. Additionally, the implementation of heapq.nsmallest(1) used to find the smallest distance in the cost list & return both the distance cost & location removes the need for maintaining a sorted order as would be required if a priority queue was used. Therefore, the pathfinding algorithm is able to easily adapt & scale to an increasing number of packages as needed.

B5. Software Efficiency & Maintainability
The purpose of this software is to solve an NP-Hard problem. This implies that there is no known fully scalable solution capable of doing so without requiring polynomial runtime. The given solution meets all requirements in terms of the current scope & will perform consistently for the given package dataset. However, as expansion plans become known by WGUPS, they should be communicated to the development team responsible for the software's maintenance. This solution has a very specific package sorting protocol for assigning packages to a truckload that's built around special notes, delivery time requirements, number of trucks/drivers, truckload capacity & shared addresses. This pre-sorting improves the accuracy of the pathfinding algorithm, which can be maintained & reused easily.

B6. Self-Adjusting Data Structures
As discussed above, the core algorithm inserts an unsorted list into a heapq.nsmallest(1) function as opposed to maintaining an ongoing min heap or priority queue for pathfinding. This reduces the need for in-place sorting during runtime. The primary self-adjusting data structure used in the program is a hashtable further discussed in section D.

D1. Explanation of Data Structure
The primary self-adjusting data structure & data source utilized throughout the program is a hashtable(**package_hashtable)**. In package.py, the WGUPSPackageFile.csv is read-in & each package is instantiated as an object of the Package class. Next, each of the package object's properties are appended to a list as follows:
p_id, street, city, state, zipcode, deliver_by, weight, address_alias, delivery_status
Finally, each package list is inserted into the **package_hashtable**.
The hashtable is composed of a table(list) that mimics an array, containing buckets(lists), containing lists of packages and their data attributes. The provided hashtable is initialized with 41 buckets. Its hashing function directly takes the package ID as the key & performs the modulo operation on the key using the length of the table(number of buckets). This provides a 1:1 ratio between packages & hashtable indices, as each package ID is unique. When a package is inserted, it is appended to the mapped bucket. Upon removal, the mapped bucket's lists are searched for the key (package ID) at index 0 of each list within the bucket. The lookup function is defined as **search_for_package** & uses the same methodology as the removal function, instead returning the package upon locating the key. The hashtable's design functionality enables it to avoid collisions through chaining if necessary in order to prevent the program from otherwise crashing. However, this also has the potential of reducing the runtime speed that would otherwise be provided by direct hashing. The sustainability of

the hashtable was prioritized over the guaranteed O(1) constant time performance of true direct hashing. The hashtable is maintained throughout the entirety of the program’s runtime, as it is heavily relied upon for data access in almost every major codeblock.

I1. Strengths Of The Chosen Algorithm
This solution has a very specific package sorting protocol for assigning packages to a truckload that’s built around special notes, delivery time requirements, number of trucks/drivers, truckload capacity & shared addresses. The pre-sorting improves the accuracy of the pathfinding algorithm, which can be maintained & reused easily. The given solution meets all requirements in terms of the current scope & will perform consistently for the given package dataset. Additionally, the chosen algorithm offers both simplicity and scalability for future software enhancements.

I2. Verification Of Algorithm
The algorithm can be verified by running the program & reviewing the output provided in the UI. The **console_logs_time_intervals.pdf** provides package statuses for all required time intervals, as well as a breakdown of departure time, return time, individual mileage travelled & number of packages delivered by each truck. The combined mileage of all routes is also included.

I3. Other Possible Algorithms
Another algorithm that could have been used to solve the problem is Dijkstra’s. Dijkstra’s gives the advantage of providing a guaranteed shortest path. However, this accuracy often leads to lengthy runtime. A second pathfinding algorithm that could have been utilized is A*. A* uses a combination of Dijkstra’s and Greedy algorithm approaches. It aims to provide a solution having the accuracy of Dijkstra’s & the runtime speed of a Greedy solution. However, because of the Dijkstra’s component, it too has the potential to yield a more exhaustive runtime cost than a Greedy approach such as Nearest Neighbor.

J. Different Approach
Like most software solutions, this program could be improved upon. Specifically, the sorting that takes place in the package.py file could be performed in a more automated & generalized fashion. This could be done by using sets & performing a series of unions, intersections, & merges to assign packages to each truck. This would also require functionality for handling the number of packages, such that no truckload exceeds the 16 package capacity.

K1A-C. Efficiency, Overhead, Implications
The hashtable's lookup & removal lead to a worst-case space-time complexity of O(n), because of its ability to handle collisions through chaining. This allows more than one package list to be appended into a given bucket. Collision avoidance is not necessary for storing the given data set as is. Nonetheless, the hashtable is used widely throughout the code for storing, finding, & updating package data. Examples of hashtable usage include: storing all package data in its current state throughout program execution, updating addresses, updating delivery statuses, searching for particular package delivery times, & providing system visibility to the end-user. This makes it imperative to be certain the hashtable remains intact. A simple user error altering the package data housed in the csv could result in system failure without accounting for unexpected collisions. WGUPS will also have the ability to reuse the hashtable class for inserting more than 40 packages with minimal adjustments needed. Please refer to section D1 above for further clarifications regarding the hashtable's efficiency, overhead, & implications.

K2. Other Data Structures
One alternative data structure to the custom hashtable that would adequately support the software is a standard Python **dictionary**. The underlying data structure of Python's standard dictionary is a hashtable that relies on open addressing for collision handling. Unlike the inner workings of a custom hashtable, most developers are familiar with the dictionary data structure. This makes it an ideal candidate in terms of the code's maintainability. A second potential data structure that would be sufficient for this project in its current scope is a **nested list**. Python lists are based on underlying array data structures. Given unique package numbers 1-40, & no need to insert a package list at a certain position of the outer list, we can access the data in a very similar manner to that provided by the custom hashtable.

L. Sources
No source is directly quoted or paraphrased.